They now have a surround format for MP3, with players to give the "surround" effect while you walk around in headphones! Walk right and it'll be louder in your right ear...cool.

C String Standard Functions

- int strlen(char *string);
- compute the length of string
- int strcmp(char *str1, char *str2);
  - return 0 if str1 and str2 are identical (how is this different from str1 == str2?)
- char *strcpy(char *dst, char *src);
  - copy the contents of string src to the memory at dst. The caller must ensure that dst has enough memory to hold the data to be copied.

Pointers to pointers (1/4) ...review...

- Sometimes you want to have a procedure increment a variable?
  - What gets printed?

```c
void AddOne(int x)
{
    x = x + 1;
}

int y = 5;
AddOne(y);
printf("y = %d
", y);
```

Pointers to pointers (2/4) ...review...

- Solved by passing in a pointer to our subroutine.
- Now what gets printed?

```c
void AddOne(int *p)
{
    *p = *p + 1;
}

int y = 5;
AddOne(&y);
printf("*q = %d
", *q);
```

Pointers to pointers (3/4)

- But what if what you want changed is a pointer?
  - What gets printed?

```c
void IncrementPtr(int *p)
{
    p = p + 1;
}

int *q = A;
IncrementPtr(q);
printf("*q = %d
", *q);
```
Pointers to pointers (4/4)

- Solution! Pass a pointer to a pointer, called a handle, declared as `**h`
- Now what gets printed?

```c
void IncrementPtr(int **h)
{
    *h = *h + 1;
}

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(&q);
printf("*q = %d\n", *q);
```

Dynamic Memory Allocation (1/3)

- C has operator `sizeof()` which gives size in bytes (of type or variable)
- Assume size of objects can be misleading & is bad style, so use `sizeof(type)`
  - Many years ago an int was 16 bits, and programs assumed it was 2 bytes

Dynamic Memory Allocation (2/3)

- To allocate room for something new to point to, use `malloc()` (with the help of a typecast and `sizeof`):
  ```c
  ptr = (int *) malloc (sizeof(int));
  
  *Now, ptr points to a space somewhere in memory of size (sizeof(int)) in bytes.
  * (int *) simply tells the compiler what will go into that space (called a typecast).
  
  *malloc is almost never used for 1 var
  ptr = (int *) malloc (n*sizeof(int));
  
  *This allocates an array of n integers.
  ```

Dynamic Memory Allocation (3/3)

- Once `malloc()` is called, the memory location contains garbage, so don’t use it until you’ve set its value.
- After dynamically allocating space, we must dynamically free it:
  ```c
  free(ptr);
  
  *Use this command to clean up.
  ```

C structures : Overview

- A struct is a data structure composed for simpler data types.
  - Like a class in Java/C++ but without methods or inheritance.

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

void PrintPoint(point p)
{
    printf("(%d,%d)", p.x, p.y);
}
```
C structures: Pointers to them

• The C arrow operator (->) dereferences and extracts a structure field with a single operator.

• The following are equivalent:

```c
struct point *p;
printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```

How big are structs?

• Recall C operator sizeof() which gives size in bytes (of type or variable)

• How big is sizeof(p) ?

```c
struct p {
    char x;
    int y;
};
```

• 5 bytes? 8 bytes?

• Compiler may word align integer y

Which are guaranteed to print out 5?

I:   main() {
    int *a-ptr; *a-ptr = 5; printf("%d\n", *a-ptr);
}

II:  main() {
    int *p, a = 5;
    p = &a; ... 
    printf("%d\n", *p);
}

III: main() {
    int *ptr;
    ptr = (int*) malloc (sizeof(int));
    printf("%d\n", *ptr);
}

Peer Instruction

Which are guaranteed to print out 5?

I:   main() {
    int *a-ptr; *a-ptr = 5; printf("%d\n", *a-ptr);
}

II:  main() {
    int *p, a = 5;
    p = &a; ...
    printf("%d\n", *p);
}

III: main() {
    int *ptr;
    ptr = (int*) malloc (sizeof(int));
    printf("%d\n", *ptr);
}

How big are structs?

• Recall C operator sizeof() which gives size in bytes (of type or variable)

• How big is sizeof(p) ?

```c
struct p {
    char x;
    int y;
};
```

• 5 bytes? 8 bytes?

• Compiler may word align integer y

Bonus: Linked List Example

• Let’s look at an example of using structures, pointers, malloc(), and free() to implement a linked list of strings.

```c
struct Node {
    char *value;
    struct Node *next;
};
typedef Node *List;

/* Create a new (empty) list */
List ListNew(void)
{
    return NULL;
}

/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node = (struct Node*) malloc(sizeof(struct Node));
    node->value = (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```

Linked List Example

```c
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node = (struct Node*) malloc(sizeof(struct Node));
    node->value = (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```
**Linked List Example**

/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node = (struct Node*) malloc(sizeof(struct Node));
    node->value = (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}

**Linked List Example**

/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node = (struct Node*) malloc(sizeof(struct Node));
    node->value = (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}

**Linked List Example**

/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node = (struct Node*) malloc(sizeof(struct Node));
    node->value = (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}

**Linked List Example**

/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node = (struct Node*) malloc(sizeof(struct Node));
    node->value = (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}

**And in Conclusion...**

- Use handles to change pointers
- Create abstractions with structures
- Dynamically allocated heap memory must be manually deallocated in C.
  - Use malloc() and free() to allocate and deallocate memory from heap.

**Peer Instruction**

```c
int main(void)
{
    int A[] = {5, 10};
    int *p = A;
    printf("%u %d %d %d\n", p, *p, A[0], A[1]);
    p = p + 1;
    printf("%u %d %d %d\n", p, *p, A[0], A[1]);
    p = p + 1;
    printf("%u %d %d %d\n", p, *p, A[0], A[1]);
}
```

If the first printf outputs 100 5 5 10, what will the other two printf output?

1: 101 10 5 10
2: 104 10 5 10
3: 101 <other> 5 10 then 101 <3-others>
4: 104 <other> 5 10 then 104 <3-others>
5: One of the two printf causes an ERROR
6: I surrender!