More Structures

Steven R. Bagley

Recap

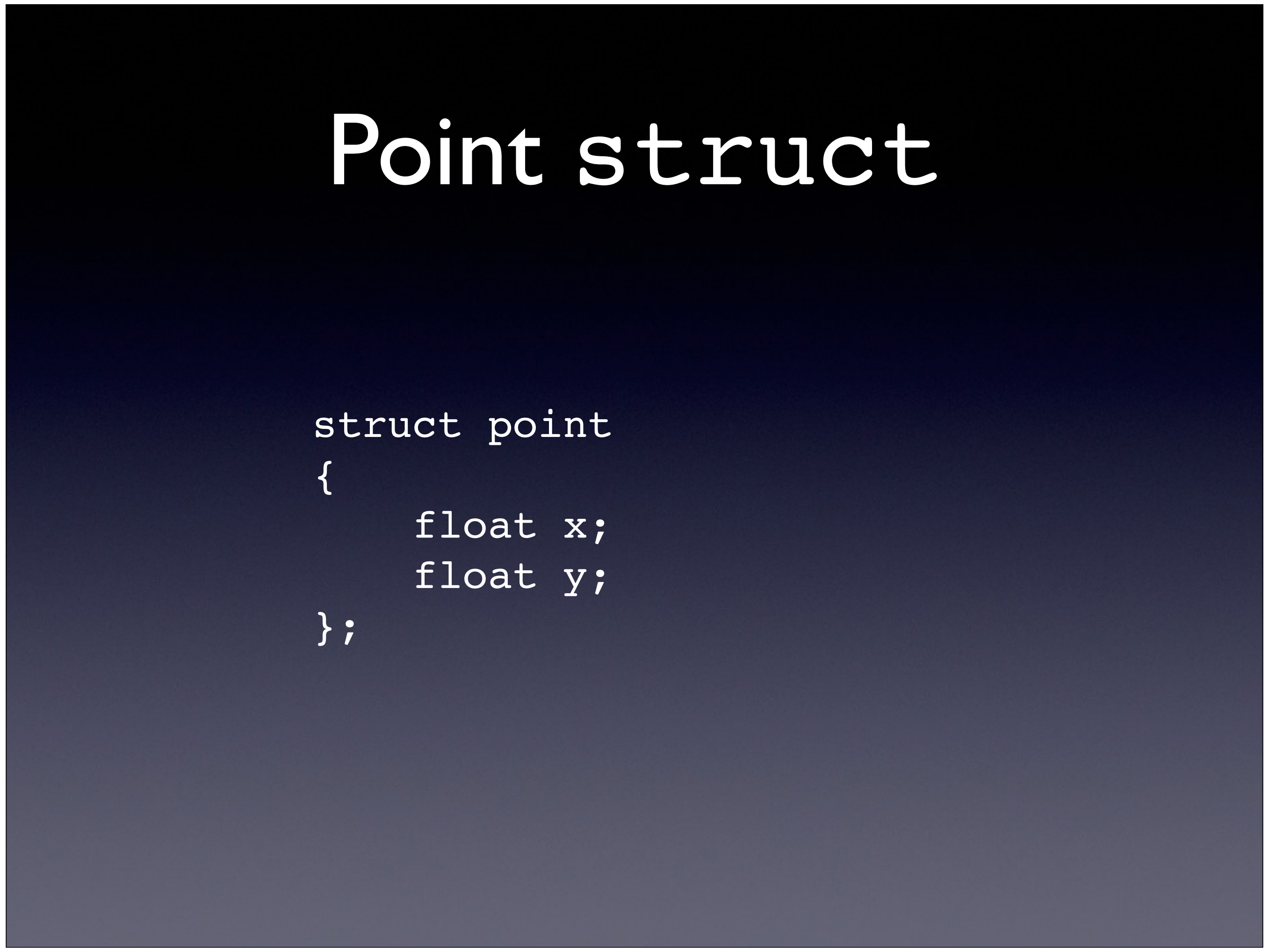
- Data is stored in variables
- Can be accessed by the variable name
- Or in an array, accessed by name and index
  `a[42] = 35;`
- Variables and arrays have a type
  `int`, `char`, `double`, etc.
- Create our own data `structures`

Large Programs

- For the next few lectures, we'll take a simplified look at how data is stored in a large program
- Use a simple example of a vector graphics package
- Shapes stored as a series of straight lines
- Use `structs`, arrays etc. to store these

A Graphical Example

- Let's have a look at how we would create a few `structs`
- Ones that would be useful for graphics
- Basic object is a Point
- Every Point has an x and y coordinate (we'll store them as `floats`)



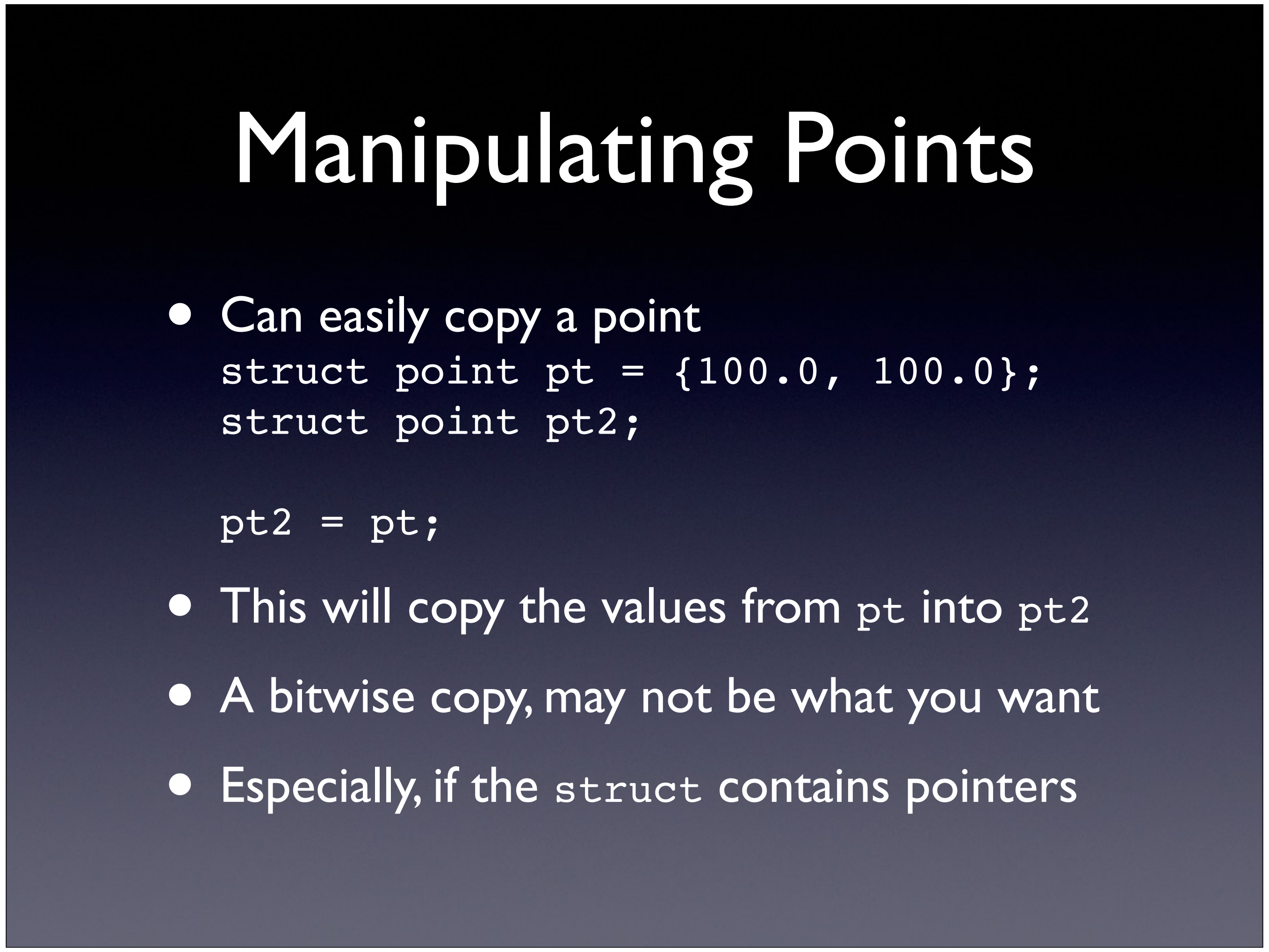
Point `struct`

```c
struct point
{
    float x;
    float y;
};
```

Go and implement in a C program

Manipulating Points

- Can easily create a point
  ```
  struct point pt = {100.0, 100.0};
  ```
- Or by `scanf()`
  ```
  scanf("%f,%f", &pt.x, &pt.y);
  ```
- Note how we can get a pointer to the individual member variables



Manipulating Points

- Can easily copy a point

```
struct point pt = {100.0, 100.0};
struct point pt2;

pt2 = pt;
```

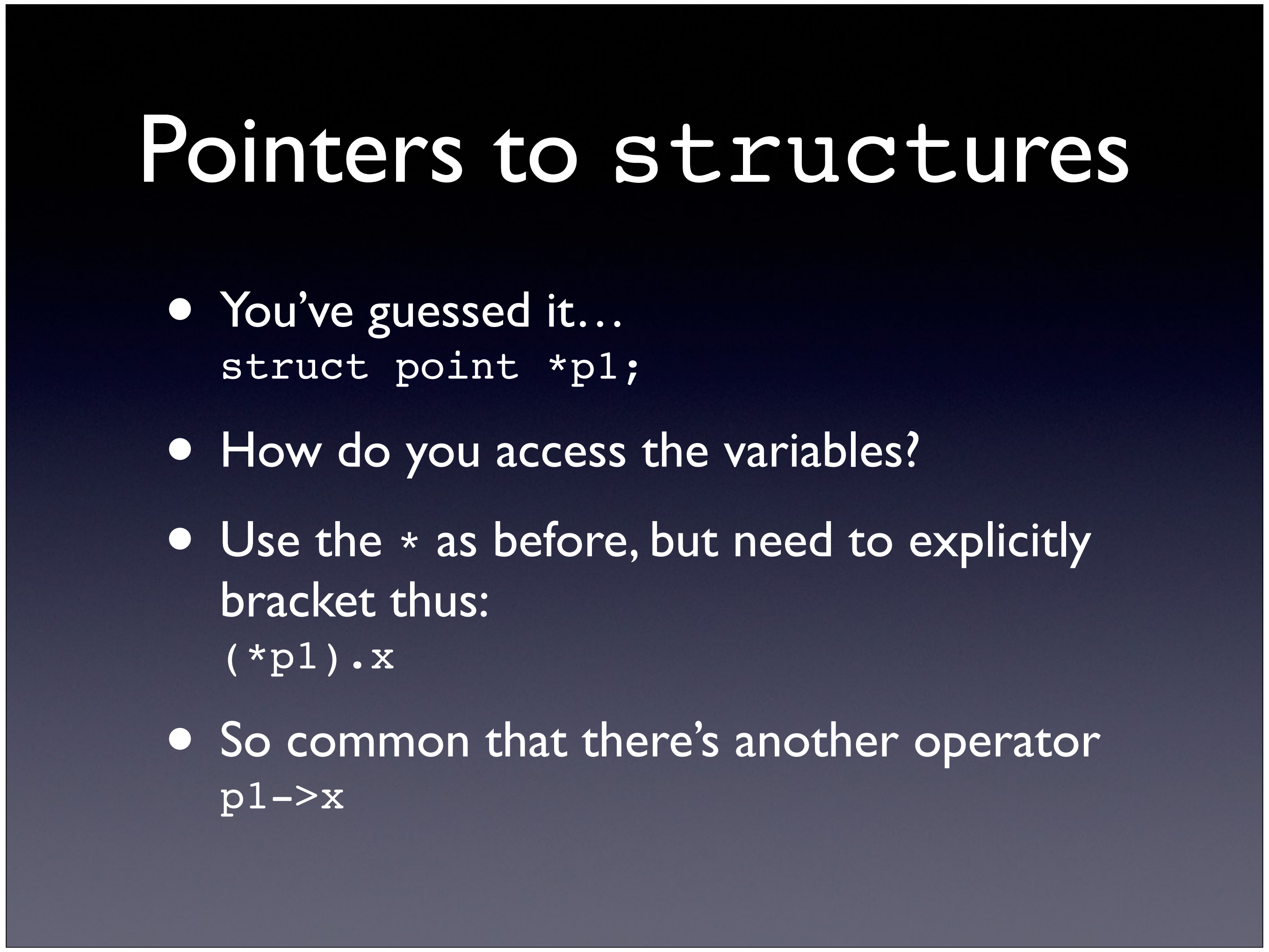
- This will copy the values from `pt` into `pt2`
- A bitwise copy, may not be what you want
- Especially, if the `struct` contains pointers

i.e copies every bit to be identical

If you don’t want a bitwise copy, then you’ll need to write your own copy routine
as a function of course :)

Manipulating Points

- Pass `struct`s as parameters to functions
  ```
  double distance(struct point p1, struct point p2)
  ```
- Note this makes a *copy* of the `struct`'s data
- This is slow…
- We could also pass a pointer to the `struct`



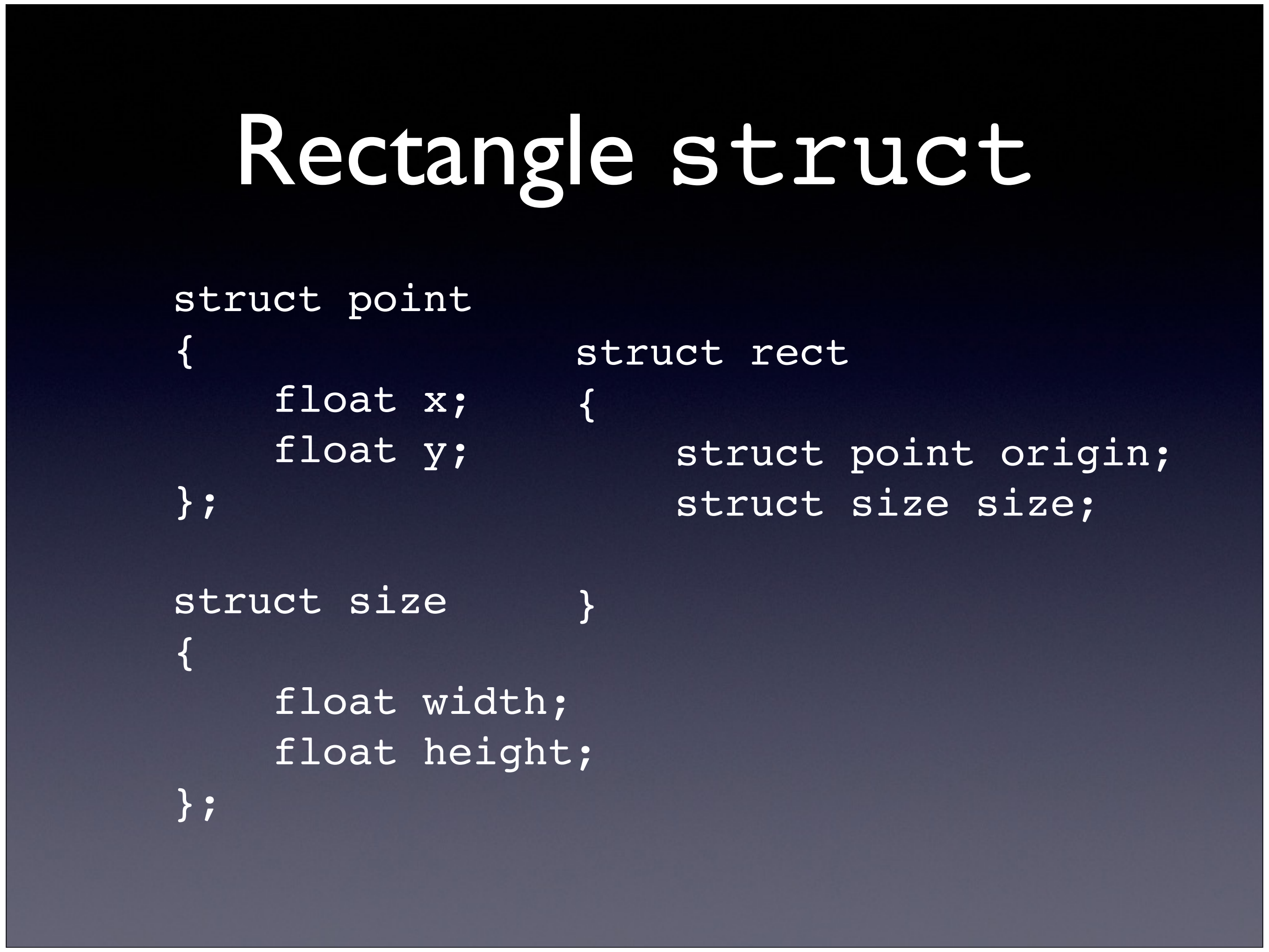
Pointers to `structures`

- You’ve guessed it…
  `struct point *p1;`
- How do you access the variables?
- Use the `*` as before, but need to explicitly bracket thus:
  `(*p1).x`
- So common that there’s another operator
  `p1->x`

p1->x and (*p1).x are equivalent, the former is just easier to remember

`structs` in `structs`

- Can also put a `struct` inside another `struct` (but not inside itself)
- So could define a rectangle as being constructed from two `struct points`
- Or a `struct point` and a `struct size`
- `structs` are nested inside each other
- Or you can point at another `struct`



Rectangle `struct`

```
struct point
{                   struct rect
    float x;        {
    float y;            struct point origin;
};                      struct size size;

struct size         }
{
    float width;
    float height;
};
```

Go and implement in a C program

Accessing nested `structs`

- Access nested `structs` just as we access normal `structs`
- Give the name of the instance followed by . followed by the name of the variable
- But the name of the instance is the name of the variable in the outer `struct` e.g.

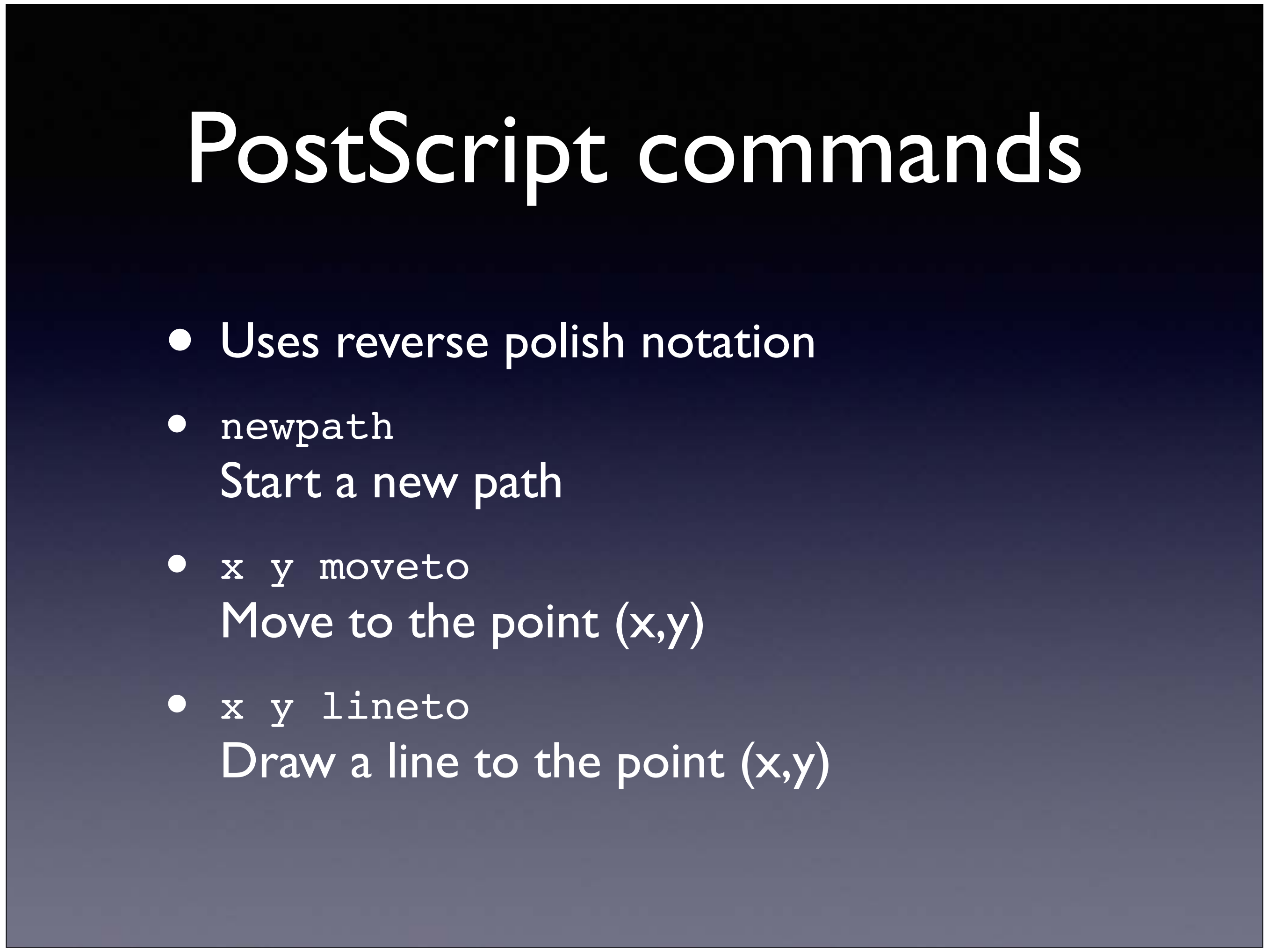
```
struct rect r1;
r1.origin.x = 42.0;
```

Drawing Pictures

- Lets put this to use to draw some pictures
- Rather than get messy trying to draw on the screen, we'll output PostScript
- PostScript is the page description language that drives laser printers
- Simple and ASCII based
- So great to generate from C

PostScript

- PostScript works in terms of a path
- Define the path
- Then fill it or stroke it
- A path is a series of points that can be connected with lines



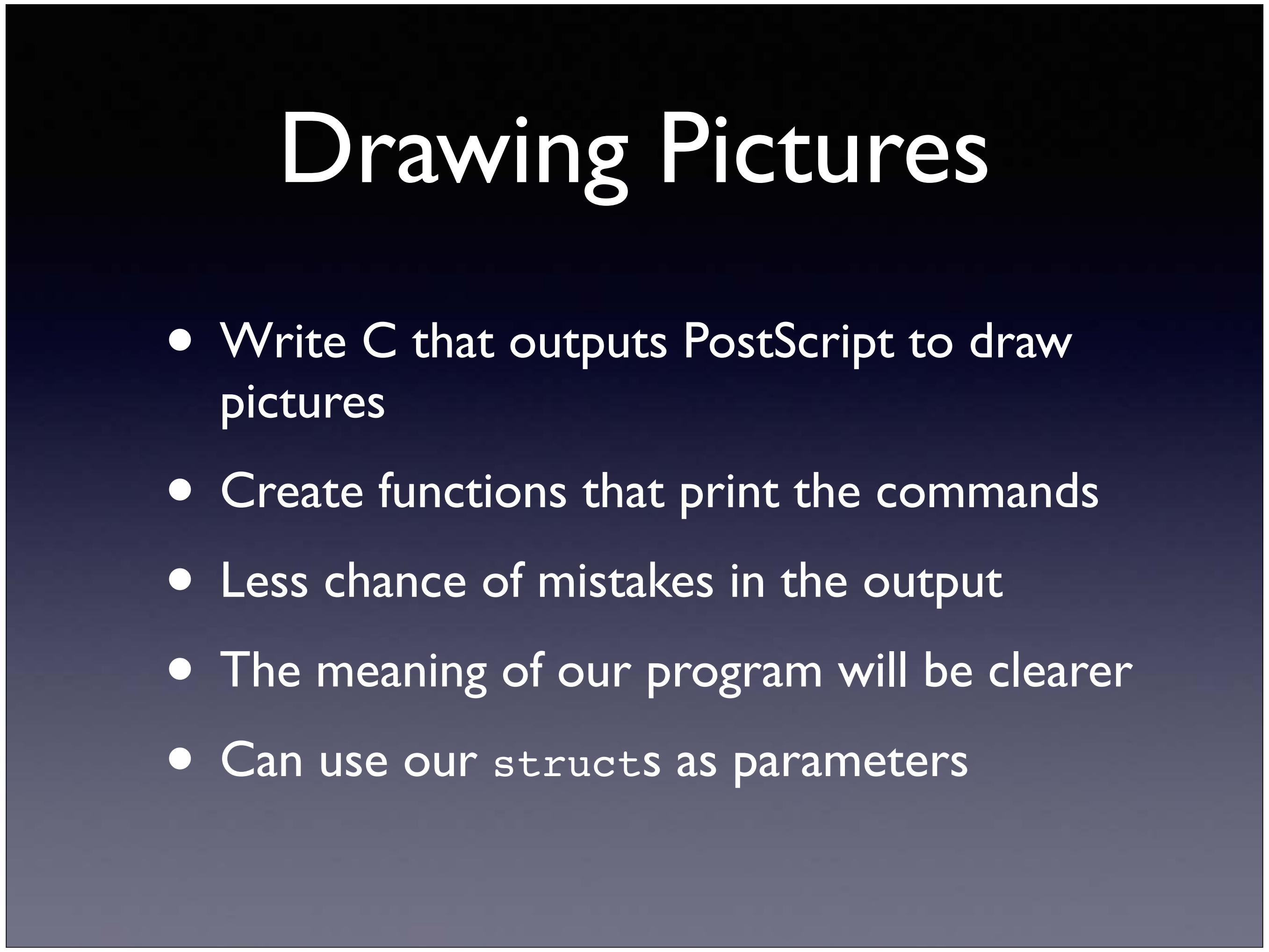
PostScript commands

- Uses reverse polish notation
- `newpath`
  Start a new path
- `x y moveto`
  Move to the point (x,y)
- `x y lineto`
  Draw a line to the point (x,y)

This means that the parameters come before the operators

PostScript commands

- `closepath`
  Connect the end of the path to the start
- `fill`
  Fill in the path
- `stroke`
  stroke the outline of the path
- `showpage`
  draw things on screen



Drawing Pictures

- Write C that outputs PostScript to draw pictures
- Create functions that print the commands
- Less chance of mistakes in the output
- The meaning of our program will be clearer
- Can use our `struct`s as parameters

Clearer, because we will see function calls that mean something (e.g. MoveToPoint, AddLineToPoint) rather than a series of printf programs

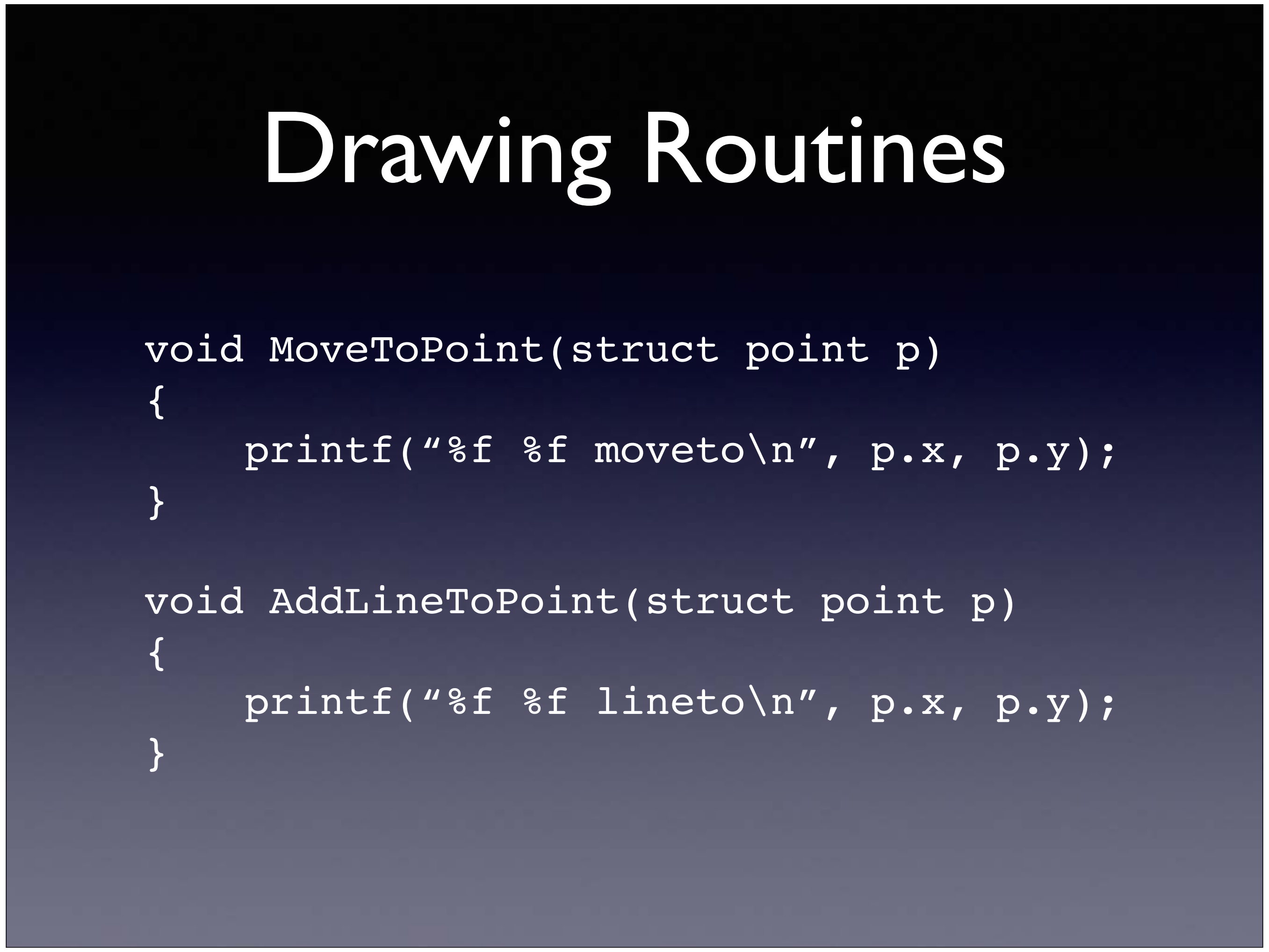
Code Generation

- Some functions are very simple
- Just print out the PostScript command
- No parameters

```
void BeginPath()
{
    printf(“newpath\n”);
}
```

Parameter based

- Others will take a `struct point` as a parameter
- Print out the relevant command, reading the values from the struct
- Can also do more work
  e.g. draw rectangle by outputting four `lineto`s

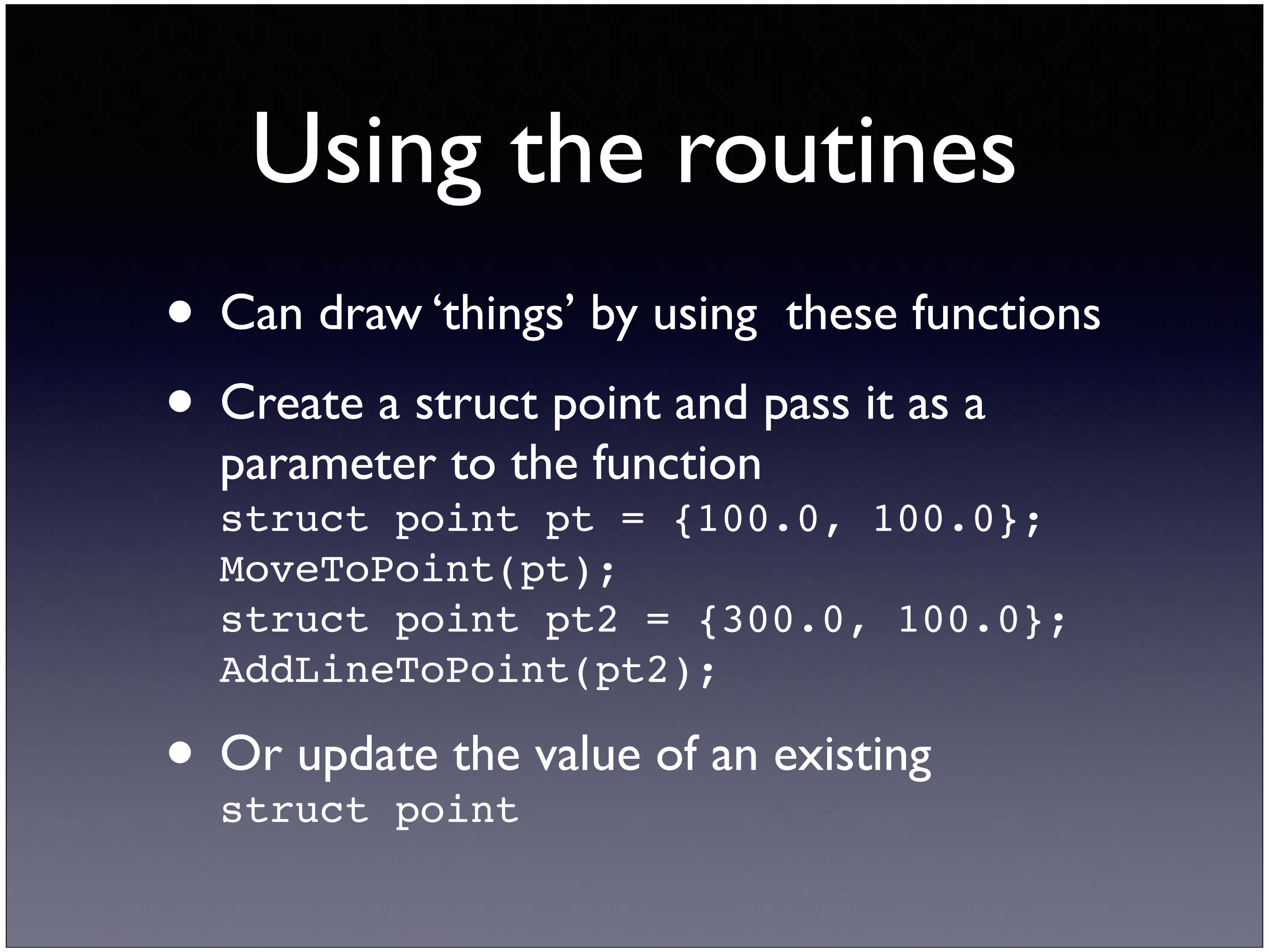


Drawing Routines

```
void MoveToPoint(struct point p)
{
    printf(“%f %f moveto\n”, p.x, p.y);
}

void AddLineToPoint(struct point p)
{
    printf(“%f %f lineto\n”, p.x, p.y);
}
```

Go and implement in a C program
Also implement Rect



Using the routines

- Can draw ‘things’ by using these functions
- Create a struct point and pass it as a parameter to the function

```
struct point pt = {100.0, 100.0};
MoveToPoint(pt);
struct point pt2 = {300.0, 100.0};
AddLineToPoint(pt2);
```

- Or update the value of an existing `struct point`

Go show an example...

Library

- Generated a library of routines
- Can use them anywhere to generate PostScript output
- This is all the standard C library is
- A set of routines defined (somewhere) to do common tasks

Real-life Libraries

- Lots and lots of libraries around to help you
- Often several come with the development platform
- These examples are based on the one used by MacOS X and iOS
- Windows has similar

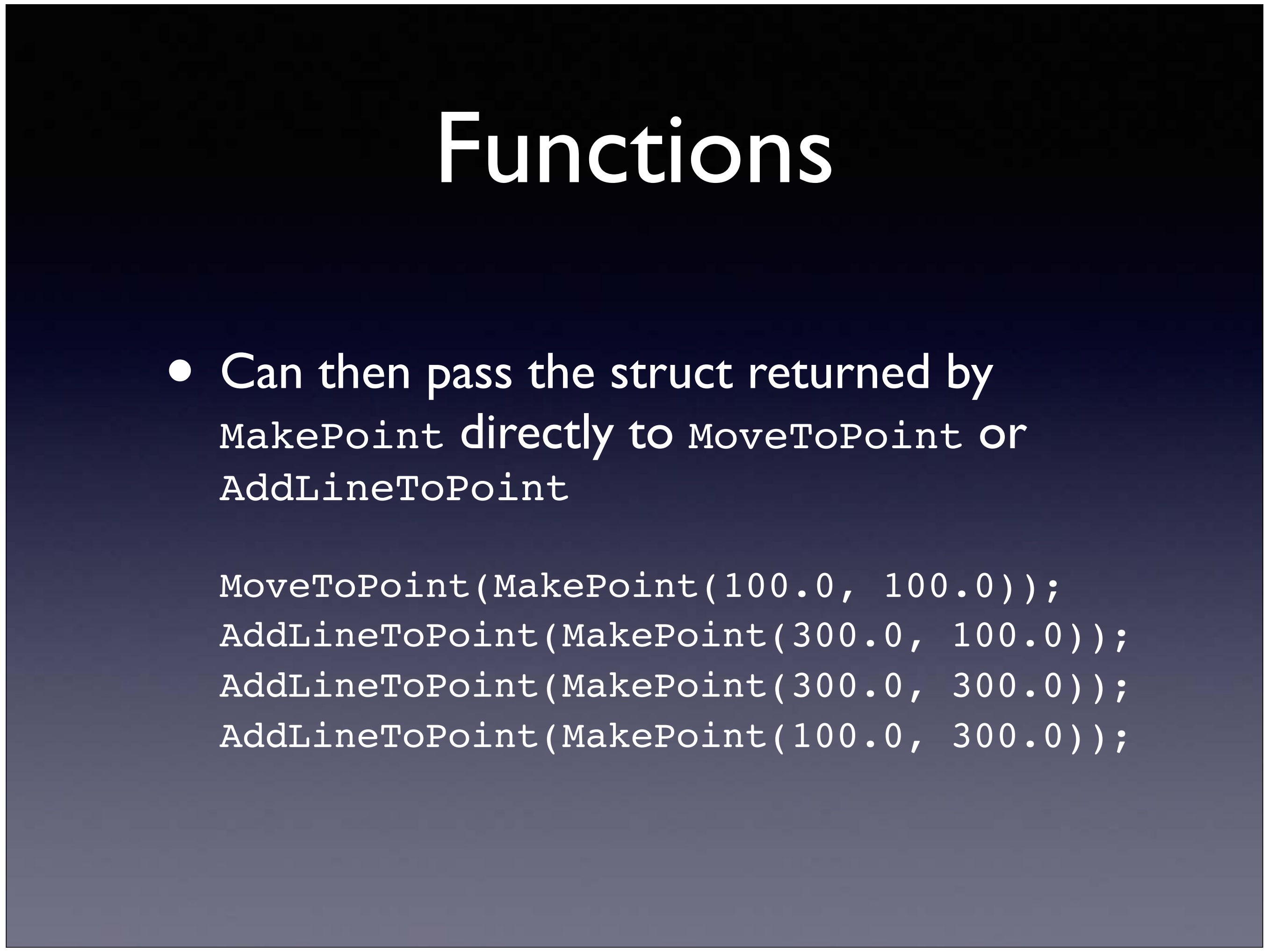
Large Program

- Let's see how we would use these libraries in our Vector Graphics program
- Need to store the data for the picture as a series of `struct points`
- Obvious starting place would be to use an array

Returning `struct`

- Can be a pain to define all these structs
- Define a function that returns `point`

```
struct point MakePoint(float x,
float y)
{
    struct point pt = {x, y};
    return pt;
}
```



Functions

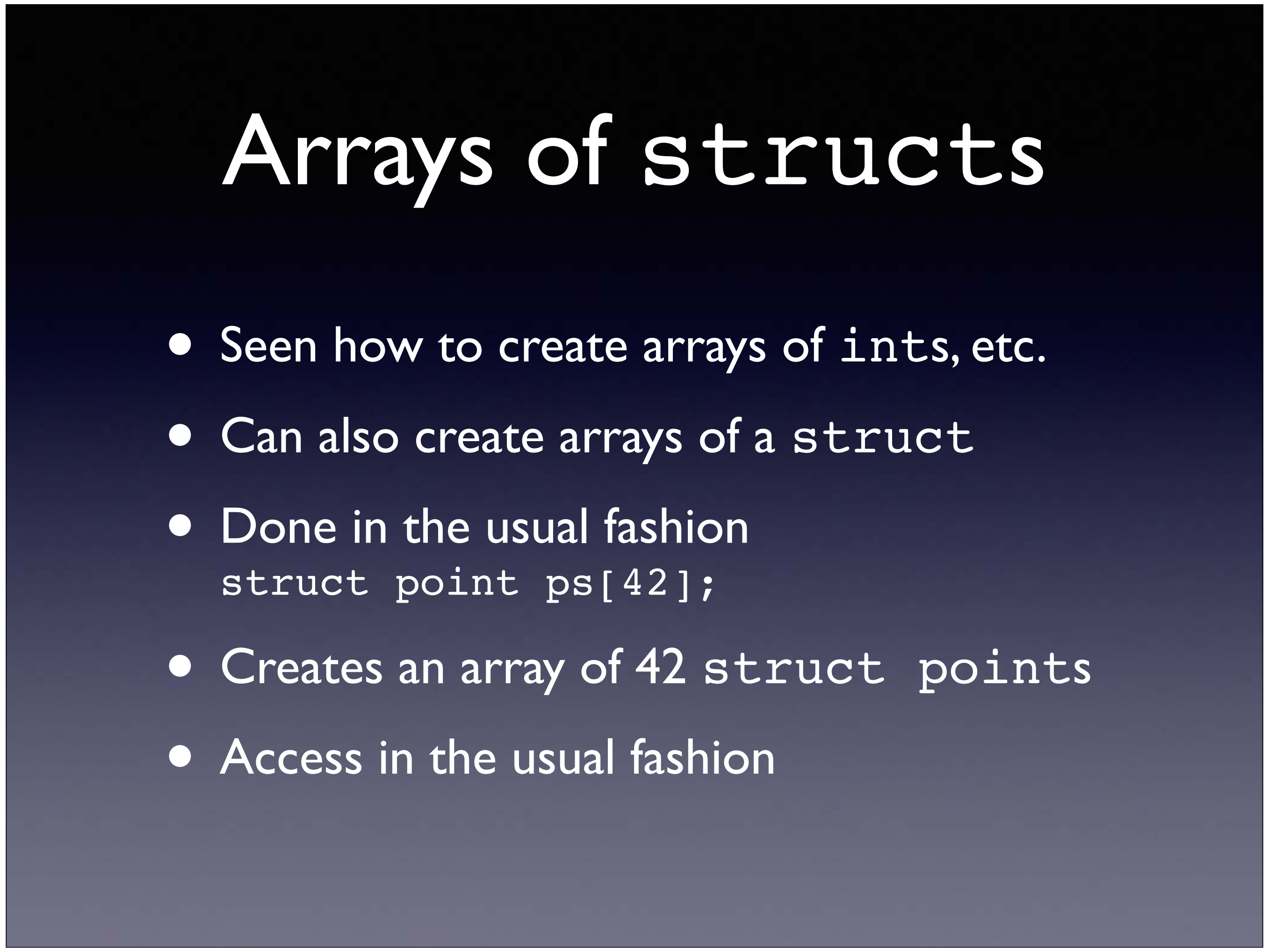
- Can then pass the struct returned by `MakePoint` directly to `MoveToPoint` or `AddLineToPoint`

```
MoveToPoint(MakePoint(100.0, 100.0));
AddLineToPoint(MakePoint(300.0, 100.0));
AddLineToPoint(MakePoint(300.0, 300.0));
AddLineToPoint(MakePoint(100.0, 300.0));
```

Go demo

Arrays of `structs`

- Just as we can have an array of `ints` etc
- We can also have an array of `structs`
- Defined in the same way
  `struct point fred[10];`
- Would declare `fred` as an array of 10 `struct points`



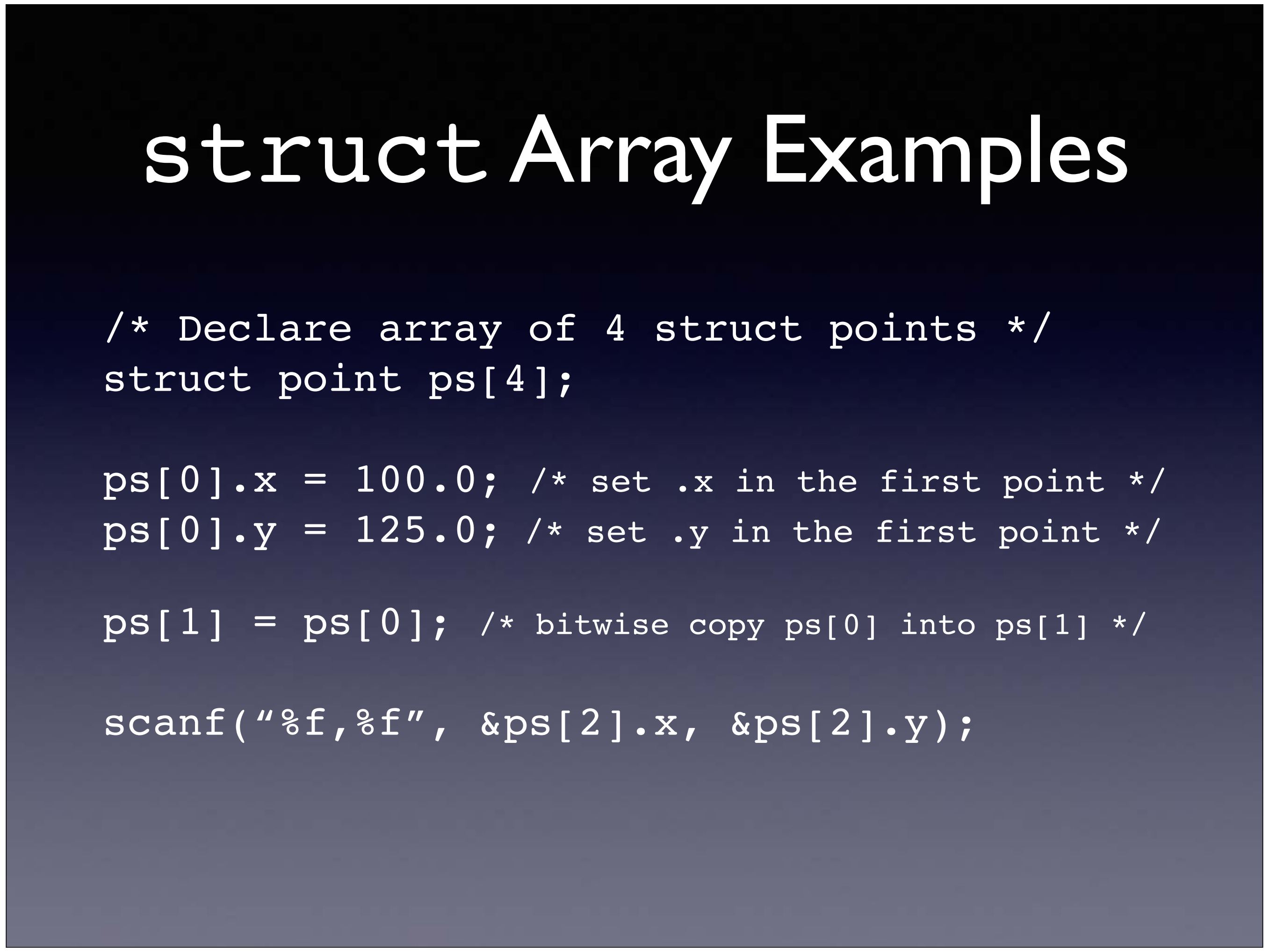
Arrays of `structs`

- Seen how to create arrays of `ints`, etc.
- Can also create arrays of a `struct`
- Done in the usual fashion
  `struct point ps[42];`
- Creates an array of 42 `struct points`
- Access in the usual fashion

Go give a demo…

Accessing a `struct` in an array

- Same syntax, specify the index inside `[]`
- This gives a specific `struct` instance
- To access a variable in that instance, use the `.` operator
- Left-hand side of `.` is instance of a `struct`
- Right-hand side of `.` is a variable in the `struct`



`struct` Array Examples

```
/* Declare array of 4 struct points */
struct point ps[4];

ps[0].x = 100.0; /* set .x in the first point */
ps[0].y = 125.0; /* set .y in the first point */

ps[1] = ps[0]; /* bitwise copy ps[0] into ps[1] */

scanf("%f,%f", &ps[2].x, &ps[2].y);
```

Note the binding of &, . and [] mean we don't need to put the brackets in...

Initializing a `struct` array

- Can initalize a struct array when it is created
- Again provide the values in braces `{ … }`

```
struct point foo[] = { {1,2}, {3,4},
                       {5,6} };
```

- Note that we enclose each instance in its one pair of braces
- Or set the values manually…

Efficiency

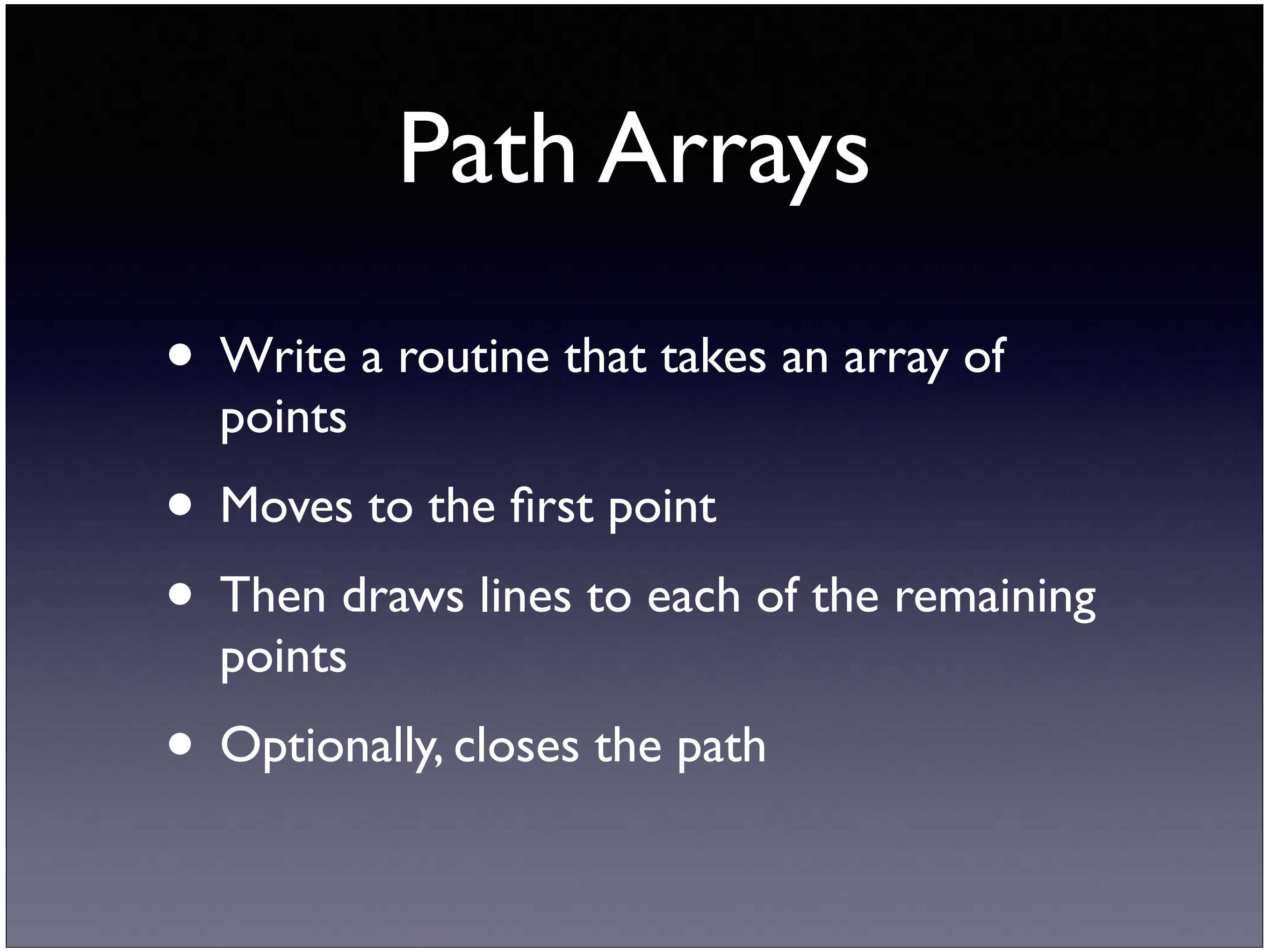
- Remember that if you auto-initialize a local array it has to copy all the values in place
- This can take a long time, especially with `structs` as they contain lots of data
- Better to use `static` or make the array global, if appropriate
- Not a problem if you don't initalize…

`struct` Pointer

- Can also treat `ps` as a pointer to a `struct`
- So `ps->x` would access the `x` value in the first `struct`
- Can even use pointer arithmetic
  `(ps+2)->x = (ps+2)->y;`
- Would access the second `struct`, same as `ps[2].x`

Remember the name of an array can be considered a (read-only)
pointer to the base of the array

Adding a value to a pointer multiplies the value by the size of the thing pointed to

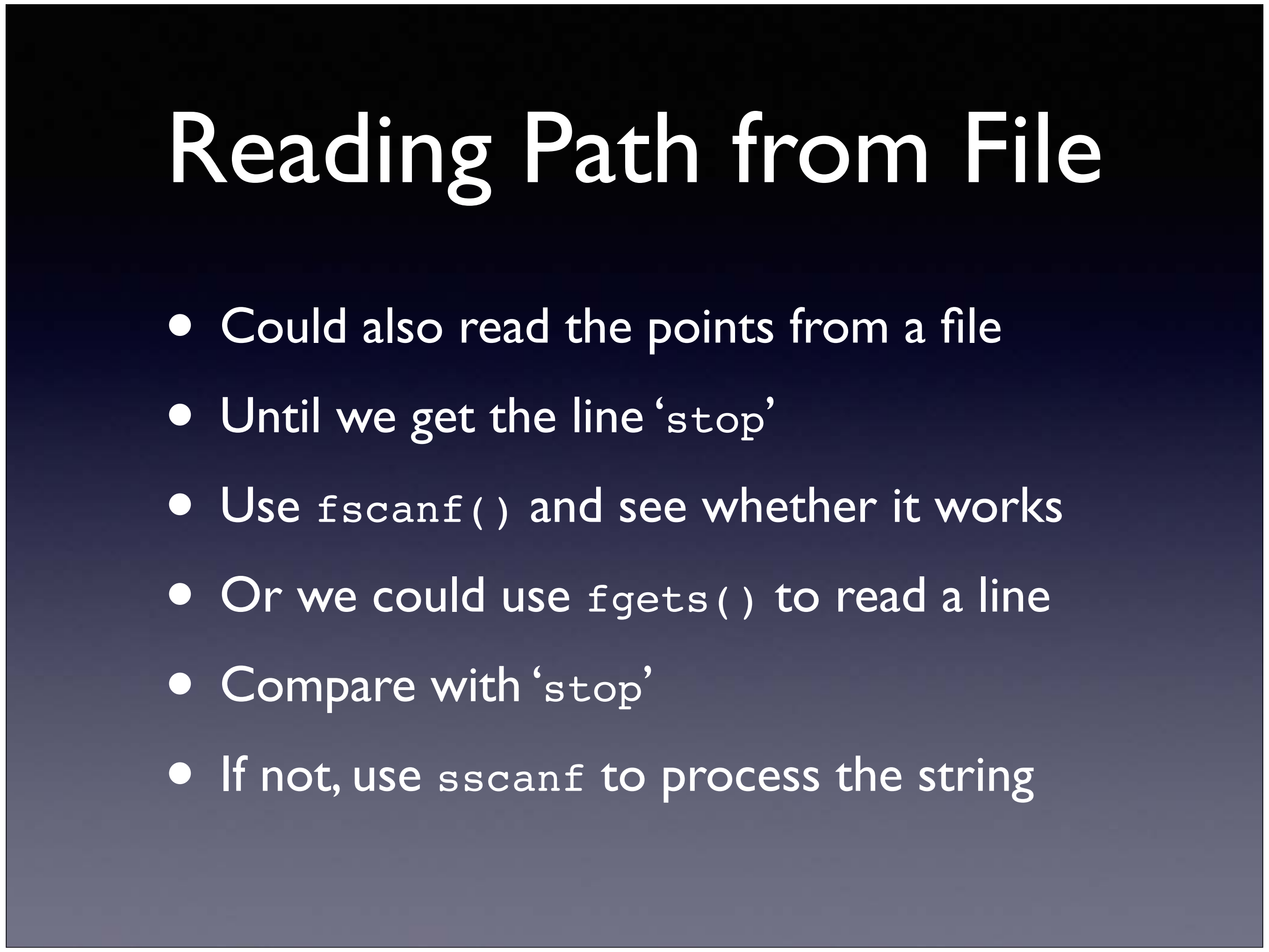


Path Arrays

- Write a routine that takes an array of points
- Moves to the first point
- Then draws lines to each of the remaining points
- Optionally, closes the path

Go write the routine as a function

```
void DrawPath(int numPoints, struct point pts[], int close);
```



Reading Path from File

- Could also read the points from a file
- Until we get the line ‘`stop`’
- Use `fscanf()` and see whether it works
- Or we could use `fgets()` to read a line
- Compare with ‘`stop`’
- If not, use `sscanf` to process the string

And loop...
Go implement
Show some examples

Out of Memory

- Problem with this routine
- What if the number of points in the file is greater than the size of the array?
- Program will CRASH!
- Could stop when we fill the array
- But that would leave half the picture undrawn