Ve 280

Programming and Introductory Data Structures

Invariants; Dynamic Memory Allocation

Outline

- Representation Invariants
- Checking for Representation Invariants
- Dynamic Memory Allocation
- Dynamic Arrays
- Overloaded Constructor and Default Argument

Invariants

- An invariant is a set of conditions that must always evaluate to true at certain well-defined points; otherwise, the program is incorrect.

- For ADT, there is so called **representation invariant**.

Invariants

- A **<u>representation invariant</u>** applies to the data members of ADT.
- It describes the conditions that must hold on those members for the representation to correctly implement the abstraction.
- It must hold <u>immediately before exiting each method</u> of that implementation – including the constructor.
  - Example: insert() member of IntSet.
  - This is called **establishing the invariant**.

Invariants

Representation Invariant

- Each method in the class can assume that the invariant is true **on entry** **if**:
  - The representation invariant holds immediately before exiting each method (including the constructor), **and**
  - Each data element is truly private.

- This is true because the only code that can change the data members belongs to the methods of that class, and those methods always establish the invariant.

Invariants

Representation Invariants

- We've seen two examples of representation invariants, both applied to the private data members of an `IntSet` representation:

```
int elts[MAXELTS];
int numElts;
```

- For the unsorted version, the invariant is:
  - The first `numElts` members of `elts` contain the integers comprising the set, with no duplicates.
- For the sorted version, the invariant is:
  - The first `numElts` members of `elts` contain the integers comprising the set, from lowest to highest, with no duplicates.

Invariants

Representation Invariants

- We used these invariants to write the methods of each implementation.
- For example:

```
insert(int v)        // unsorted version
  if v not in elts // don't allow duplicates
    elts[numElts] = v  // this breaks invariant
    numElts++          // this restores it

insert(int v)        // sorted version
  if v not in elts // don't allow duplicates
    make gap in array // this breaks invariant
    elts[gap] = v  // restore elts invariant
    numElts++      // restore numElts invariant
```

Invariants

Representation Invariants

- The representation invariant plays a crucial role in implementing an abstract data type.
- Before writing a **single** line of code, write down the rep invariant!
- That tells you **how** to write each method.

- Essentially, for each method, you should:
  - Do the work of the method (i.e. insert)
  - Repair the invariants you broke

Outline

- Representation Invariants
- Checking for Representation Invariants
- Dynamic Memory Allocation
- Dynamic Arrays
- Overloaded Constructor and Default Argument

Invariants

Checking for Representation Invariants

- Invariants can also be coded, to check the sanity of the structure.
- For even moderately complicated data structures, it is worth writing a function to check for invariants.

- In the IntSet case, we **can** check to see if the array satisfies the respective invariants such as there is no duplication or the array is sorted.

Invariants

Checking for Representation Invariants

- Use sorted representation for example. We will write the following function to check the invariants:

```
bool strictSorted(int a[], int size)
  // REQUIRES: a has size elements
  // EFFECTS: returns true if a is sorted
  //              with no duplicates
```

- How can we tell if an array is sorted with no duplicates?
  - If size ≤ 1, the array is sorted with no duplicates.
  - If size > 1, then the array must satisfy
    a[0] < a[1] < … < a[size-1]

Invariants

Checking for Representation Invariants

```
bool strictSorted(int a[], int size)  {
  // REQUIRES: a has size elements
  // EFFECTS: returns true if a is sorted
  //          with no duplicates

  if (size <= 1) return true;

  for (i=0; i<size-1; i++){
    if (a[i] >= a[i+1]) {
      return false;
    }
  }
  return true;
}
```

Invariants

Checking for Representation Invariants

- Writing these "checker" functions is very useful – you can use them for **defensive programming**.
- So, you can write a **private** method to check whether all invariants are true (**before exiting**, or after entering, each method):

```
bool repOK();
// EFFECTS: returns true if the
//          rep. invariants hold
```

- For the sorted version, `repOK` would be:

```
bool repOK() {
  return strictSorted(elts, numElts);
}
```

Invariants

Checking for Representation Invariants

- Next, add the following code right before returning from any function that modifies any of the representation:

```
assert(repOK());
```

- If you are truly paranoid, you can write the same line at the beginning of every method, too; this checks that the assumption the method relies on is true.

Outline

- Representation Invariants
- Checking for Representation Invariants
- Dynamic Memory Allocation
- Dynamic Arrays
- Overloaded Constructor and Default Argument

Memory

Dynamic Allocation

- So far, the data structures we've **built** have all had room for "at most N" elements.
  - E.g., the two `IntSet` implementations could have at most `MAXELTS` distinct integers.
- Sometimes, a **fixed-sized** structure is reasonable.
  - E.g., a deck of cards has 52 individual cards in it
- However, there is no meaningful sense in which "a set of integers" is limited to some particular size.
  - No matter how big you make the set's capacity, an application that needs more will eventually come along.

Memory

Dynamic Allocation

- We have seen two types of variables so far:
  1. Global Variables
  2. Local Variables

1. Global Variables
   - These are defined anywhere outside of a function definition.
   - Space is set aside for these variables **before** the program begins execution, and is reserved for them **until** the program completes.
   - This space is reserved at **compile time**.

Memory

Dynamic Allocation

2. Local Variables
   - Local variables are defined **within a block**.
     - These include function arguments.
   - Space is set aside for these variables when the relevant block is entered, and is reserved for them until the block is exited.
   - This space is reserved at **run time**, but the size is known to the compiler.

- Since the compiler must know how big all of these variables will be, it is **static** information, and must be declared by the programmer.

Memory

Dynamic Allocation

- It turns out that there is a **third** type of object you can create, a "**dynamic**" one.
- They are dynamic in the sense that the compiler:
  - Doesn't need to know **how big it is.**
  - Doesn't need to know **how long it lives.**

- For example:
  - Our implementation of `IntSet` should be able to grow as big as any client needs it to grow, subject to the limits of the physical machine.
  - The `IntSet` should last as long as the client needs to use it, after which the **client** should be the one responsible for **destroying** it.

Memory

Dynamic Allocation

- Dynamic object creation is accomplished through the **dynamic storage management** facilities in the language.

- These facilities consist of two operations:
  - **`new`**: Reserve space for an object of some type, initialize the object, and return a pointer to it.
  - **`delete`**: Given a pointer to an object **created by new**, destroy the object and release the space previously occupied by that object.

Memory

Dynamic Allocation – new

```
int *ip = new int;
```

- This creates new space for an integer, and returns a pointer to that space, assigning it to `ip`.
- Note that we didn't do anything to initialize the integer – it could be any random integer value.
- We can initialize it to a specific value with an "initializer":

```
int *ip = new int(5);
```

- We can also `new` a class type. E.g.,

```
IntSet *isp = new IntSet;
```

- The **constructor** is called. `isp` points to an empty `IntSet` object with zero elements.

Memory

Dynamic Allocation – delete

- If objects were created by `new`, they can be destroyed by `delete`:

```
delete ip;
```

- This **releases the space**.

- Note: you cannot **`delete`** an object not created by **`new`**!

```
int a = 5;
int ip = &a;
delete ip; // Error
```

Memory

Dynamic Allocation – delete

- We can also destroy instances of class that were created by new:

```
delete isp;
```

- In this specific case (deleting an `IntSet`), the `IntSet` consists of only "ordinary" types (`int`s, arrays-of-`int`s), so we don't need to do anything to destroy it.
- That won't be true of all class-destruction events!
- Just as we have **constructors** to create objects, sometimes we will need **destructors** to properly destroy them.
  - We will see this later …

Memory

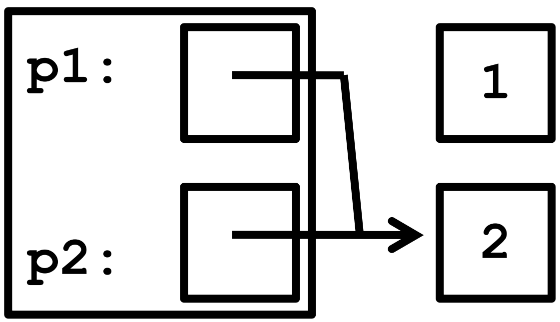
Dynamic Allocation – delete

- Note that a dynamic object's lifetime is completely under the control of the program – it lives until it is **explicitly** destroyed.
- This is true even if you "forget" the pointer to the object.

```
int *p1 = new int(1);
int *p2 = new int(2);
p1 = p2;
```

Any problem?

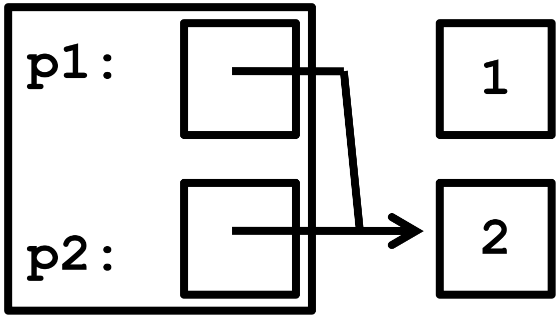
- This leaves us with:



Memory

Dynamic Allocation – delete

```
int *p1 = new int(1);
int *p2 = new int(2);
p1 = p2;
```



- Two pointers point to the object "2", and **none** to the object "1".
- There is no way to release the memory occupied by "1“.
- And worse:

```
delete p1;
delete p2;
```

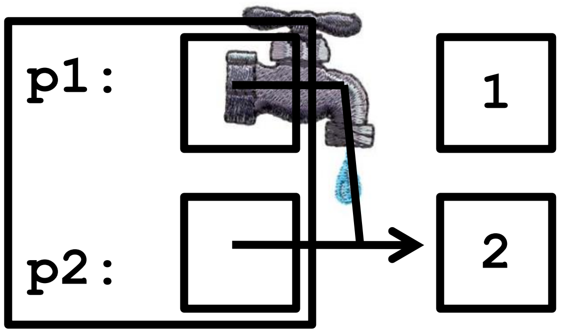
"releases" the memory reserved for "2" **twice**.

- This is surly not good!

Memory

Dynamic Allocation – delete

- Note there is an important difference between the lifetime of a pointer variable and the lifetime of the object it points to!



- In the previous example, exiting the block that defines p1 causes the local object p1 to vanish, but the dynamic object it points to remains!
- This leaves us with an allocated dynamic object that we have no means of recycling. This is called a **memory leak**.
- If memory leaks occur often enough, your program may reach a point where it can no longer allocate new dynamic objects.

Checking Memory Leak

- Tool to use: `valgrind`
- Command:

```
valgrind --leak-check=full ./program <args>
```

  - Function: search for memory leaks and give details of each individual leak.

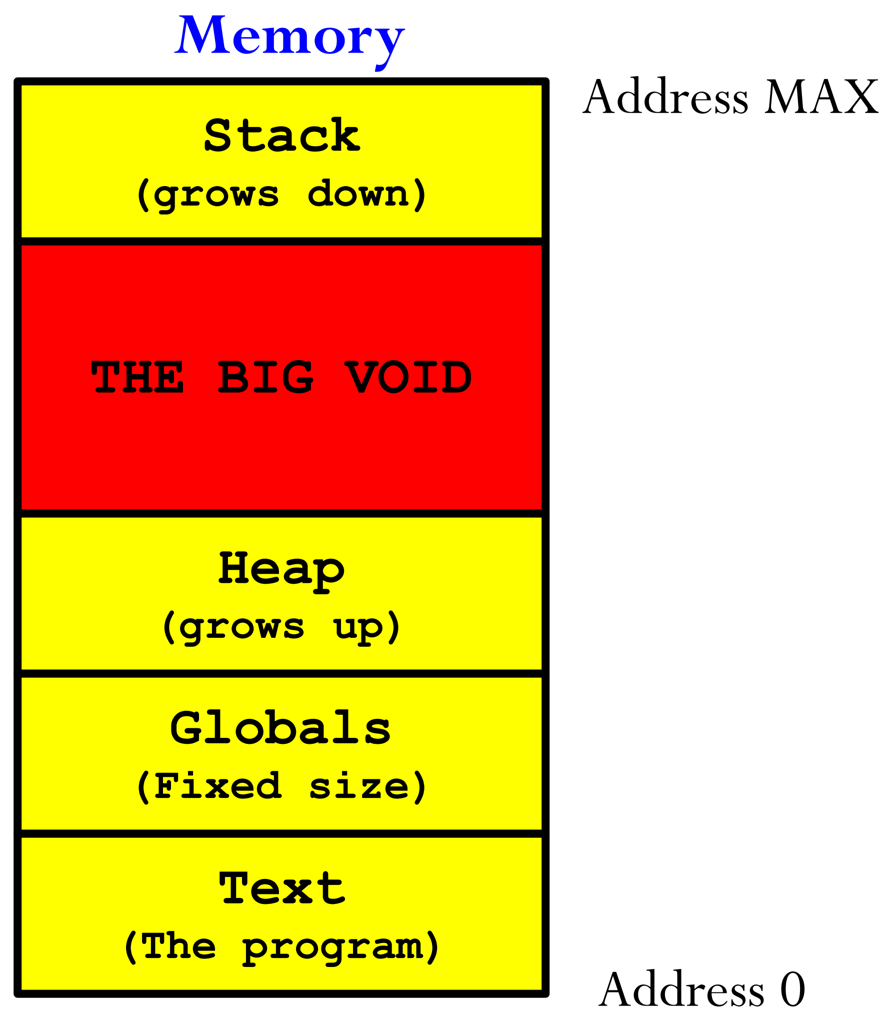
  - To install, type the command:

    ```
    sudo apt-get install valgrind
    ```

Memory

The heap

- The space for objects created via `new` comes from a location in memory called the **heap**.
  - Stack is for function calls.



Outline

- Representation Invariants
- Checking for Representation Invariants
- Dynamic Memory Allocation
- Dynamic Arrays
- Overloaded Constructor and Default Argument

Dynamic Arrays

Creating

- So far, the things we create **dynamically** have sizes **known** to the compiler.
  - E.g., `int`, `IntSet`
- However, one can also create objects whose sizes are **unknown** to the compiler, by creating **dynamic arrays**.
- Syntax:

```
int *ia = new int[5];
```

  It creates an array of five integers in the heap, and stores a pointer to the first element of that array in `ia`.
- The size is put inside []. It could even be a variable.

```
int n = 20;
int *ia= new int[n];
```

This is different from static array.

```
int ia[n]; // Wrong!
```

Dynamic Arrays

Freeing

- Freeing an array works slightly different than freeing a single object:

```
delete[] ia;
```

- If you allocate an **`array-of-T`**, you **absolutely must** use the `delete[]` operator, and **not** the "plain" `delete` operator.
- They are completely different:
  - Mixing them leads to undefined behavior.

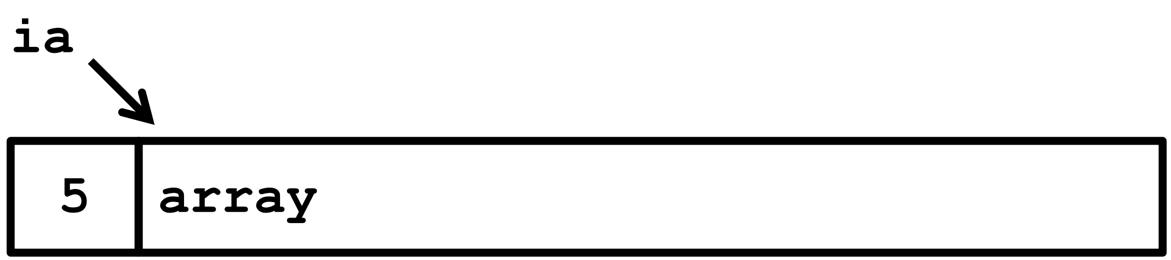
Dynamic Arrays

Freeing

- When the new operator sees it is allocating an array, it stores the **size of the array** along with the array.
- It does this by carving out space for the array, plus a bit extra:

- The space **before** the array records the number of elements in the array, in this case, 5:

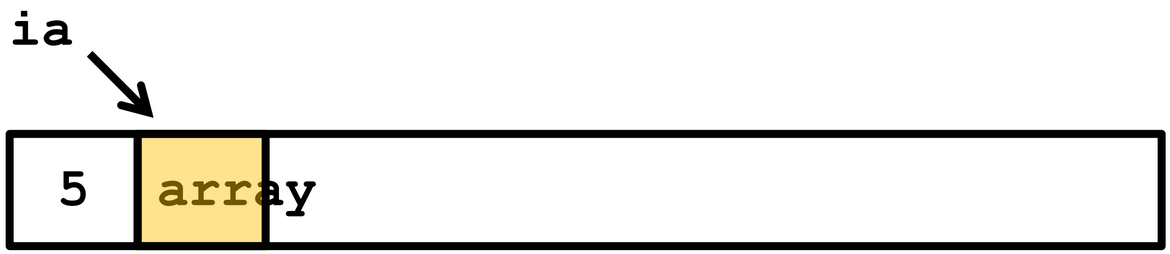
- And a pointer to the beginning of the array is returned:



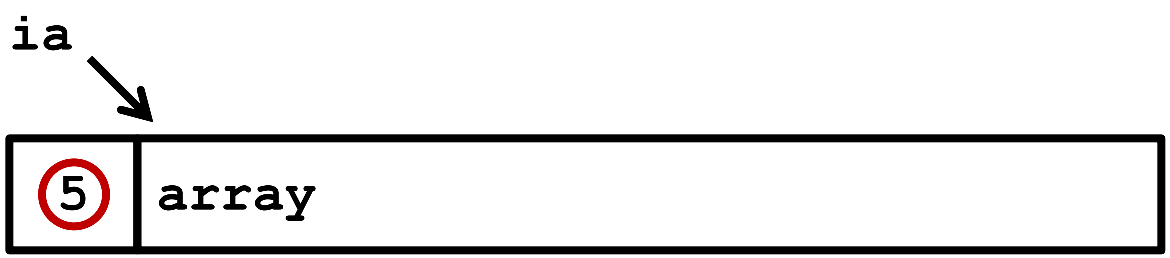
Dynamic Arrays

Freeing

- Now, if you just `delete ia`; the `delete` operator thinks it is only returning enough space for a single integer to the heap.



- The `delete[]` operator knows to look "just before" the pointer, to see **how many** elements to return to the heap.



Dynamic Arrays

Building a new `IntSet`

- We now build a version of `IntSet` that allows the client to specify how large the capacity of the set should be.

- The data elements will change slightly:

Rather than hold an array explicitly, we have a pointer that will (eventually) point to a dynamically-created array.

```
class IntSet {
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  ...
};
```

- **`sizeElts`** **tells us the size of the allocated array (which is not necessarily** **`MAXELTS`)**
- **`numElts`** **still tells us how many elements there actually are.**

Dynamic Arrays

Building a new `IntSet`

- We'll base our changes on the **unsorted** implementation.
- The methods are mostly unchanged. There is a new **default constructor**:

```
IntSet::IntSet() {
  // Allocate the "default-size" array
  elts = new int[MAXELTS];
  sizeElts = MAXELTS;
  numElts = 0;
}
```

Dynamic Arrays

Building a new `IntSet`

- Alternatively, we can write the default constructor using the **initialization syntax**:

```
IntSet::IntSet(): elts(new int[MAXELTS]),
    sizeElts(MAXELTS), numElts(0)
{
}
```

Outline

- Representation Invariants
- Checking for Representation Invariants
- Dynamic Memory Allocation
- Dynamic Arrays
- Overloaded Constructor and Default Argument

Dynamic Arrays

Building a new `IntSet`

- In addition to the default, we can write an "alternate constructor".
- It has the same name as the default, but a different type signature:

```
class IntSet {
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
 public:
  IntSet();    // default constructor
    // EFFECTS: create a MAXELTS capacity set
  IntSet(int size); // constructor with
                    // explicit capacity
    // REQUIRES: size > 0
    // EFFECTS: create a size capacity set
};
```

Function Overloading

- This is called **function overloading**.
  - Two different functions with exactly the **same name**, but **different argument count** and/or **argument types**.

a) int average(int a, int b);

b) double average(double a, double b);

c) int average(int a, int b, int c);

- Compiler tells which function to call based on the actual argument count and types.

average(2, 3); → int average(int a, int b);

average(2, 3, 5); → int average(int a, int b, int c);

average(2.0, 3.0); → double average(double a, double b);

Dynamic Arrays

Building a new `IntSet`

- The alternate constructor creates an array of the specified size:

```
IntSet::IntSet(int size): elts(new int[size]),
    sizeElts(size), numElts(0)
{
}
```

Dynamic Arrays

Building a new `IntSet`

- Since the compiler knows the argument count and types, it can pick the "right" constructor when a new object is created.

- For example:

```
IntSet is1;   // No arguments
              // Call default constructor
IntSet is2(200); // Integer argument
                 // Call alternate
```

Dynamic Arrays

Building a new `IntSet`

```
IntSet::IntSet(int size):
   elts(new int[size]),
   sizeElts(size),
   numElts(0) {
}
```

```
IntSet::IntSet():
   elts(new int[MAXELTS]),
   sizeElts(MAXELTS),
   numElts(0) {
}
```

- Notice that the two constructors are nearly identical:
  - The only difference is whether we use `size` or `MAXELTS`.
  - Otherwise the code is duplicated.
- This is bad: when we find ourselves writing the same code over and over, we should try to use parametric generalization.

Dynamic Arrays

Building a new constructor

- One way to solve this problem of duplicate definitions is to use **default argument**.
- We can define **just one** constructor, but make its argument **optional**.
- First, we have to re-declare the constructor in IntSet:

```
class IntSet {
   int *elts;     // pointer to dynamic array
   int sizeElts; // capacity of array
   int numElts;  // current occupancy
public:
   IntSet( int size = MAXELTS );
     // EFFECTS: create a set with specified
     //     capacity.  It defaults to MAXELTS if
     //     not supplied.
};
```

Default Argument

- `int add(int a, int b, int c = 1)`
  - The default value of c is 1.

- Using default arguments allows you to call the function with different number of arguments.

```
add(1, 2) // a = 1,b =2,c = 1 (default value)
add(1, 2, 3) // a = 1, b = 2, c = 3
```

- There could be multiple default arguments in a function, but they must be the last arguments.

```
int add(int a, int b = 0, int c = 1) // legal
int add(in a, int b = 1, int c) // illegal!
```

Dynamic Arrays

Building a new constructor

- Then, we implement the constructor in a same way as before.

```
IntSet::IntSet(int size):
   elts(new int[size]), sizeElts(size),
   numElts(0)
{
}
```

Don't add "**= MAXELTS**"!

References

- **Problem Solving with C++ (8th Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 9.1 Pointers
  - Chapter 9.2 Dynamic Arrays
  - Chapter 11.4 Classes and Dynamic Arrays
  - Chapter 10.2 Constructors for Initialization (pp. 560-570)
  - Chapter 6.3 Default Arguments for Functions (pp. 344-345)