dynamic memory and recursive data structures

martin emms

november 30, 2016

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

- 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
'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:

```cpp
int *x;
```

dereferencing a pointer: to get at the pointed-to value, use *, eg.:

```cpp
cout << *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.
  ```cpp
  float *ptr;
  ptr = new float;
  ```

- & Identifier give address of an identifier chosen at compile time
  eg.
  ```cpp
  float x;
  float *ptr;
  ptr = &x;
  ```

  ```cpp
  vector<float> v;
  vector<float> *ptr;
  ptr = &v;
  ```

these slides are concerned with the new situation

**HEAP SITUATION**

<table>
<thead>
<tr>
<th>var</th>
<th>cell content</th>
<th>cell address</th>
</tr>
</thead>
<tbody>
<tr>
<td>float *ptr</td>
<td>5000</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

**HEAP AREA**

<table>
<thead>
<tr>
<th>3.1415926</th>
<th>5000 &lt;-++++</th>
</tr>
</thead>
<tbody>
<tr>
<td>5004</td>
<td></td>
</tr>
<tr>
<td>5008</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
</tbody>
</table>

**De-referencing a pointer**

- With any pointer value, you can apply the * operation, which is called dereferencing the pointer.
- suppose the ptr has value $a$ (an address), then

```cpp
Generally *ptr will give the data at address a
```

```cpp
The exception is in *ptr = (lefthand side of an assignment): here the address a 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 Allocation

- 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()
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 ------||----------- |

main ()

```c
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";
```
How the list grows step by step

To begin with the list is a null pointer.

\[
\text{thelist} ---- | |
\]

to push "c"
set pointer \(n\) to point to a new list_item object

\[
\text{set info field to "c"} \quad n->\text{info} = x;
\]

\[
\text{set next field of this object to current value of thelist – the NULL pointer} \quad n->\text{next} = \text{thelist};
\]

\[
\text{set thelist to be } n, \text{ ie. to point to the dynamically allocated object} \quad \text{thelist} = n;
\]

\[
\text{thelist} ----> ------------ | \text{info = "c"} | \text{next} -----> |
\]

Shrinking the list with pop

Now consider the member function pop. pop transform this:

\[
\text{thelist} -----> ------------ | \text{info = x} | \text{next} -----> .. -------------- |
\]

\[
\text{this is done with:}
\]

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

\[
\text{this simpler version is not right:}
\]

```
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.
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();
}
```