Pointers to structs, dynamic data structures

- looked at “new” to dynamically allocate and deallocate arrays
- sometimes want to allocate/deallocate single items
- structs and classes particularly interesting, since they can also contain pointers to other dynamically-allocated items
- allows us to create lists, trees, graphs, and other data structures that incrementally grow and shrink over time
- we'll start with the basic syntax for new/delete on single items, then syntax specifically for structs, then move on to dynamic data structures in general

New and delete for a single item

- to allocate a single item we leave off the [size], e.g.

```
int* iptr = new int;
float* fptr = new float;
string* sptr = new string;
```

- to access the item we use the *ptr, e.g.

```
cin >> (*iptr);
cout << (*iptr);
```

- to delete the single item we use delete without [], e.g.

```
delete iptr;
delete fptr;
```

New and delete for a single struct

- new and delete work for structs (or arrays of structs) too

```
struct Point {
   int x, y;
};
int main() {
   Point *p = new Point;
   Point *arrayPts = new Point[10];
   // use them as long as we need them
   delete p;
   delete [] arrayPts;
}
```

Syntax 1 for ptrs to structs: (*p).f

- we need * to get at the struct we're pointing too, and a . to get at the individual field
- I highly recommend the use of (*ptrname).fieldname for clarity (for you and the compiler!)

```
Point *p = new Point;
if (p != NULL) {
   cout << “Enter x and y values” << endl;
   cin >> (*p).x >> (*p).y;
}
```

Syntax 1 for structs of structs

- suppose we have a Circle struct with a dynamically allocated Point as a field

```
struct Point { int x, y; };
struct Circle { Point *pt; float radius; };
```

- say we dynamically allocate a circle and its point:

```
Circle* cptr = new Circle;
if (cptr != NULL) (*cptr).pt = new Point;
```

- accessing the x,y fields directly is kinda ugly:

```
cin >> (* ((*cptr).pt) ).x;
cin >> (* ((*cptr).pt) ).y;
```

Syntax 2 for ptrs to structs: p->f

- an alternative syntax is supported for accessing fields through a pointer to a struct, using p-> instead of (*p).f
- previous example becomes cleaner

```
Circle* cptr = new Circle;
if (cptr != NULL) cptr->pt = new Point;
cin >> cptr->pt->x;
// i.e. go from cptr into pt, and from there into x
// much easier to follow than syntax 1 approach
// cin >> (*((*cptr).pt)).x;
```

Chaining items together

- how about a struct with a pointer to its own type

```
struct Item {
    string name;
    float price;
    Item* next; // can point to another item
};
```

- each item has some data fields but can also access another (dynamically allocated) item
- can use this to string together any number of items, as long as we keep track of whichever one is at the front

Setting up a list of items

- allocate items dynamically
- use NULL to indicate no item present
- keep track of front item in list

```
int main() {
   Item* front = NULL;  // no items at first
   front = new Item; // try to allocate first
   if (front != NULL) { // in case new fails cuz out of memory
      cin >> front->name >> front->price;
      front->next = NULL; // no items after this one, so far
   }
```

List of items: adding to front

- create a new item, make it's next point to old front item, then make front point to the new item

```
Item* tmp = new Item;
if (tmp != NULL) {
   cin >> tmp->name >> tmp->price;
   tmp->next = front;
   front = tmp;
   cout << “Added “ << tmp->name << “:” << tmp->price << endl;
}
```

- can repeat as often as we like to keep adding more

Printing all the items

- start from front, print each item's data, stop at NULL

```
Item* current = front;
cout << “Current items in the list are:” << endl;
while (current != NULL) {
   cout << current->name << “:” << current->price << endl;
   current = current->next;
}
```

- goes through whole chain from front to end, one item at a time, with current pointing to whichever one we're on now

List of items: remove from front

- can chop an item out, do something with it, and delete it
- get a temporary pointer to it, adjust front to bypass it, then use and delete via the temporary pointer

```
Item* current = front;
if (front != NULL) {
   front = front->next; // bypass current
   current->next = NULL; // make sure it's detached from rest
   // do whatever with current
   delete current;
}
```

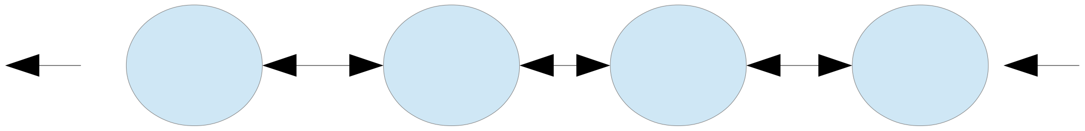
A note on memory leaks

- if we forget to delete an item we've chopped out of our list, and don't keep a pointer to it, then we lose the ability to ever delete it
- the memory is thus unrecoverable (until the program ends)
- if this happens every time we remove an item, and we do a lot of adds/removes over time, then gradually our program is chewing up system memory
- eventually we run out of memory and get a crash
  - how many odd crashes have you seen in games after you have kept them running for a long time...?

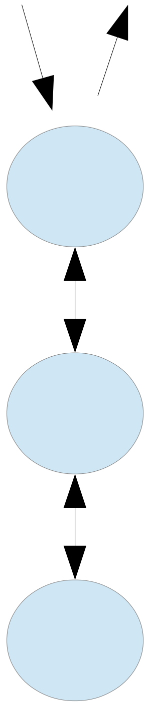
Dynamic data structures

- many programs rely on being able to incrementally add to/remove from collections of data
- simplest forms are lists, usually where we add to the front or back, usually each item has a pointer to the things just ahead of/behind them in the list
- common variants are queues (add to back, remove from front) and stacks (add to front/top, remove from front/top)
- more sophisticated structures can be created if we give the items pointers to greater numbers of other items

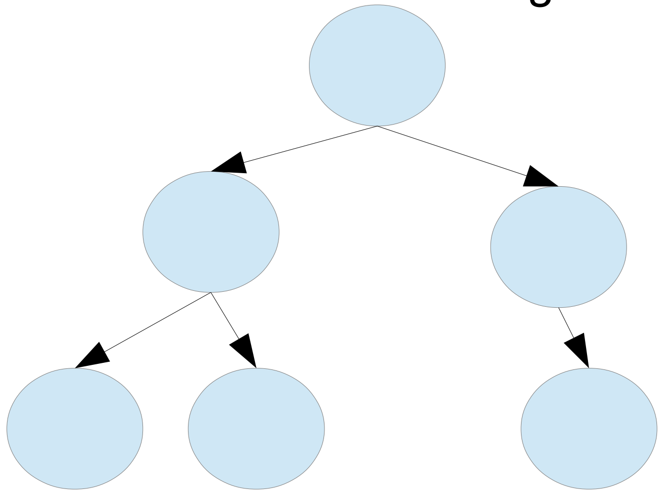
Dynamic data structure examples

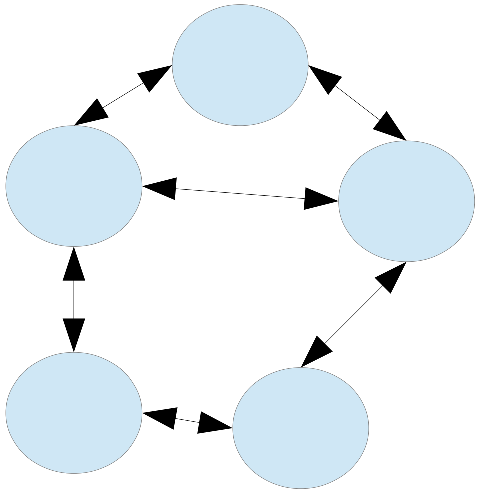

queue: insert at back, remove from front

stack: push/pop at top



tree: access through “root”




graph: anything can connect to anything