Programming techniques

Week 3 – Topic 2

Pointers and Dynamic Memory (cont)

2/2014

Agenda

- Review
- Dynamically allocating structures
- Combining the notion of classes and pointers
- Destructors

Review of Pointers

- What is a pointer?
- How would you define a pointer variable, that can point to a float?
- Would this change if you wanted the pointer to reference an array of floats?
- Show how to dynamically allocate an array of 20 floats
- Show two ways of accessing element 19

Review of Pointers

- What operator allocates memory dynamically?
- What does it really mean to allocate memory? Does it have a name?
- Why is it important to subsequently deallocate that memory?
- What operator deallocates memory?

Dynamic Structures

- Let's take these notions and apply them to dynamically allocated structures
- What if we had a video structure, how could the client allocate a video dynamically?

```
video *ptr = new video;
```

- Then, how would we access the title?

```
*ptr.title
```

? Nope! WRONG

Dynamic Structures

- To access a member of a struct, we need to realize that there is a “precedence” problem.
- Both the dereference (*) and the member access operator (.) have the same operator precedence....and they associate from right to left
- So, parens are required:

```
(*ptr).title
```

Correct (but ugly)

Dynamic Structures

- A short cut (luckily) cleans this up:

```
(*ptr).title
```

Correct (but ugly)

Can be replaced by using the indirect member access operator (->) ... it is the dash followed by the greater than sign:

```
ptr->title
```

Great!

Dynamic Structures

- Now, to allocate an array of structures dynamically:

```
video *ptr;
ptr = new video[some_size];
```

- In this case, how would we access the first video's title?

```
ptr[0].title
```

Notice that the -> operator would be incorrect in this case because ptr[0] is not a pointer variable. Instead, it is simply a video object. ptr is a pointer to the first element of an array of video objects

Dynamic Structures

- What this tells us is that the -> operator expects a pointer variable as the first operand.
  - In this case, ptr[0] is not a pointer, but rather an instance of a video structure. Just one of the elements of the array!
  - the . operator expects an object as the first operand...which is why it is used in this case!

Dynamic Structures

- Ok, what about passing pointers to functions?
- Pass by value and pass by reference apply.
  - Passing a pointer by value makes a copy of the pointer variable (i.e., a copy of the address).
  - Passing a pointer by reference places an address of the pointer variable on the program stack.

Dynamic Structures

- Passing a pointer by value:

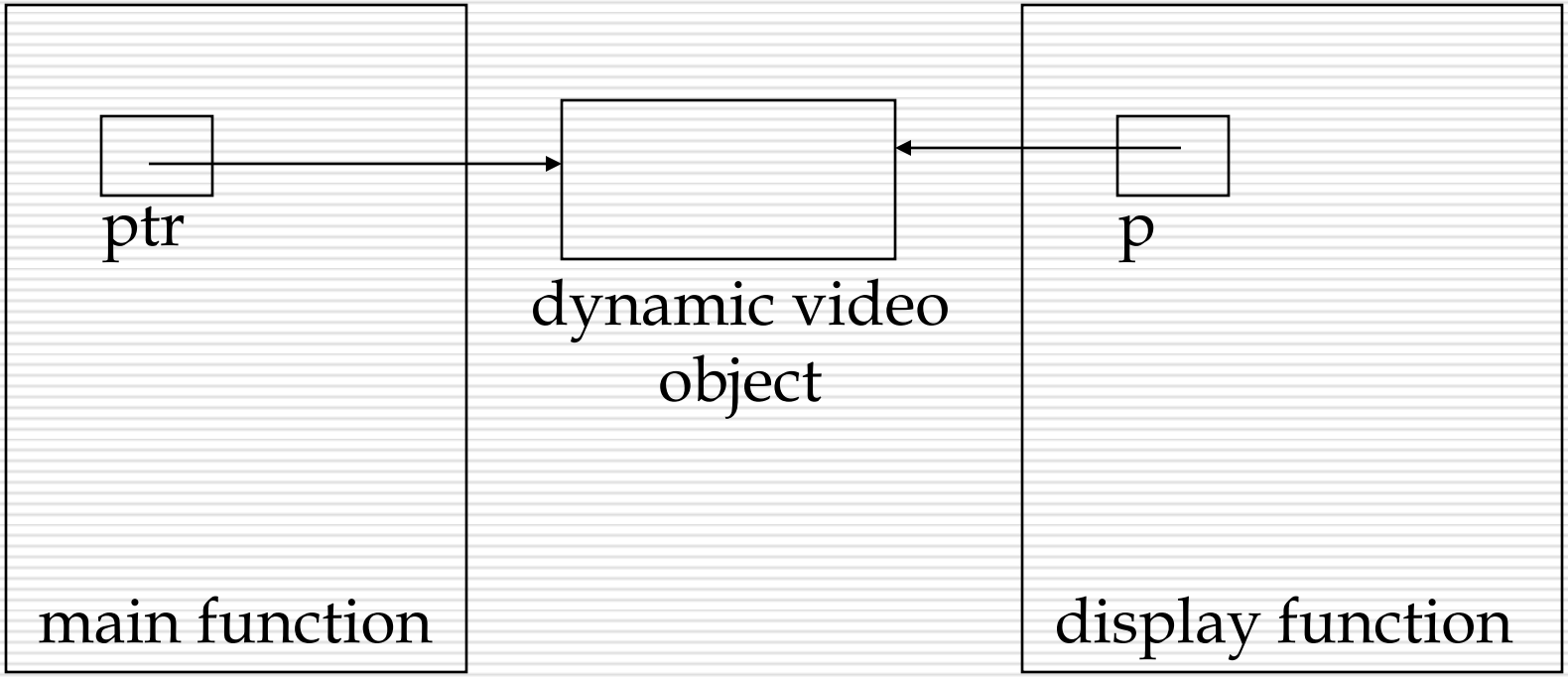
```
video *ptr = new video;
display(ptr);
```

```
void display(video * p) {
  cout << p->title <<endl;
}
```

p is a pointer to a video object, passed by value. So, p is a local variable with an initial value of the address of a video object

Dynamic Structures

- Here is the pointer diagram for the previous example:



Dynamic Structures

- Passing a pointer by reference allows us to modify the calling routine's pointer variable (not just the memory it references):

```
video *ptr; set(ptr);  cout << ptr->title;

void set(video * & p){
   p = new video;
   cin.get(p->title,100,'\n');
   cin.ignore(100,'\n');
}
```

The order of the * and & is critical!

Dynamic Structures

- But, what if we didn't want to waste memory for the title (100 characters may be way too big (Big, with Tom Hanks)
- So, let's change our video structure to include a dynamically allocated array:

```
struct video {
  char * title;
  char category[5];
  int quantity;
  };
```

Dynamic Structures

- Rewriting the set function to take advantage of this:

```
video *ptr;        set(ptr);
```

```
void set(video * & p) {
   char temp[100];
   cin.get(temp,100,'\n');
   cin.ignore(100,'\n');
   p = new video;
   p->title = new char[strlen(temp)+1];
   strcpy(p->title,temp); }
```

watch out for where the +1 is placed!

Dynamic Structures

- But, what about that list of videos discussed earlier this term?
- Let's write a class that now allocates this list of videos dynamically, at run time
- This way, we can wait until we run our program to find out how much memory should be allocated for our video array

Dynamic Structures

- What changes in this case are the data members:

```
class list {
   public:
      list();
      int add (const video &);
      int remove (char title[]);
      int display_all();
   private:
      video *my_list;
      int video_list_size;
      int num_of_videos;
};
```

Replace the array with these

Default Constructor

- Now, let's think about the implementation.
- First, what should the constructor do?
  - initialize the data members

```
list::list() {
        my_list = NULL;
        video_list_size = 0;
        num_of_videos = 0;
}
```

Another Constructor

- Remember function overloading? We can have the same named function occur (in the same scope) if the argument lists are unique.
- So, we can have another constructor take in a value as an argument of the number of videos
  - and go ahead and allocate the memory, so that subsequent functions can use the array

2nd Constructor

```
list::list(int size) {
        my_list = new video [size];
        video_list_size = size;
        num_of_videos = 0;
}
```

Notice, unlike arrays of characters, we don’t need to add one for the terminating nul!

Clients creating objs

- The client can cause this 2nd constructor to be invoked by defining objects with initial values

```
list fun_videos(20);  //size is 20
```

If a size isn't supplied, then no memory is allocated and nothing can be stored in the array....

Default Arguments

- To fix this problem, we can merge the two constructors and replace them with a single constructor:

```
list::list(int size=100) {
        my_list = new video [size];
        video_list_size = size;
        num_of_videos = 0;
}
(Remember, to change the prototype for the
   constructor in the class interface)
```

Destructor

- Then, we can deallocate the memory when the lifetime of a list object is over
- When is that?
- Luckily, when the client's object of the list class lifetime is over (at the end of the block in which it is defined) -- the **destructor** is implicitly invoked

Destructor

- So, all we have to do is write a destructor to deallocate our dynamic memory.

```
list::~list() {
        delete [] my_list;
        my_list = NULL;
        ...
}
```

(Notice the ~ in front of the function name)

(It can take NO arguments and has NO return type)

(This too must be in the class interface....)