Dynamic Memory and Recursive Data Structures

Martin Emms

November 13, 2018

- often need data structures whose size can only be determined when the program is run
  - eg. a parse-tree for an input sentence
- all arrays, structs, classes lack this property: their size is fixed at compile time
- dynamic memory allocation is the answer to this
- vectors and strings are implemented using dynamic memory allocation

- a computer’s memory a sequence of memory cells
- each stores 8 bits (a byte)
- each has an address: the address is the position in the sequence
- typically somewhere between 0 and roughly 4000 million (approx $2^{32}$)
- for all programming language there is a great deal of ‘behind the scenes’ activity involving addresses
- C and C++ are unusual in putting addresses ‘in front of the curtain’, in the hands of the programmer: lines of C and C++
  - can mention addresses
  - store them in variables
  - pass them as parameters to functions

Outline

Pointers and Dynamic Allocation
  - Pointers and Address introduced
  - structs and classes and pointers
  - Dynamic Allocation

The linked list
  - Linking with pointers
  - Extending the list algorithmically: push
  - Shrinking the list algorithmically: pop
  - Destructors and dynamic allocation
'pointer' = new kind of value, the bits involved function as a memory address

pointer types: if a memory cell stores a value of type \( T \), the type for a pointer to that cell is \( T^* \)

eg. if \( x \) is to store pointers to int values:
\[
\text{int } *x;
\]

dereferencing a pointer: to get at the pointed-to value, use \(*\), e.g.:
\[
\text{cout } \ll \text{*x;}
\]

▷ will print the int which is at the address stored in \( x \)
▷ not the same as just printing \( x \)'s value

there are two sources of pointer values

new Type

gives an address from heap chosen at run-time eg.
\[
\text{float } *\text{ptr;}
\text{ptr } = \text{new float;}
\]

&Identifier
give address of an identifier chosen at compile time eg.
\[
\text{float } x;
\text{float } *\text{ptr;}
\text{ptr } = &x;
\]
\[
\text{vector<float> v;}
\text{vector<float> } *\text{ptr;}
\text{ptr } = &v;
\]

these slides are concerned with the new situation

De-referencing a pointer

▷ With any pointer value, you can apply the \(*\) operation, which is called dereferencing the pointer.
▷ suppose the \( \text{ptr} \) has value \( a \) (an address), then

▷ Generally \( *\text{ptr} \) will give the data at address \( a \)
▷ The exception is in \( *\text{ptr} = \) (lefthand side of an assignment): here the address \( a \) is made the target of the assignment
int *x;
x = new int; // x contains some heap address
*x = 3;       // a LHS use of *x, so 3 is stored at that address
// not a LHS use of *x, so get value from address ie 3
cout << *x << " the int at the address " << endl;
cout << x << " the address itself " << endl;

Dynamic Allocation

- There is memory area called the heap. A call to new Type picks out an unused address in the heap, suitable for storing an instance of Type.
- It returns the address of the storage. Eg. 'Give me a pointer to a fresh instance of my class A':
  A *a_ptr;
a_ptr = new A(p1,..,p2);
- The heap object persists long after control passes out of the block of code which contained new. It persists until there is a call to delete
  delete a_ptr;
- This makes the memory locations which were in use available for use by a call to new

Dynamic Memory and Recursive Data Structures

- Pointers and Dynamic Allocation

Structs and classes and pointers

- Suppose a class A. You can have a variable storing the address of an instance of A, ie. an A 'object'. The variable would be declared:
  A *ptr;
- There is a shorthand for accessing the members of a pointed-to object, as illustrated by:
  ptr->age = 4;    // same as (*ptr).age
  ptr->print();   // same as (*ptr).print()

Dynamic Memory and Recursive Data Structures

- Pointers and Dynamic Allocation

The linked list

- Linking with pointers

Having one object point to another

- An item-on-a-list can be defined with:
  class list_item {
    public:
      string info;
      list_item *next;
  };

- So list_item objects contain:
  - An info member of string type
  - A next member, pointing to data which is also a list_item object.
  - This is recursive: recursion bottoms out in an object whose next is the NULL pointer
3 linked list_item objects

3 list_item objects linked together, representing the sequence "a" "b" "c":

```
| info = "a" | next ----> | info = "b" | next ----> | info = "c" | next ---->
------------- | ----------- | ----------- | ----------- | ----------- | -----------
```

Building the list by hand

```
main ()

list_item *first, *second, *third;
third = new list_item;
second = new list_item;
first = new list_item;
third->next = NULL;
second->next = third;
first->next = second;
third->info = "c";
second->info = "b";
first->info = "a";
```

Example: dynamic allocation on demand

In preceding examples dynamic allocation was overkill: the program had decided that there will be exactly 3 parts of the list. Now write functions for extending and shrinking.

```
class list_item {
public:
 string info;
 list_item *next;
};

class list {
public:
 list(void);
 list_item *thelist;
 void push(string i);
 string pop(void);
};

list::list(void) {
 thelist = NULL;
}

void list::push(string x) {
 list_item *n;
 n = new list_item;
 n->info = x;
 n->next = thelist;
 thelist = n;
}

push(x) transforms this:

```
| info = sn | next ----> .. -----------
------------- | ----------- | ----------- | ----------- | ----------- | -----------
```

into this:

```
| info = x | next ----> .. -----------
------------- | ----------- | ----------- | ----------- | ----------- | -----------
```

◮ thelist now points to a new list_item
◮ the old value of thelist is the value of next in the new list_item.
◮ we have have not moved any data about
How the list grows step by step

To begin with the list is a null pointer.

```
thelist ----> ------------
 | info = "c"  |
 | next ------|
```

to push "c"

```
set pointer n to point to a new list_item object
```

```
n = new list_item;
```

```
set info field to "c"
```

```
n->info = x;
```

```
set next field of this object to current value of thelist – the NULL pointer
```

```
n->next = thelist;
```

```
set thelist to be n, ie. to point to the dynamically allocated object
```

```
thelist = n;
```

```
thelist ----> ------------
 | info = "c"  |
 | next ------|
```

Shrinking the list with `pop`

Now consider the member function `pop`. `pop` transform this:

```
string list::pop(void) {
    string x;
    list_item *tail;
    x = thelist->info;
    tail = thelist->next;
    delete thelist;
    thelist = tail;
    return x;
}
```

this is done with:

```
string list::pop(void) {
    string x;
    x = thelist->info;
    thelist = thelist->next;
    return x;
}
```

because the list_item-sized chunk of memory which was the top should be freed up

so `tail` holds `thelist->next`, so that `thelist` can be freed, before `thelist` is reset to the value saved in `tail`.

∆ thelist now points to what was the 2nd item on the list

∆ the old value of thelist is forgotten
Destructors and dynamic allocation

a method to empty the list:

```cpp
void list::empty() {
    for(int i = size; i > 0; i--) {
        pop();
    }
}
```

each call to pop leads to delete

```cpp
string list::pop(void) {
    string x;
    list_item *tail;
    x = thelist->info;
    tail = thelist->next;
    delete thelist;
    thelist = tail;
    size--;
    return x;
}
```

when a list object is finished with, list.empty() really should be called
but who is going to remember to do that?
by defining a destructor, the compiler will remember that

```cpp
list::~list() {
    empty();
}
```