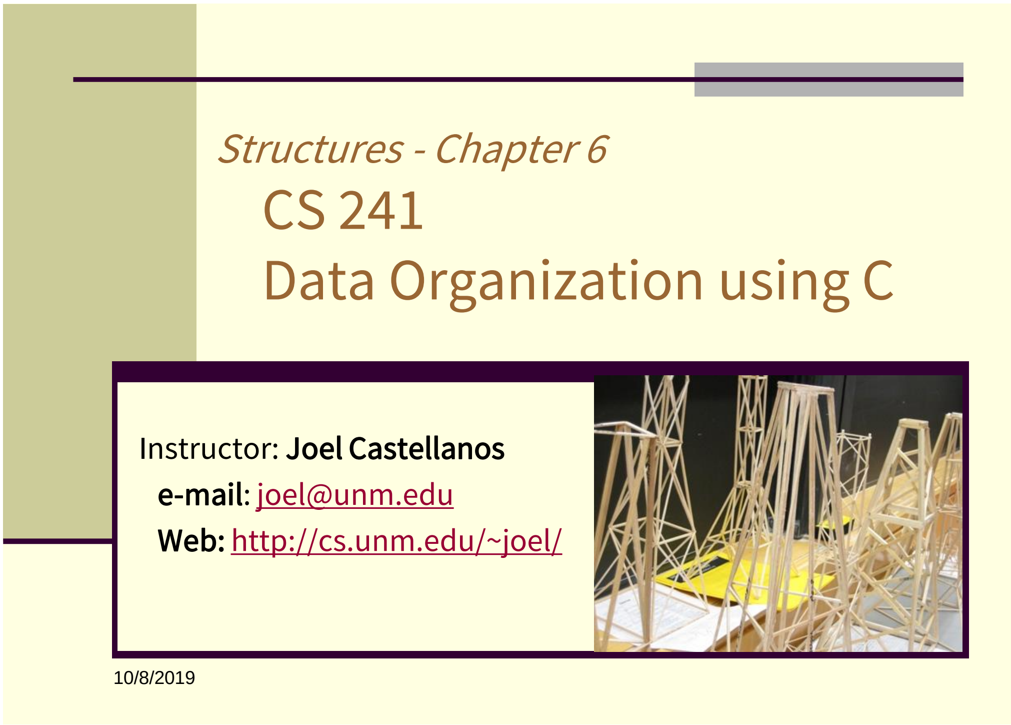


Structures - Chapter 6

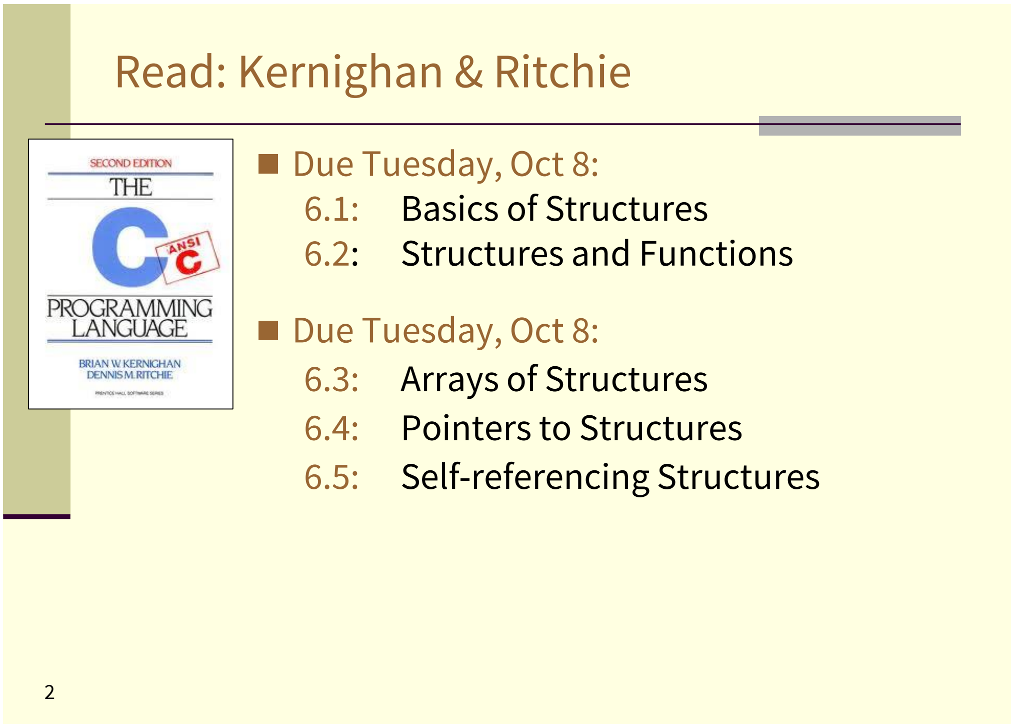
CS 241
Data Organization using C

Instructor: **Joel Castellanos**

e-mail: joel@unm.edu

Web: http://cs.unm.edu/~joel/

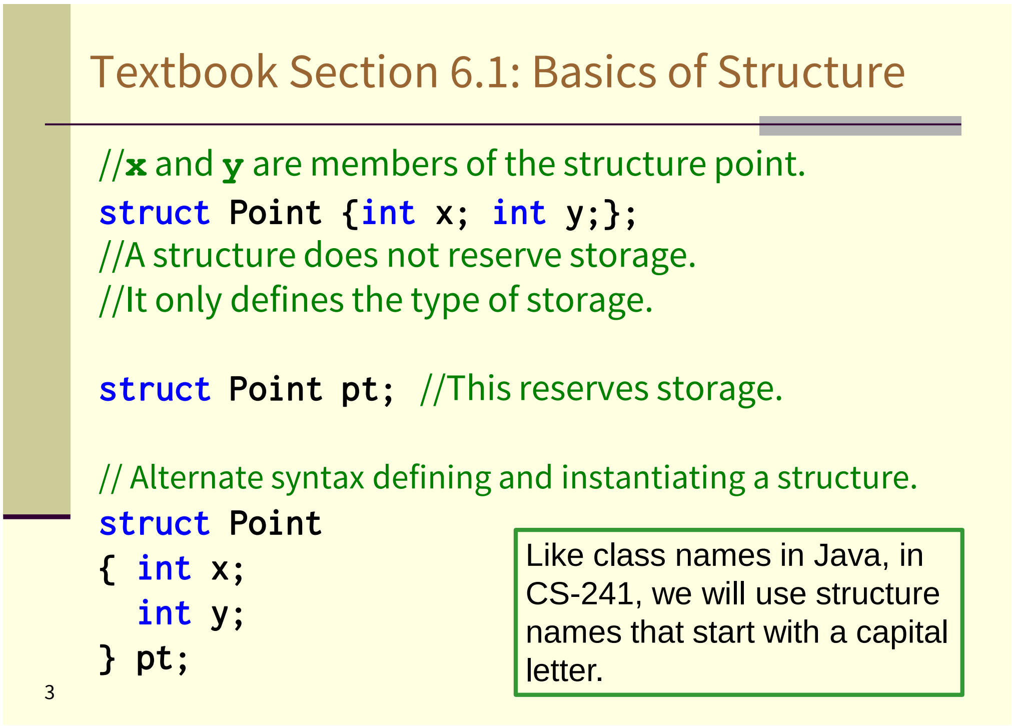
10/8/2019



Read: Kernighan & Ritchie

SECOND EDITION
THE
C
ANSI C
PROGRAMMING LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

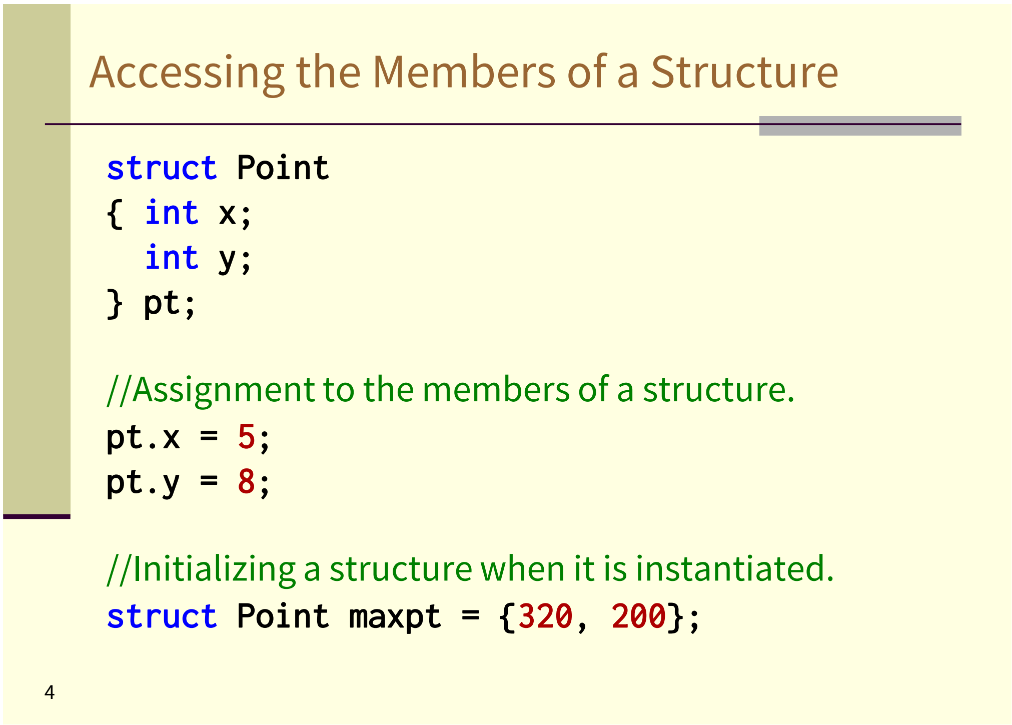
- Due Tuesday, Oct 8:
  - 6.1: Basics of Structures
  - 6.2: Structures and Functions
- Due Tuesday, Oct 8:
  - 6.3: Arrays of Structures
  - 6.4: Pointers to Structures
  - 6.5: Self-referencing Structures



Textbook Section 6.1: Basics of Structure

```
//x and y are members of the structure point.
struct Point {int x; int y;};
//A structure does not reserve storage.
//It only defines the type of storage.

struct Point pt;  //This reserves storage.

// Alternate syntax defining and instantiating a structure.
struct Point
{ int x;
  int y;
} pt;
```

Like class names in Java, in CS-241, we will use structure names that start with a capital letter.



Accessing the Members of a Structure

```
struct Point
{ int x;
  int y;
} pt;

//Assignment to the members of a structure.
pt.x = 5;
pt.y = 8;

//Initializing a structure when it is instantiated.
struct Point maxpt = {320, 200};
```

CS-241 Coding Standard

```c
struct Point
{ int x;
  int y;
};
struct Point pt;
```

```c
struct Point
{ int x;
  int y;
} pt;
```

```c
struct Point {int x; int y;} pt;
```

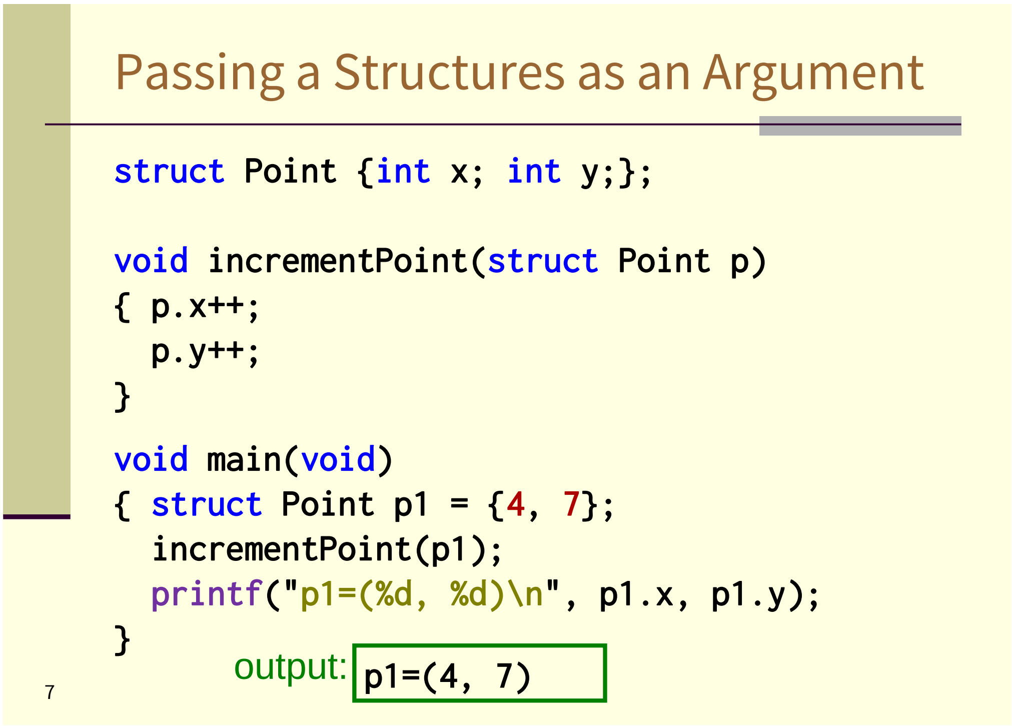
🚫

```c
          struct Point {
                int x; int y;
          } pt;
```

Section 6.2: Structures and Functions

```c
struct Point {int x; int y;};
struct Point makepoint(int x, int y)
{ //reuse of the variable names x and y is good.
  struct Point temp;
  temp.x = x;
  temp.y = y;
  return temp;
}
void main(void)
{ struct Point p1 = makepoint(5,7);
  printf("p1=(%d, %d)\n", p1.x, p1.y);
}
```

Unlike a Java class, **temp** is returned by value.

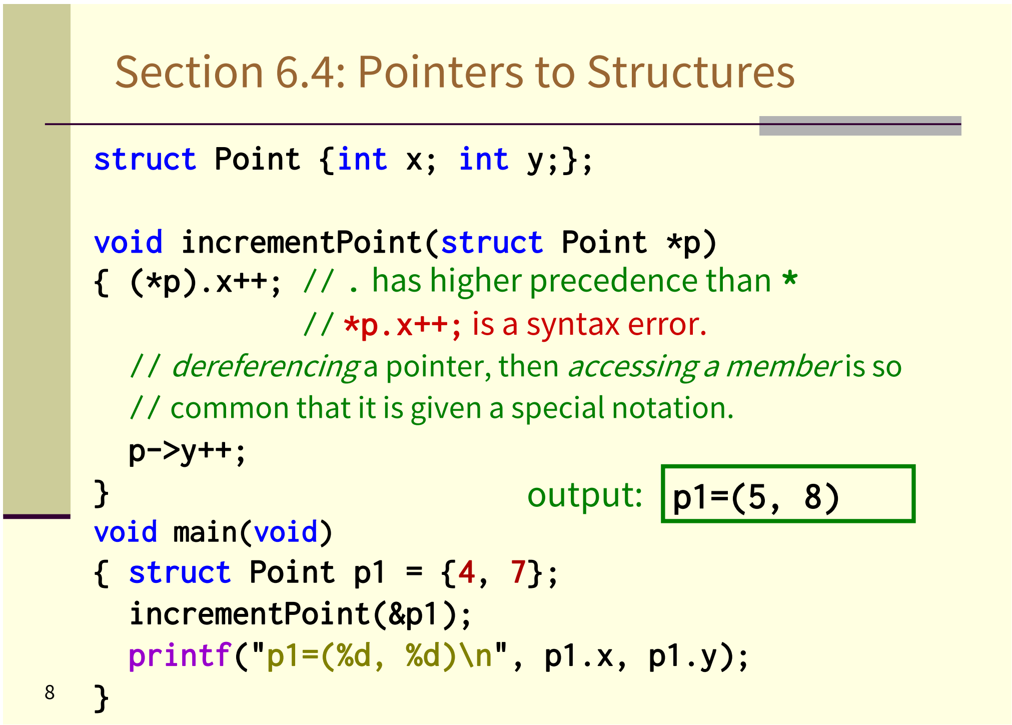


Passing a Structures as an Argument

```
struct Point {int x; int y;};

void incrementPoint(struct Point p)
{ p.x++;
  p.y++;
}
void main(void)
{ struct Point p1 = {4, 7};
  incrementPoint(p1);
  printf("p1=(%d, %d)\n", p1.x, p1.y);
}
```

output:

```
p1=(4, 7)
```



Section 6.4: Pointers to Structures

```
struct Point {int x; int y;};

void incrementPoint(struct Point *p)
{ (*p).x++; // . has higher precedence than *
            // *p.x++; is a syntax error.
  // dereferencing a pointer, then accessing a member is so
  // common that it is given a special notation.
  p->y++;
}
void main(void)
{ struct Point p1 = {4, 7};
  incrementPoint(&p1);
  printf("p1=(%d, %d)\n", p1.x, p1.y);
}
```

output:

```
p1=(5, 8)
```

Quiz: Structures and Functions

```
struct Point {int x; int y;};
struct Point incPoint(struct Point p)
{ p.x++;   p.y++;
  return p;
}
void main(void)
{ struct Point p1 = {12, 3};
  struct Point p2 = incPoint(p1);
  printf("p1=(%d, %d) p2=(%d, %d)\n",
     p1.x, p1.y, p2.x, p2.y);
}
```

a) p1=(12, 3) p2=(13, 4)

b) p1=(12, 3) p2=(12, 3)

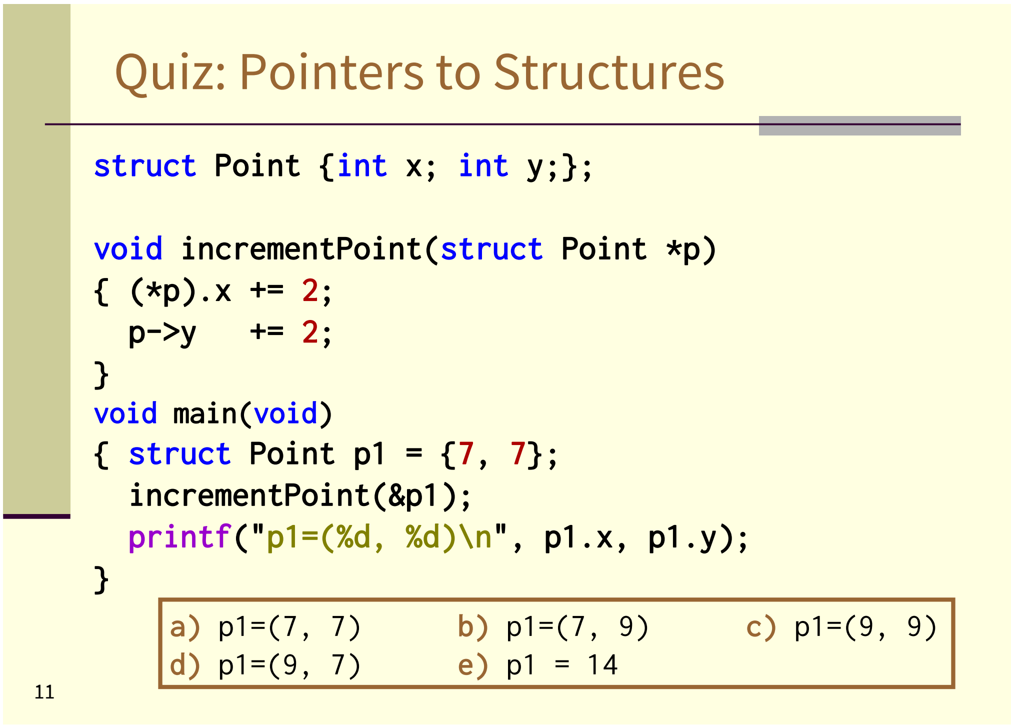
c) p1=(13, 4) p2=(13, 4)

d) p1=(13, 4) p2=(12, 3)

e) The value returned into **p2** was stored on the stack in **incPoint**. Therefore, the value in **p2** is unpredictable!

Warning: Function Returns Address of Local Variable

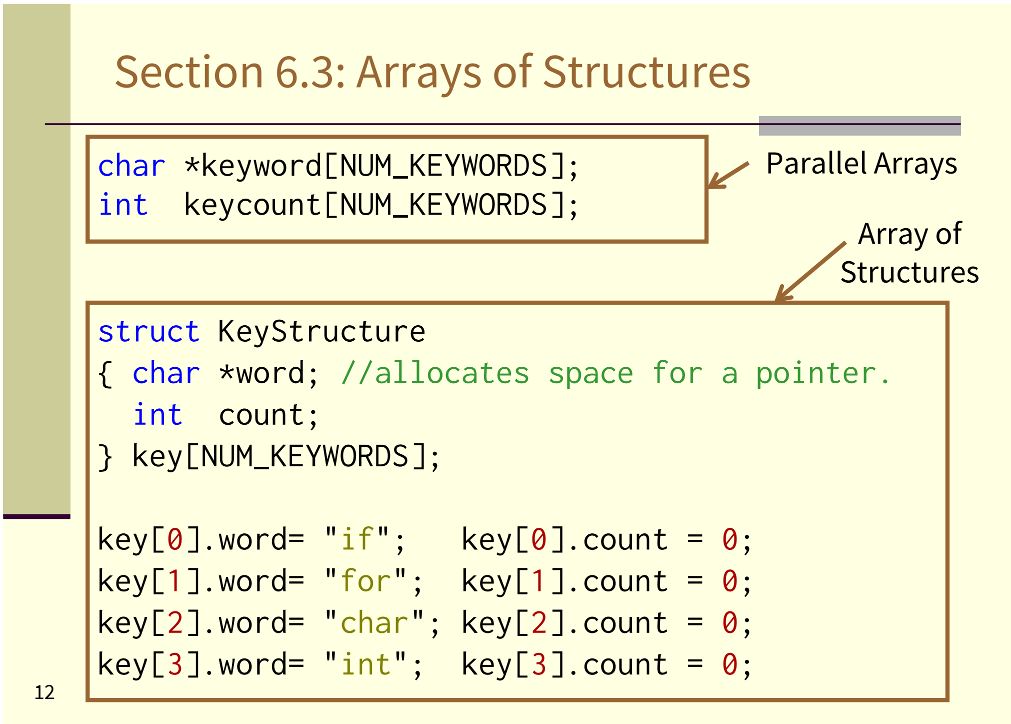
```
#include <stdio.h>
struct Point {int x; int y;};

struct Point* badPointer(int x, int y)
{
  struct Point temp;
  temp.x = x;
  temp.y = y;
  return &temp;
}

void main(void)
{ struct Point* p1 = badPointer(5,7);
  printf("p1->(%d, %d)\n", (*p1).x, (*p1).y);
}
```



Quiz: Pointers to Structures

```
struct Point {int x; int y;};

void incrementPoint(struct Point *p)
{ (*p).x += 2;
  p->y   += 2;
}
void main(void)
{ struct Point p1 = {7, 7};
  incrementPoint(&p1);
  printf("p1=(%d, %d)\n", p1.x, p1.y);
}
```

```
a) p1=(7, 7)      b) p1=(7, 9)      c) p1=(9, 9)
d) p1=(9, 7)      e) p1 = 14
```



Section 6.3: Arrays of Structures

```
char *keyword[NUM_KEYWORDS];
int   keycount[NUM_KEYWORDS];
```

Parallel Arrays

Array of Structures

```
struct KeyStructure
{ char *word; //allocates space for a pointer.
  int  count;
} key[NUM_KEYWORDS];

key[0].word= "if";   key[0].count = 0;
key[1].word= "for";  key[1].count = 0;
key[2].word= "char"; key[2].count = 0;
key[3].word= "int";  key[3].count = 0;
```

gcc Compile Error

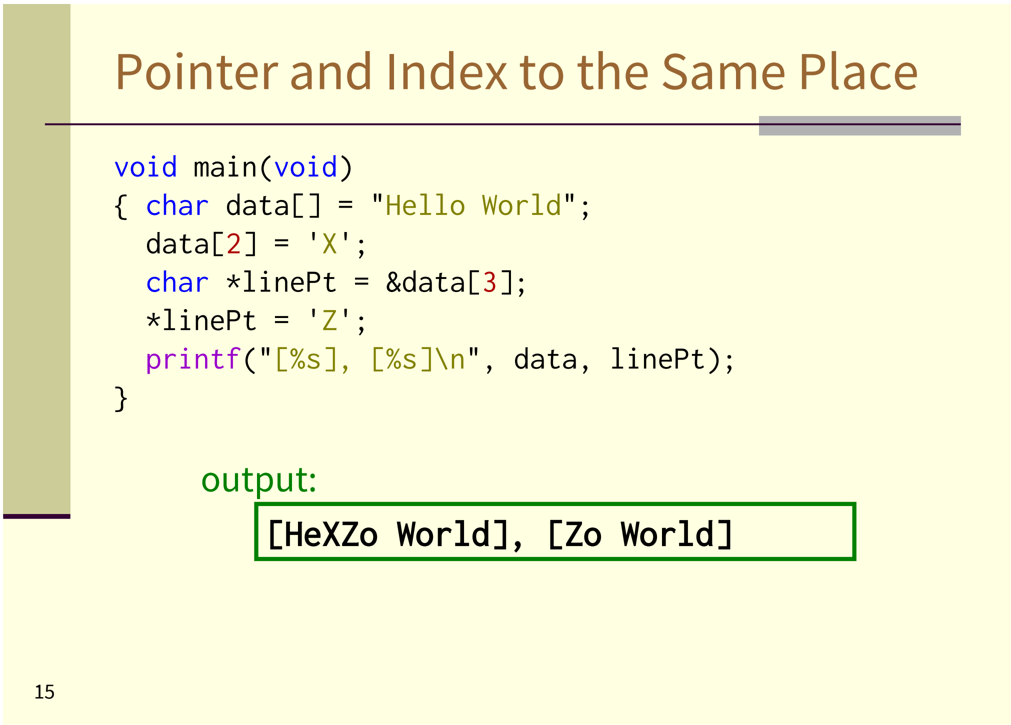
```
1. #include <stdio.h>
2. #define NUM_KEYWORDS 32;
3. int main()
4. { struct KeyStructure
5.   { char *word;
6.     int  count;
7.   } key[NUM_KEYWORDS];
8.
```

```
foo.c: In function ‘main’:
foo.c:7: error: expected ‘]’ before ‘;’ token
```

Quiz: Pointers to Structures

```
#include <stdio.h>
#include <math.h>
struct Point {double x; double y;};
void foo(struct Point *p)
{ double d = sqrt((p->x)*(p->x) + (p->y)*(p->y));
  p->x   /= d;
  p->y   /= d;
}
void main(void)
{ struct Point p1 = {2, 3};
  foo(&p1);
  printf("p1=(%5.2f, %5.2f)\n", p1.x, p1.y);
}
```

```
a) p1=(1.50,  1.75)        b) p1=( 0.55,  0.83)
c) p1=( 2.00,  3.00)       d) p1=(2, 3)
e) p1=( 1.41,  1.73)
```

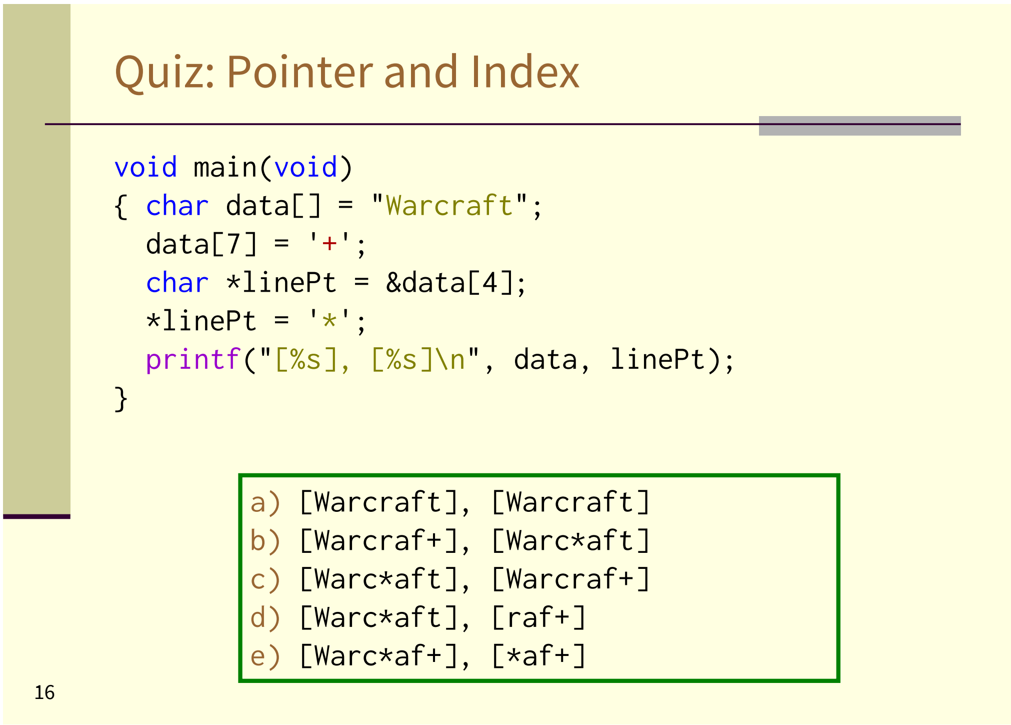


Pointer and Index to the Same Place

```
void main(void)
{ char data[] = "Hello World";
  data[2] = 'X';
  char *linePt = &data[3];
  *linePt = 'Z';
  printf("[%s], [%s]\n", data, linePt);
}
```

output:

```
[HeXZo World], [Zo World]
```



Quiz: Pointer and Index

```
void main(void)
{ char data[] = "Warcraft";
  data[7] = '+';
  char *linePt = &data[4];
  *linePt = '*';
  printf("[%s], [%s]\n", data, linePt);
}
```

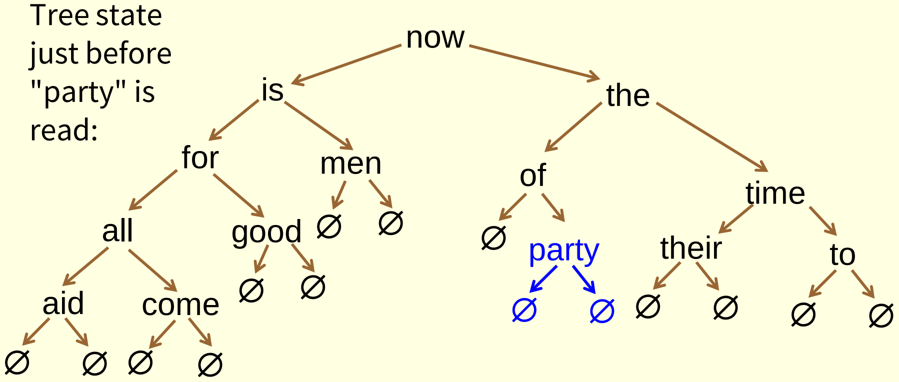
```
a) [Warcraft], [Warcraft]
b) [Warcraf+], [Warc*aft]
c) [Warc*aft], [Warcraf+]
d) [Warc*aft], [raf+]
e) [Warc*af+], [*af+]
```

Binary Tree: Kernighan and Ritchie 6.5

Read a file and count the occurrences of each word.

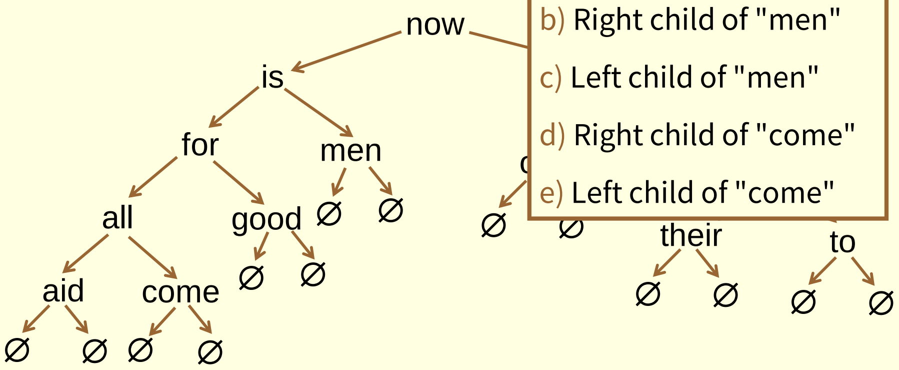
`now is the time for all good men to come to the aid of their party`

Tree state just before "party" is read:



Quiz: Binary Tree - Insert

Given the binary tree in the state shown below, if a node is added with the name "hello", where would it be placed?



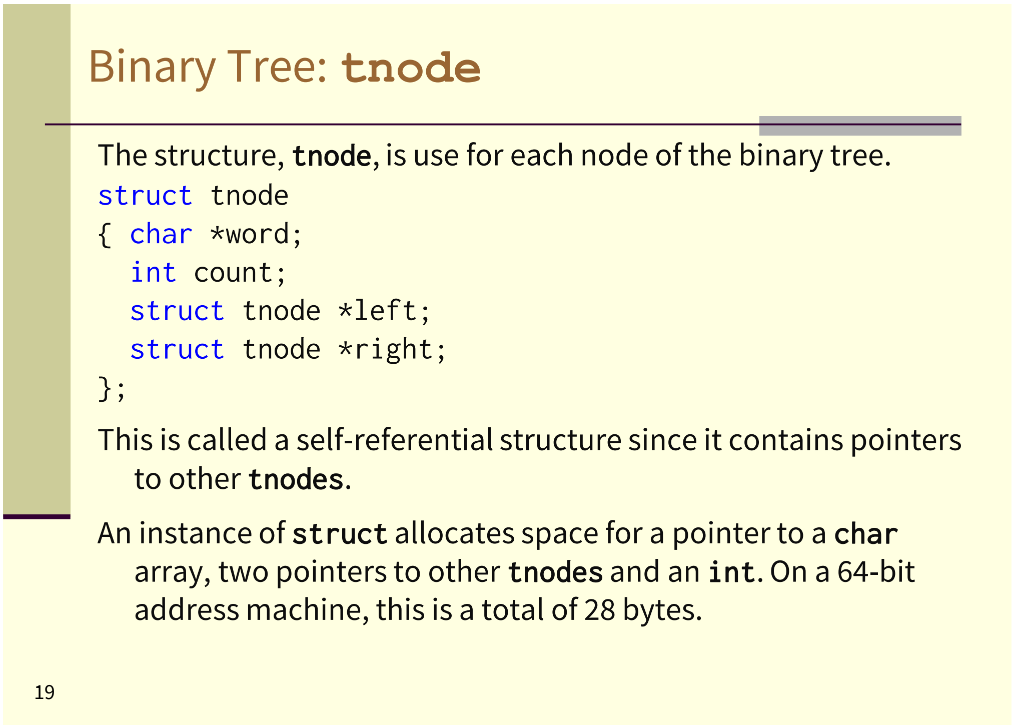
a) Right child of "good"

b) Right child of "men"

c) Left child of "men"

d) Right child of "come"

e) Left child of "come"



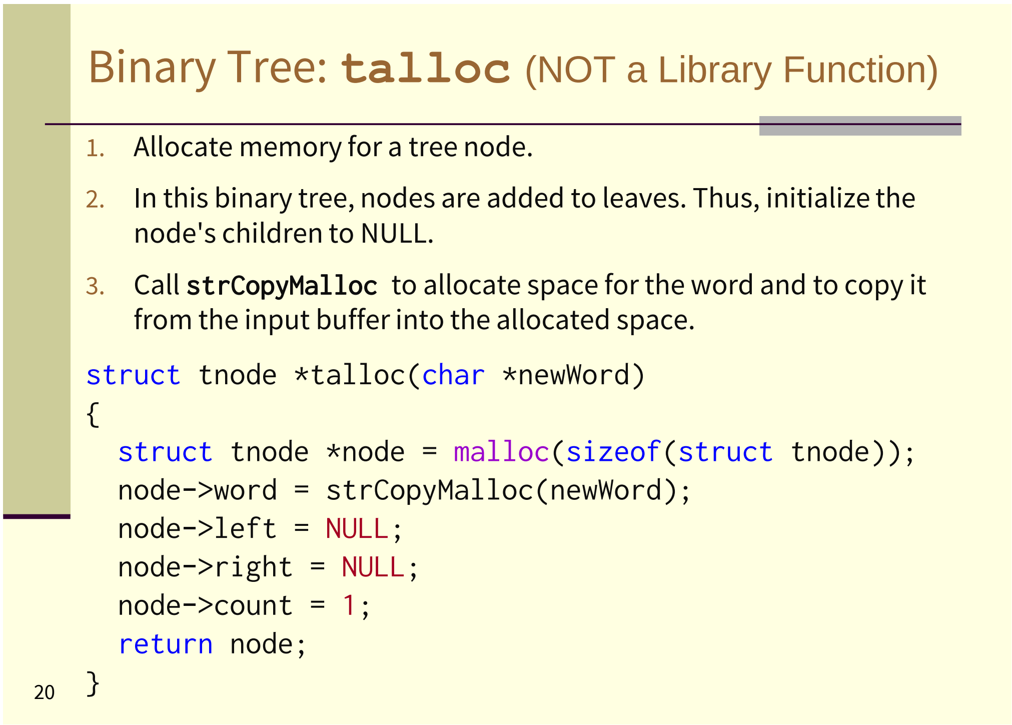
Binary Tree: tnode

The structure, **tnode**, is use for each node of the binary tree.

```
struct tnode
{ char *word;
  int count;
  struct tnode *left;
  struct tnode *right;
};
```

This is called a self-referential structure since it contains pointers to other **tnodes**.

An instance of **struct** allocates space for a pointer to a **char** array, two pointers to other **tnodes** and an **int**. On a 64-bit address machine, this is a total of 28 bytes.



Binary Tree: talloc (NOT a Library Function)

1. Allocate memory for a tree node.
2. In this binary tree, nodes are added to leaves. Thus, initialize the node's children to NULL.
3. Call **strCopyMalloc** to allocate space for the word and to copy it from the input buffer into the allocated space.

```
struct tnode *talloc(char *newWord)
{
  struct tnode *node = malloc(sizeof(struct tnode));
  node->word = strCopyMalloc(newWord);
  node->left = NULL;
  node->right = NULL;
  node->count = 1;
  return node;
}
```

Binary Tree: strCopyMalloc

1. Allocate memory for a copy of **newWord**.
2. Copy each character from **newWord** into the allocated block.
3. Return a pointer to the start of the allocated block.

```
char *strCopyMalloc(char *source)
{
  char *sink;

  sink = malloc(strlen(source)+1);

  if (sink != NULL) strcpy(sink, source);
  return sink;
}
```

Binary Tree: Memory Leaks

In Kernighan and Ritchie's binary tree, *Memory is allocated and never freed!!!!*

MEMORY LEAK WARNING: DO NOT free a node until:

a) Its children have been freed, or pointers to its children have been saved somewhere else.

b) Its **word** has been freed.

```
  struct tnode *root;
  root = talloc("Memory");
  root->right = talloc("Leak");
  root->left = talloc("Bad");
  free(root);  //Leaves "unreachable" memory.
```

valgrind

Valgrind is an open source programming tool for detection of memory leak, invalid memory usage and profiling.

The name valgrind comes from the main entrance to Valhalla in Norse mythology.

```
valgrind a.out   //Runs a.out in a virtual machine.
```

If **a.out** is the Binary Tree code in the textbook:

```
...
==24825== LEAK SUMMARY:
==24825==    definitely lost: 32 bytes in 1 blocks
==24825==    indirectly lost: 88 bytes in 5 blocks
...
```

Binary Tree: **freeSubTree**

Recursive function that frees all allocated memory in a subtree.

Any references to **node** (such as would be in node's parent) MUST NOT BE USED AFTER calling this. **Best practice is to set such references to NULL.**

```
void freeSubtree(struct tnode *node)
{
  if (node == NULL) return;

  freeSubtree(node->left);
  freeSubtree(node->right);

  free(node->word);
  free(node);
}
```

Is this done here?

If not, could it be done here?

If so, between which lines and with what code?

Binary Tree: Simple Tests Case

This **main()** demonstrates usage and offers a simple test of creating, setting printing and freeing **tnode**.

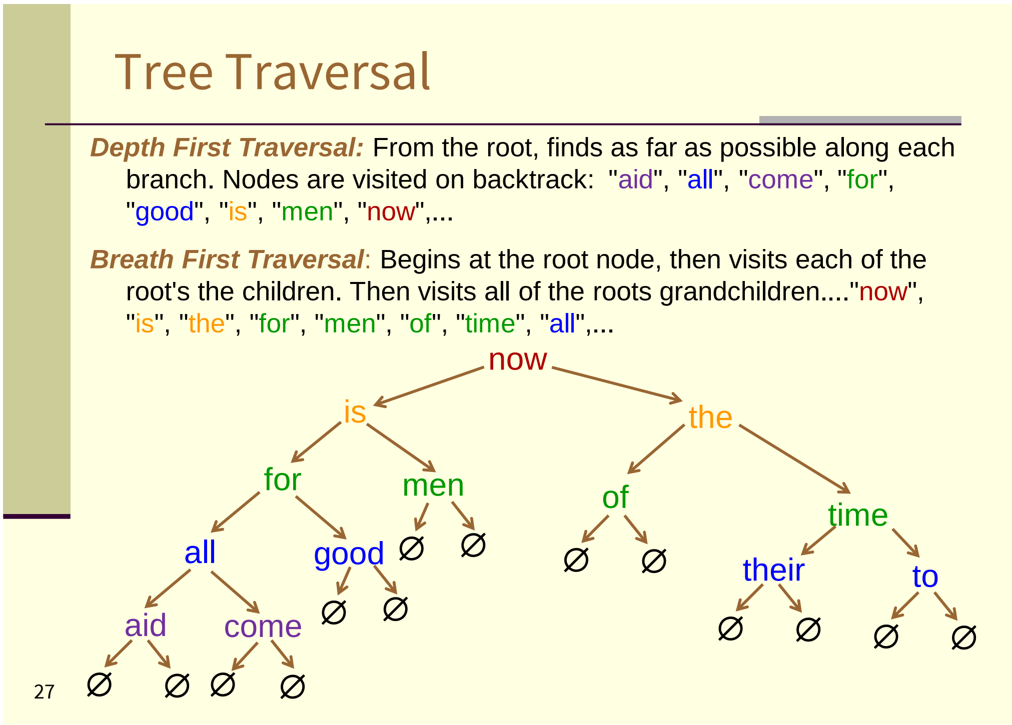
```
void main(void)
{ struct tnode *root;
  root = talloc("joel");
  root->left = talloc("cool");
  root->right = talloc("inspirational");

  printf("node: %s (L)=%s, (R)=%s\n", root->word,
     root->left->word,    root->right->word);

  freeSubtree(root);
  root = NULL; //"Best practice" (no effect on valgrind)
}
```

Binary Tree: No Leaks Are Possible

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct tnode
{ //fields
};

char *strCopyMalloc(char *source)
{ //body
}
```

```
struct tnode *talloc(char *newWord)
{ //body
}

void freeSubtree(struct tnode *node)
{ //body
}

void main(void)
{ //body
}
```

```
node: joel (L)=cool, (R)=inspirational
==24066== HEAP SUMMARY:
==24066==     in use at exit: 0 bytes in 0 blocks
==24066==   total heap usage:
                       6 allocs, 6 frees, 120 bytes allocated
==24066==
==24066== All heap blocks were freed -- no leaks are possible
```



Tree Traversal

Depth First Traversal: From the root, finds as far as possible along each branch. Nodes are visited on backtrack: "aid", "all", "come", "for", "good", "is", "men", "now",...

Breath First Traversal: Begins at the root node, then visits each of the root's the children. Then visits all of the roots grandchildren...."now", "is", "the", "for", "men", "of", "time", "all",...

now

is the

for men of time

all good ∅ ∅ ∅ ∅ their to

aid come ∅ ∅ ∅ ∅ ∅ ∅

∅ ∅ ∅ ∅