



COMP327

Mobile Computing

Session: 2011-2012

**Lecture Set 2c - Arrays, Pointers and
Memory Management**

In these Tutorial Slides...

- We cover static and dynamic memory management
 - Arrays, and how they are stored
 - Pointers, and pointer arithmetic
 - Dynamic Memory Management
 - Structs and Data Records



Arrays in C

- Arrays provide a way of storing several associated homogeneous values
 - Each element in the array is the same size
- Normally declared statically or dynamically
 - Static allocation: number of elements is explicitly defined
 - Dynamic allocation: no size is initially stated, but memory must be allocated later
- Elements can then be indexed
 - First element is index 0, to index size - 1
 - However, bounds are not checked!!!

All elements are of the same type

```
int array[10];
int b;

array[0] = 3;
array[9] = 4;
array[10] = 5;
array[-1] = 6;
```

First element (index 0)

Last element (index 9)

Allowed, and will cause no compiler or runtime error **BUT**, may overwrite other variables (e.g. b)

Computer Memory

- Data is stored in memory slots
- Each slot has a unique address
- Different architectures have different size slots
 - E.g. slots may be 16-bit, 32bit etc
 - Therefore some data types may span multiple slots, or use parts of slots
- Normally, the size of memory slots is unimportant from the programmer's perspective

| Address | Value | Content | |
|---------|--------|-------------------|--|
| 0x1000 | 37 | int i = 37; | |
| 0x1001 | 0x1000 | int *p = &i; | <i>Pointer to an int</i> |
| 0x1002 | 38.882 | float f = 38.882; | <i>Float takes multiple slots</i> |
| 0x1003 | | | |
| 0x1004 | 56 | short s = 56; | <i>short uses half a slot</i> |
| 0x1005 | 609 | int i[0] = 609; | <i>Array of two ints</i> |
| 0x1006 | 38 | int[1] = 38; | |
| 0x1007 | 'h' | char s[0] = 'h'; | <i>Array of four chars, containing the string "hi"</i> |
| 0x1008 | 'i' | char s[1] = 'i'; | |
| 0x1009 | '\0' | char s[2] = '\0'; | |
| 0x100A | '?' | char s[3] = ???; | |
| 0x100B | ... | | |

Memory Addresses

- Storage cells are typically viewed as being byte-sized
 - Usually the smallest addressable unit of memory
 - Few machines can directly access bits individually
 - Such addresses are sometimes called *byte-addresses*
- Memory is often accessed as words
 - Usually a word is the largest unit of memory access by a single machine instruction
 - A *word-address* is simply the byte-address of the word's first byte

Getting the size of an object

- sizeof() returns the size of an object in bytes
- Can be used to find the size of a statically defined array (i.e. number of elements), or of a typedef struct

```
int main(int argc, char **argv)
{
    char buffer[10]; /* Array of 10 chars */

    strncpy(buffer, argv[1], sizeof(buffer) - sizeof(char));

    /* Set the last element of the buffer equal to null */
    buffer[sizeof(buffer) - 1] = '\0';
}
```

- commonly used to get the size of a datatype for memory allocation

```
/* pointer to type int, used to reference our allocated data */
int * pointer = malloc(sizeof(*pointer) * 10);
```

Computer Memory

- Altering the value of a variable is equivalent to changing the content of memory

- `i = 40;`
- `a[1] = 207;`
- `s[0] = 's';`

| Address | Value | Content | |
|---------|------------|-------------------------|--|
| 0x1000 | 40 | int i = 40; | |
| 0x1001 | 0x1000 | int *p = &i; | <i>Pointer to an int</i> |
| 0x1002 | 38.882 | float f = 38.882; | <i>Float takes multiple slots</i> |
| 0x1003 | | | |
| 0x1004 | 56 | short s = 56; | <i>short uses half a slot</i> |
| 0x1005 | 609 | int a[0] = 609; | <i>Array of two ints</i> |
| 0x1006 | 207 | int a[1] = 207; | |
| 0x1007 | 's' | char s[0] = 's'; | <i>Array of four chars, containing the string "si"</i> |
| 0x1008 | 'i' | char s[1] = 'i'; | |
| 0x1009 | '\0' | char s[2] = '\0'; | |
| 0x100A | ? | char s[3] = ???; | |
| 0x100B | ... | | |

Computer Memory

- A Pointer stores the address of another entity
- It refers to a memory location

```
int i=40;      /* integer declaration */
int *p;       /* declare a pointer */
p = &i;       /* store address of i in p */

/* refer to value at address p */
printf("Value at p is %d\n", *p);

/* Display value of p */
printf("Value of p: %d\n", p);
```

Output

```
Value at p is 40
Value of p: 0x1001
```

| Address | Value | Content | |
|---------|--------|-------------------------|---|
| 0x1000 | 40 | int i = 40; | |
| 0x1001 | 0x1000 | int *p = &i; | Pointer to an int |
| 0x1002 | 38.882 | float f = 38.882; | Float takes multiple slots |
| 0x1003 | | | |
| 0x1004 | 56 | short s = 56; | short uses half a slot |
| 0x1005 | 609 | int a[0] = 609; | Array of two ints |
| 0x1006 | 207 | int a[1] = 207; | |
| 0x1007 | 's' | char s[0] = 's'; | Array of four chars, containing the string "si" |
| 0x1008 | 'i' | char s[1] = 'i'; | |
| 0x1009 | '\0' | char s[2] = '\0'; | |
| 0x100A | '?' | char s[3] = '???'; | |
| 0x100B | ... | | |

Pointers in C

- In C, every value is stored somewhere in memory
 - Can therefore be identified with that address.
 - Such addresses are called pointers.
- Because C is designed to allow programmers to control data at the lowest level, pointers can be manipulated just like any other kind of data.
 - In particular, you can assign one pointer value to another, which means that the two pointers end up indicating the same data value.
- Pointers are references to an object or element in memory, not the object or element itself.

Declaring a Pointer Variable in C

- Pointers are differentiated from regular variables by using the “*” syntax

```
int x;  
int *px;
```

- Declaring a variable allocates the memory for that variable
- Declaring a pointer allocates the memory for the pointer
 - Some other action is needed to create the memory for the object the pointer ultimately points to
 - This is similar to the issue of creating instances in JAVA.

Pointer Operators in C

- There are two main operators
 - The address-of operator “&”
 - This is written before a variable name (or any expression to which you could assign a value) and returns the address of that variable.
 - Thus, the expression `&total` gives the address of `total` in memory.
 - The dereference operator “*”
 - This is written before a pointer expression and returns the actual value to which the pointer points.
 - Example:

```
double x = 2.5;
double *px = &x;
```

 - Here, `px` points to the variable `x`, and `*px` contains the value 2.5
- If a pointer isn't pointing to anything, it can be assigned to **NULL**
 - Declared pointers may have arbitrary values, so when it is not pointing to something meaningful, it is a good idea to set its value to **NULL**

Example of pointers

The declarations of x, y, and z should be familiar.

A pointer is declared by specifying

*type * variable;*

As with normal variables, pointers can be declared and defined simultaneously

The value of ip contains the memory address (or location) of the variable x

```
int x=1, y=2, z[10];
int *pp;      /* pp is a pointer to int */
int *ip = &x; /* ip points to x */

y = *ip;      /* y is now 1 */
*ip = 3;      /* x is now 3 */
*ip += 5;     /* x is now 8 */
ip = &z[0];   /* ip now points to z[0] */
```

Dereferenced pointers can appear in any expression, and act just as a normal variable

If we change the value of the dereferenced pointer, that will also change the value of the variable the pointer points to

However, if we change the value of the pointer itself, then it simply points to another variable or location in memory

Pointers and Function Arguments

- Pointers can be passed to functions as arguments
- Provide a way of “side-affecting” variables
 - When a function is called, copies of the arguments are passed to the function
 - Changing the values of these copied arguments **does not affect their** value in the calling code
 - By passing the addresses of variables, a copy of the address is made
 - However, they still point to the calling code’s variables in memory, which **can** be changed

```
/* THIS WILL NOT WORK */
void swap (int x, int y) {
    int tmp;

    tmp = x;
    x = y;
    y = tmp;
}

swap(a,b); /* This fails */
```

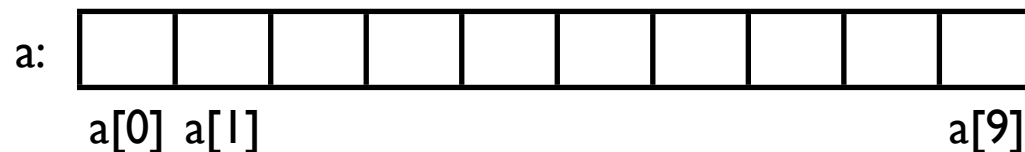
```
/* THIS IS CORRECT */
void swap (int *px, int *py) {
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;
}

swap(&a,&b); /* This succeeds */
```

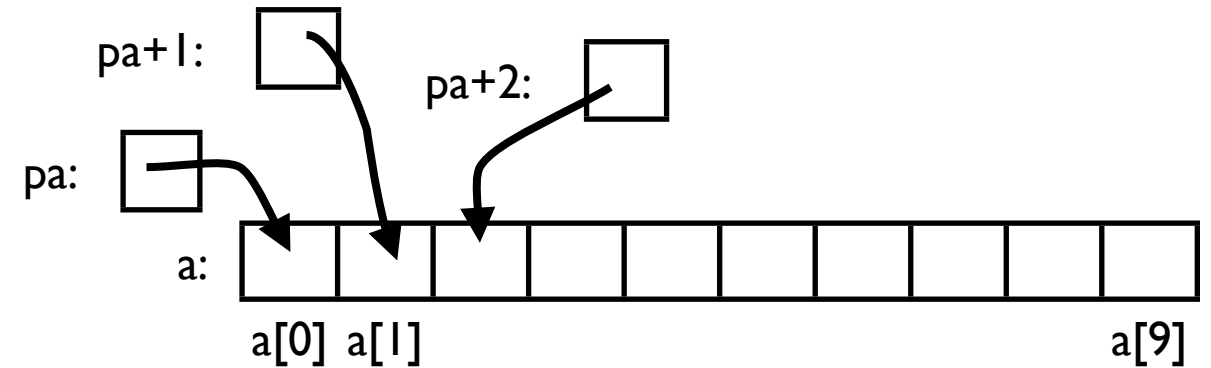
Pointers and Arrays

- In C, there is a strong relationship between pointers and arrays
 - Any operation that can be done by an array subscript can be done with pointers
 - Pointers are usually **faster**
 - Consider the declaration `int a[10];`



- This defines an array of size 10
 - i.e. a contiguous block of 10 ints in memory that can be accessed using `a[0]`, `a[1]`, ..., `a[9]`

Pointers and Arrays



```
int a[10];
int *pa;

pa = &a[0];    /* points to first element in a */
x = *pa;      /* x contains a copy of the element in a[0] */
y = *(pa+1);  /* y contains a copy of the element in a[1] */
```

- Subsequent array elements can be accessed by incrementing the value of a pointer, pointing to an array
 - This works **regardless** of the type or size of the array
 - As long as the pointer type is the same as the array type
- Array names are also pointers
 - `a[i]` is equivalent to `*(a+i)`
 - `a` is equivalent to `&a[0]` (i.e. the address of the first element)
- However, pointers are variables, array names are special
 - Cannot assign values to array names, or change their value

```
int a[10];
int *pa;

pa = a;    /* legal */
pa++;     /* legal */

a = pa;    /* illegal */
a++;     /* illegal */
```


Pointers, Arrays and Functions

- When an array name is passed to a function
 - the address of the first element is actually passed
- Empty arrays are just pointers
 - `char s[];` and `char *s;` are equivalent
 - However, `char *s` is more common

```
/* strlen: return length
 * of a string
 */
int strlen(char *s) {
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return n;
}
```

```
char array[20];
char *ptr = &array[0];

strlen("hello world");
strlen(array);
strlen(ptr);
...
```

Advantages of C pointers

- Pass values by reference
 - When calling methods, if a variable is passed to a method, a copy is made
 - This takes up space on the stack
 - The resulting value is immutable
 - By passing a pointer
 - Takes up less space if the pointer refers to a large object such as an array or string
 - Methods can side-effect external data
 - The method can write to the memory pointed to by the pointer, which is accessible by the calling function
 - Useful if a number of values are to be returned

Pointer Arithmetic

- If `p` is a pointer to some element, `p++` increments `p` to point to the subsequent element
 - This is regardless of the type of `p`
 - The “value” added to `p` is equivalent to the size of the element `p` points to
 - Therefore the programmer doesn’t have to worry about the size of type
- Arithmetic operators and expressions can be applied to pointers
 - e.g. relations such as `==`, `<`, `>=` etc can be used to compare pointers
 - e.g. `p < q` is true if `p` points to an earlier member of some array than `q`
 - Arithmetic is meaningful when referring to some allocated structure (e.g. an array), but not to separate variables

```
/* strlen: return length of a string */
int strlen(char *s) {
    char *p = s;
    while (*p != '\0')
        p++;
    return p-s;
}
```

The difference between these pointers is equivalent to the number of chars between them...
... i.e. the string length!

Example of using char * pointers

- These versions of strcpy illustrate how pointers and arrays are used interchangeably
- Note that in the last example, the end of string character '\0' is equivalent to zero , thus the assignment will return cause the while loop to exit

```
/* strcpy: Copy t to s; array subscript version */  
void strcpy(char *s, char *t) {  
    int i = 0;  
    while ((s[i] = t[i]) != '\0')  
        i++;  
}
```

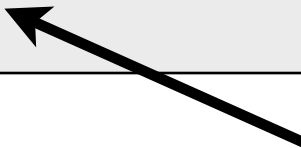
```
/* strcpy: Copy t to s; pointer version */  
void strcpy(char *s, char *t) {  
    while ((*s = *t) != '\0') {  
        s++;    /* go to next character in s */  
        t++;    /* go to next character in t */  
    }  
}
```

```
/* strcpy: Copy t to s; advanced pointer version */  
void strcpy(char *s, char *t) {  
    while (*s++ = *t++)  
        ;  
}
```

Generic Pointers

- **void *** - a pointer to anything
 - Loses all information about what type of object is pointed to
 - Reduces the effectiveness of type-checking
 - Can't use pointer arithmetic
 - However, valuable when developing generic functions that can manage pointers to different types of objects
 - e.g. container data structures such as linked lists.

```
void    *p;  
int     i;  
char    c;  
...  
p = &i;  
p = &c;  
putchar(*(char *)p);
```



Note that we cast the **void *** pointer to another pointer type (e.g. **char ***), so that the compiler knows how to handle this, i.e. **(char *)p**

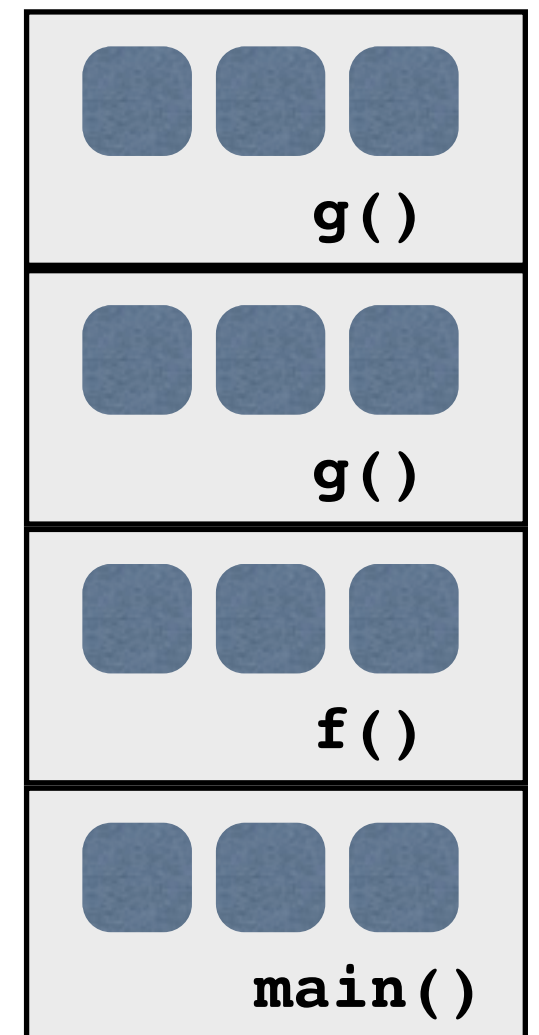
We then reference the value and pass it to **putchar**

Overview of Memory Management

- **Stack-allocated memory**

- When a function is called, memory is allocated for all of its parameters and local variables.
- Each active function call has memory on the stack (with the current function call on top)
- When a function call terminates, the memory is deallocated (“freed up”)

- For example, `main()` calls `f()`, which then calls `g()` which calls itself...



Overview of Memory Management

- **Heap-allocated memory**

- This is used for persistent data, that must survive beyond the lifetime of a function call
 - global variables
 - dynamically allocated memory – C statements can create new heap data (similar to new in Java/C++)
- Heap memory is allocated in a more complex way than stack memory
- Like stack-allocated memory, the underlying system determines where to get more memory
 - the programmer doesn't have to search for free memory space!

Allocating new heap memory

- `void *malloc(size_t size);`
- Allocate a block of size bytes,
 - return a pointer to the block
 - (NULL if unable to allocate block)
- The returned pointer should be cast to the appropriate type

Note that the size allocated is `n` times `sizeof(char)`, where `n` is the length of string `s` plus 1 for `'\0'`.

When allocating memory for `char *` elements, the `sizeof()` is often omitted, as `sizeof(char) = 1`

```
/* make a duplicate of s */
char *strdup(char *s) {
    char *p;

    /* Allocate space for the length
     * of the string +1 for '\0'
     */
    p = (char *) malloc((strlen(s)+1) * sizeof(char));
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

Allocating new heap memory

- `void *calloc(size_t num_elements, size_t element_size);`
- Allocate a block of `num_elements * element_size` bytes,
 - initialise every byte to zero,
 - return pointer to the block
 - (NULL if unable to allocate block)

```
int *ip;  
  
ip = (int *) calloc(n, sizeof(int));
```

- Whilst this is similar to `malloc`, it is used less frequently
 - Useful if the initialisation is necessary
 - However, often there is no need to do any initialisation, as the new object is defined after allocation

Allocating new heap memory

- `void *realloc(void *ptr, size_t new_size);`
- Given a previously allocated block starting at `ptr`,
 - change the block size to `new_size`,
 - return pointer to resized block
 - If block size is increased, contents of old block may be copied to a completely different region
 - In this case, the pointer returned will be different from the `ptr` argument, and `ptr` will no longer point to a valid memory region
- If `ptr` is `NULL`, `realloc` is identical to `malloc`
- Note: may need to cast return value of `malloc/calloc/realloc`:
 - `char *p = (char *) malloc(BUFFER_SIZE);`

Deallocating heap memory

- `void free(void *pointer);`
- Given a pointer to previously allocated memory,
 - put the region back in the heap of unallocated memory
- Note: easy to forget to free memory when no longer needed...
 - especially if you're used to a language with "garbage collection" like Java
 - This is the source of the notorious "**memory leak**" problem
 - Difficult to trace – the program will run fine for some time, until suddenly there is no more memory!

Common memory errors

- There are several memory errors that can cause problems with memory
 - These can occur through sloppy programming
 - Not always easy to spot, however
 - Often occur with specific data scenario
 - Or worse, simply because of the presence of other processes running concurrently!!!
- Problems can often be responsible for security holes
 - e.g. when writing into other parts of memory due to stack overflow
- Problems include
 - Using memory that you have not initialised
 - Using memory that you do not own
 - Using more memory than you have allocated
 - Using faulty heap memory management

Using memory you've not initialised

- Uninitialised memory read
- Uninitialised memory copy
- not necessarily critical – unless a memory read follows

```
void foo(int *pi) {
    int j;
    *pi = j;
    /* ERROR: j is uninitialised, copied into *pi */
}

void bar() {
    int i=10;
    foo(&i);
    printf("i = %d\n", i);
    /* ERROR: Using i, which is now junk value */
}
```

Using memory you don't own

- Null pointer read/write
- Zero page read/write

```
/* Define a new data structure */
typedef struct node {
    struct node* next;
    int val;
} Node;

int findLastNodeValue(Node* head) {
    /* Next line fails if the pointer is null */
    while (head->next != NULL) {
        head = head->next;
    }
    return head->val; /* Zero page access */
}
```

This creates a new structure, and defines it as a variable type with two sub elements, a pointer to another instance of that note, and a value (int)

Given a pointer to a struct, the expression `->` refers to the sub-element in the structure
`x->y` is equivalent to `(*x).y`

What happens if head is NULL ???

Using memory you don't own

- Beyond stack read/write

`result` is a local array name, allocated on the stack

Function returns pointer to stack memory. However, this won't be valid after the function returns

```
char *append(const char* s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[128];

    int i=0, j=0;
    for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
        result[i] = s1[j];
    }
    for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
        result[i] = s2[j];
    }
    result[++i] = '\0';
    return result;
}
```

Using memory you haven't allocated

- Array bound read/write

Memory is allocated for 10 chars. However, if a string is to be stored in this array, then space should be made for the end of string character.

We call `strncpy` which copies `n` characters (in this case 10) from one string to another.

This will succeed.

```
void genArrayBoundError() {
    const char *name = "Safety Critical";
    char *str = (char*) malloc(10*sizeof(char));

    strncpy(str, name, 10);
    str[11] = '\0'; /* Writing out of bounds */
    printf("%s\n", str); /* Error will occur */
}
```

However, there are two problems here.

- 1) if we want to write at the end of an array of size `n`, the last element is in `arr[n-1]`
- 2) this line is writing beyond this.

Whilst this line will work, it is overwriting some other part of memory. At best, the code will crash here...!

Faulty Heap Management

- Memory Leak

2) Memory is allocated for 8 ints. However, pi is a global, and had previously been pointing to other memory, whose reference is now lost

3) The newly allocated memory is freed (returned to the heap), but we still have no access to the original leaked memory

```
int *pi;
void foo() {
    pi = (int*) malloc(8*sizeof(int));
    /* Allocate memory for pi */
    /* Oops, leaked the old memory pointed to by pi */
    ...
    free(pi); /* foo() is done with pi, so free it */
}
void main() {
    pi = (int*) malloc(4*sizeof(int));
    /* Expect MLK: foo leaks it */
    foo();
}
```

1) We allocate memory to four ints, and store a pointer to this memory in pi, which is global.

Structures

- A compound data type, consisting of different types
 - grouped together by a single name
 - analogous to data stored in a class within OOP languages
 - except that there are no accessor methods, just the data structure
- Help to organise complex data, as the new object can be managed as a single entity

Struct Syntax

```
struct name {  
    variable list  
} var;
```

- The keyword `struct` introduces the structure definition
 - a list of variables, or members
 - the names of these are local to the structure, and could appear in other structures
- The struct declaration defines a type
 - Declarations of variables of this type follow the close-brace
 - Once defined, the struct can be used to declare other variables
- The name of the struct is optional, but provides a way of referring to the struct type
- Elements in the structure are accessed by the dot syntax
- `sizeof()` can be used to get the size of a struct for memory allocation

```
struct point {  
    int x;  
    int y;  
} a, b;  
  
struct point c;  
  
a.x = 5;  
a.y = 8;  
b = {3,4};
```

Structures and Functions

- When passing a structure to a function, the whole structure is copied
 - just as with other variables
- structs can be returned from functions
 - values are again copied, so receiving structure must be memory allocated
- With large structs, it is more efficient to pass a pointer!!!

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

/* makepoint: make a point
 * from x and y components
 */
struct point makepoint(int x, int y) {
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
}

main() {
    struct point middle, llb, urt;
    llb = makepoint(0,0);
    urt = makepoint(XMAX, YMAX);

    middle = makepoint(
        (llb.x + urt.x)/2,
        (llb.y + urt.y)/2);
}
```

Pointers to Structures

- Pointers to structures are like ordinary pointers

```
struct point *pp;    /* ptr to struct */
struct point origin;

pp = &origin;
printf("origin: (%d,%d)\n", (*pp).x, (*pp).y);
```

- Parentheses are needed to access the struct elements

- this is due to precedence
 - . is higher than *

```
*pp.x /* element x in pp is a pointer!!! */
```

- is illegal here as x is not a pointer

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

/* makepoint: make a point
 * from x and y components
 */
void makepoint(struct point *ppt,
               int x,
               int y) {
    (*ppt).x = x;
    (*ppt).y = y;
}

main() {
    struct point middle, llb, urt;

    makepoint(&llb, 0,0);
    makepoint(&urt, XMAX, YMAX);
    makepoint(&middle,
              (llb.x + urt.x)/2,
              (llb.y + urt.y)/2);
}
```

Pointers to Structures

- Because pointers to structures are so common, an alternative notation exists
- Again, precedence of this notation is very high, so care is needed

```
struct {
    int len;
    char *str;
} *p;
...
++p->len;      /* increments len, not p) */
(++p)->len;    /* increments p, not len */
*p->str;       /* fetches what str points to */
(*p->str)++;   /* increments what str points to */
```

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

/* makepoint: make a point
 * from x and y components
 */
void makepoint(struct point *ppt,
               int x,
               int y) {
    ppt->x = x;
    ppt->y = y;
}

...
```

p->member-of-structure

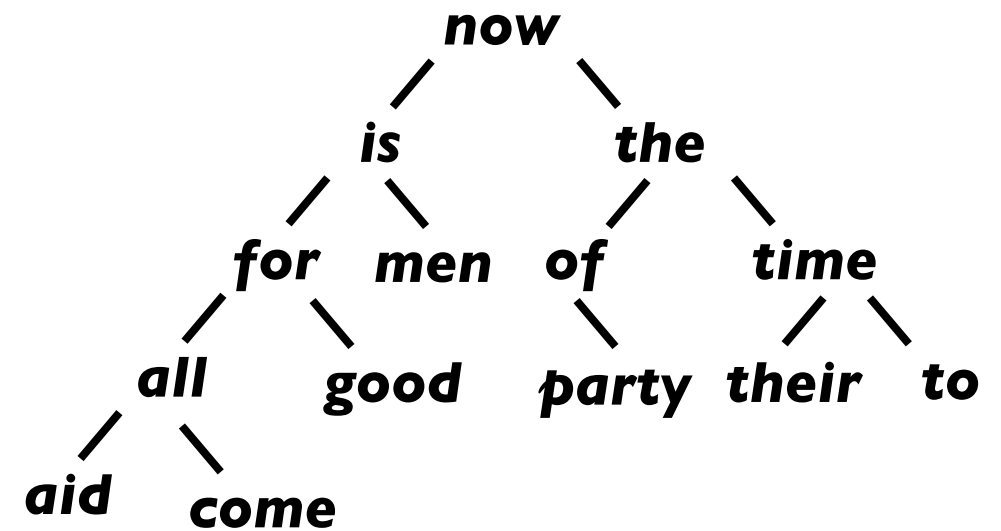
Self-referential Structures

- Many data models use a structure that is self referential
 - i.e. the structure is a node within a linked list, tree, etc.
 - This data may be held in some order, such as sorted alphabetically, or given some count, etc.
 - The number of nodes depends on the quantity of data being processed
- However, how can a struct reference itself before it has been declared?
 - It is illegal for a struct to contain an instance of itself
 - Pointers can be used to reference other nodes, as the pointer is the address of another node, not the node itself

Example: Binary Tree

The data structure opposite represents a binary tree, with strings (char *) organised so that at any node the left subtree contains only words that are lexicographically less than the word at the node, and the right subtree contains only words that are greater.

This is the tree for the sentence “*now is the time for all good men to come to the aid of their party*”, as built by inserting each word as it is encountered.



The struct defined below can be used to represent each node in this structure. As each word is encountered, a new node is filled in, and its address is inserted into the tree, by traversing the tree lexically, until a leaf node is found (i.e. either **left** or **right** is null).

Note: this assumes that each time a node is created, it is initialised so that the **left** and **right** pointers are defined as NULL.

```
struct tnode { /* a node in a binary tree */
    char *word; /* points to the payload */
    int count; /* number of occurrences of word */
    struct tnode *left; /* left child */
    struct tnode *right; /* right child */
};
```

Adding the word to the binary tree

```
struct tnode *talloc(void);           /* allocate memory for a node */
char *strdup(char *);                 /* allocate memory and copy string */

/* addtree: add a node with w, at or below p */
struct treenode *addtree(struct tnode *p, char *w) {
    int cond;

    if (p == NULL) {                  /* a new word has arrived */
        p = talloc();                 /* make a new node */
        p->word = strdup(w);          /* copy the string */
        p->count = 1;                 /* initialise the count */
        p->left = p->right = NULL;    /* set the pointers to NULL */
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;                   /* repeated word */
    else
        if (cond < 0)                 /* less than into left subtree */
            p->left = addtree(p->left, w);
        else                           /* greater than into right subtree */
            p->right = addtree(p->right, w);
    return p;
}
```

Typedef

- The construct typedef is used for creating new data type names
- Can be useful to create “meaningful” data types

```
typedef int Length;  
...  
Length len, maxlen;  
Length *lengths[];
```

```
typedef char *String;  
...  
String p, lineptr[MAXLINES], alloc(int);  
int strcmp(String, String);  
p = (String) malloc(100);
```

- Note that the type being declared appears in the position of the variable name, not after typedef
- Syntactically, it is like extern, static, etc.
- Often, typedefs have capitalised names

Typedefs and structs

This code fragment defines a structure of a node in a binary tree that holds a payload, consisting of a char * (assuming the string has been allocated elsewhere) and a frequency count.

This could be used in a sorted binary tree, thus improving lookup time.

We need the name of the struct to be specified, as pointers to this struct appear in the struct definition

The name of the typedef appears after the struct declaration

```
typedef struct tnode {      /* a node in a binary tree */
    char *word;             /* points to the payload */
    int count;              /* number of occurrences of word */
    struct tnode *left;    /* left child */
    struct tnode *right;   /* right child */
} Treenode;

/* Function to create new tree nodes from heap */
Treenode *talloc(void) {
    return (Treenode *) malloc(sizeof(Treenode));
}
```

The return type of this function is a pointer to a Treenode

sizeof() returns the size of the new structure, and can be used when allocating memory

Unions

- A union is a variable that may hold (at different times) objects of different types and sizes
 - Can provide flexible storage
- Definition is similar to a struct, but **only one variable can be used at a time**
 - The size of the union is equivalent to the largest variable
- Can be challenging to use, however
 - programmer is responsible for knowing what the type of the stored value is
 - no automatic way of checking
- Access to a union is similar to that of a struct
 - `union-name.member`
 - `union-pointer->member`

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;

u.ival = 6;
u.sval = "hello";
```

Example of a union

utype is used to track the union type. Three symbolic constants have been defined, INT, FLOAT, STRING. An enum could also have been used.

```
struct {
    char *name;
    int flags;
    int utype;
    union {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```

A union element is defined, that stores one value at a time. As float is the biggest data type, this union will be the size of a float.

```
...
if (utype == INT)
    printf("%d\n", u.ival);
if (utype == FLOAT)
    printf("%f\n", u.fval);
if (utype == STRING)
    printf("%s\n", u.sval);
else
    printf("bad type %d in utype\n", utype);
```

This code fragment defines a symbol table which consists of an array of structs. Each struct contains several variables, including one that is used to track the type of the union variable.

The advantage of using this union, is that it reduces storage requirements, rather than maintaining space for three separate data types

utype is checked to determine the type of value in the union, before deciding how this will be used. Remember, only one data element can be stored at any one time.

To Summarise

- In this Tutorial Set we covered
 - Memory, arrays, and how data is stored
 - Memory allocation (static vs dynamic)
 - Problems with bad memory management
 - Pointers, their relationship with arrays
 - Pointer arithmetic
 - Structs, Typedefs and Unions
 - Self referential data structures

Exercises

- Exercise 5-4. Write the function `strend(s,t)`, which returns 1 if the string `t` occurs at the end of the string `s`, and zero otherwise.
- Exercise 5-5. Write versions of the library functions `strncpy`, `strncat`, and `strncmp`, which operate on at most the first `n` characters of their argument strings. For example, `strncpy(s,t,n)` copies at most `n` characters of `t` to `s`.
- Exercise 5-13. Write the program `tail`, which prints the last 10 lines of its input. The program should behave rationally no matter how unreasonable the input. Write the program so it makes the best use of available storage; lines should be stored in a sorted way (e.g. a tree or linked list), not in a two-dimensional array of fixed size.
- Exercise 6-4. Write a program that prints the distinct words in its input sorted into decreasing order of frequency of occurrence. Precede each word by its count.

Epilogue

- If you want to look at how code written in C can be **abused**, then check out the The International Obfuscated C Code Contest
 - <http://www.ioccc.org/>
 - The Goals:
 - To write the most Obscure/Obfuscated C program under the rules below.
 - To show the importance of programming style, in an ironic way.
 - To stress C compilers with unusual code.
 - To illustrate some of the subtleties of the C language.
 - To provide a safe forum for poor C code. :-)

```

extern int
errno
;char
grrr
r,
;main(
int argc
argv, argc )
r ;
char *argv[];{int
#define x int i, j,cc[4];printf("
x ;if (P( ! i ) choo choo\n" ) ;
& P(j )>2 ? j : i ){* argv[i++ +!-i]
; for (i= 0;; i++ );
_exit(argv[argc- 2 / cc[1*argc]|-1<<4 ] ) ;printf("%d",P("")));}}
P ( a ) char a ; { a ; while( a > " B "
/* - by E ricM arsh all- */); }

```

Epilogue

```

#include <stdio.h>
#include <malloc.h>
#define ext(a) (exit(a),0)
#define I " .:\';+<?F7RQ&%#*"
#define a "%s?\n"
#define n "0?\n"
#define C double
#define o char
#define l long
#define L sscanf
#define i stderr
#define e stdout
#define r ext (1)
#define s(O,B) L(++J,O,&B)!=1&&c>+q&&L(v[q],O,&B)!=1&&--q
#define F(U,S,C,A) t=0,*+J&&(t=L(J,U,&C,&A)),(!t&&c>+q&&!(t=L(v[q],U,\
&C,&A)))?--q:(t<2&&c>+q&&!(t=L(v[q],S,&A))&&--q
#define T(E) (s("%d",E),E||(fputs(n,i),r))
#define d(C,c) (F("%lg,%lg","%lg",C,c))
#define O (F("%d,%d","%d",N,U),(N&&U)||(fputs(n,i),r))
#define D (s("%lg",f))
#define E putchar

    C
    G=0,
    R
    =0,Q,H
    ,M,P,z,S
    =0,x=0
    , f=0;l b,j=0, k
    =128,K=1,V,B=0,Y,m=128,p=0,N
    =768,U=768,h[]={0x59A66A95,256
    ,192,1,6912,1,0,0},t,A=0,W=0,Z=63,X=23
    ;o*J,_;main(c,v)l c;o**v;{l q=1;for(;q<
    c ?(((J=v[q])[0]&&J[0]<48&&J++,((_= *J)<99|
    _/2== '2'| |(-1)/3=='\'| |_==107| |_/05*2==','| |
    >0x074)?( fprintf(i,a,v[q]),r):_>0152?(_/4>27?(_&1?
    O,Z=N,X=U): (W++,N=Z,U=X)):_&1?T(K):T(k)):_>103?(d(G,
    R ),j=1):_&1? d(S,x):D,q++),q--,main(c-q,v+q):A==0?(A=
    1,f|| (f=N/4.),b=((N-1)&017)<8),q=((N+7)>>3)+b)*U,(J=malloc(q)
    )||(perror("malloc"),r),S--=(N/2)/f,x+=(U/2)/f):A==1?(B<U?(A=2,V
    = 0,Q=x-B/f,j || (R=Q),W&&E('\n',e),E(46,i)): (W&&E('\n',
    e),E('\n',i ),h[1]=N,h[2]=U,h[4]=q,W|| (fwrite(h,1,32,
    e),fwrite (J,1,q,e),free(J),ext(0)):A==2?(V<N?(j?
    (H=V/f +S,M=Q):(G=V/f+S,H=M=0),Y=0,A=03):( (m&0x80
    ) || (m=0x80,p++),b&&(J[p++]=0),A=1,B++):( (Y
    <k&&(P=H*H)+(z=M*M)<4.)?(M=2*H*M+R,H=P-z
    +G,Y++):(W&&E(I[0x0f*(Y&K)/K],e),Y&K?J
    [p]&=~m:(J[p]=m),(m>=1)|| /*/
    (m=128,u--),A==6?ext(1):B<u
    . e=3,l=2*c*/( m
    =0x80,
    p++),V++
    ,A=0x2
    )
    ));
}

```

```

./a.out -text
.*** ** * * * ***** * ***** **
.* ***** * * * * * * ***** *
.* ***** * * * * * * * ***** *
.* ***** * * * * * * * * * *****
.***** * * * * * * * * * *****
.* ** * * * * * * * * * *****
.* ***** * * * * * * * * * *****
.* * * * * * * * * * * * * * *****
.* ***** * * * * * * * * * *****
.* ** * * * * * * * * * * * * * *****
.* ***** * * * * * * * * * *****
.* ***** * * * * * * * * * *****
.* ***** * * * * * * * * * *****
.* ***** * * * * * * * * * *****
.* ***** * * * * * * * * * *****
.* ***** * * * * * * * * * *****
.* ***** * * * * * * * * * *****

```