More from Wednesday’s lecture

1. `#define` macros may go anywhere. Thereafter the name is replaced with the replacement text. It is usually good style to put all `#defines` at the top so that reordering code doesn’t cause bugs.

2. `void` * pointers used to be `char` * pointers (before ANSI C). Therefore, partially to maintain compatibility, `++` incrementing a `void` * pointer via increments it by 1 byte.

3. `const` type qualifier announces objects are not to be changed. Implementation-dependent storage and violation penalty.

---

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`?)

- `int 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\n", y);
```

- y = 6

---

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("*y = %d\n", *y);
```

```c
*q = 50
```

---

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

```c
50 60 70
```
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
```c
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.

Binky Pointer Video (thanks to NP @ SU)

```html
Binky Pointer Video (thanks to NP @ SU)
```

Administrivia

• One extra credit lab checkoff pt!
  • Sign up to get your lab checked off by the first hour and you will get 1 bonus checkoff point to count toward final grade. (Not 1/300, +1 out of 4 for that lab)
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.

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

```c
void PrintPoint(point p) {
    printf("(%d,%d)\n", p.x, p.y);
}
```

C structures: Pointers to them

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

```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)`?

```
struct p {
    char x;
    int y;
};
• 5 bytes? 8 bytes?
• Compiler may word align integer y
```

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", a);
}

III: main() {
    int *ptr;
    ptr = (int*) malloc(sizeof(int));
    *ptr = 5;
    printf("%d\n", *ptr);
}

```

Linked List Example

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

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

```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;
}

node: ?
list: ...
string: "abc"

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;
}

node: ?
list: ...
string: "abc"

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;
}

node: ?
list: ...
string: "abc"

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;
}

node: ...
list: ...
string: "abc"

“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.