Overview

- Pointers to Pointers
- Linked List
- Shift Operators

Review: C memory allocation

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack pointer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Space for saved procedure information
- Explicitly created space, e.g., malloc(); C pointers
- Variables declared once per program

Program

Review: C memory allocation

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Stack: Space for saved procedure information
- Heap: Explicitly created space, e.g., malloc(); C pointers

Static

- Variables declared once per program

Code

- Program

Pointer Arithmetic Summary

- `x = *(p+1)  \Rightarrow x = *(p+1) ;`
- `x = *p+1    \Rightarrow x = (*p) + 1 ;`
- `x = (*p)++   \Rightarrow x = *p ; *p = *p + 1 ;`
- `x = *p++ = (*p++) = *(p)++ = *(p++)`  
  \[ \Rightarrow x = *p ; p = p + 1 ; \]
- `x = +++p    \Rightarrow p = p + 1 ; x = *p ;`

Lesson?

- Avoid the confusing syntaxes!
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 (#1/5)

° 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);
```

Pointers to pointers (#2/5)

° 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);
```

Pointers to pointers (#3/5)

° 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
int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(q);
printf("*q = %d\n", *q);
```
Pointers to pointers (#4/5)

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);
```

```c
*q = 60
50 60 70
```

Pointers to pointers (#5/5)

```c
int **ptr1
int *ptr2
```

Recall: C Syntax; Arguments to main

To get the main function to accept arguments, use this:

```c
int main (int argc, char *argv[])
```

What does this mean?

- `argc` will contain the number of strings on the command line (the executable counts as one, plus one for each argument).
- `argv` is a pointer to an array containing the rest of the arguments as strings

```
   argv
   "baseconvert"
   "abx"
   "36"
   "19"
```

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

Peer Instruction

Which are guaranteed to print out 5?

I:   main() {
    int *a_ptr; *a_ptr = 5; printf("%d", *a_ptr);
}

II:  main() {
    int *p, a = 5;
    p = &a; ...
    /* code: a & p NEVER on LHS of = */
    printf("%d", a);
}

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

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>II</th>
<th>III</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>YES</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>5</td>
<td>YES</td>
<td>-</td>
<td>YES</td>
</tr>
<tr>
<td>6</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>7</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>8</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

Linked List Example (#1/7)

° 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;
}

ELEC2041 lec04-C-language-III.pdf
Saeid Nooshabadi
**Linked List Example (#6/7)**

```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 (#7/7)**

```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;
}
```

**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 then 101 11 5 11
2: 104 10 5 10 then 104 11 5 11
3: 101 <other> 5 10 then 101 <3-others>
4: 104 <other> 5 10 then 104 <3-others>
5: One of the two printfs causes an ERROR
6: I surrender!

**References:**

- **Nick Parlante:** Stanford CS Education Library ([http://cslibrary.stanford.edu/](http://cslibrary.stanford.edu/)); A Collection of very useful material including:
  - **Binky Pointer Video** ([http://cslibrary.stanford.edu/104](http://cslibrary.stanford.edu/104)) Silly but memorable 3 minute animated video demonstrating the basic structure, techniques, and pitfalls of using pointers.
  - **Pointer Basics** ([http://cslibrary.stanford.edu/106](http://cslibrary.stanford.edu/106)) The companion text for the Binky video.
  - **Pointers and Memory** ([http://cslibrary.stanford.edu/102](http://cslibrary.stanford.edu/102)) A 31 page explanation of everything you ever wanted to know about pointers and memory.
  - **Linked List Basics** ([http://cslibrary.stanford.edu/103](http://cslibrary.stanford.edu/103)) A 26 page introduction to the techniques and code for building linked lists in C.
### Important Logical Operators

° Logical AND (&&) and bitwise AND (&) operators:

```
char a=4, b=8, c;
c = a && b;
/* After this statement */
c =1*/
c = a & b; /* After this statement */
c =0*/
```

<table>
<thead>
<tr>
<th>Dec</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0000 0100</td>
</tr>
<tr>
<td>b</td>
<td>0000 1000</td>
</tr>
</tbody>
</table>

- Logical shift left (<<) and shift right (>>) operators:

```
char a=4, b=8, c;
c = a << 2;
/* After this statement */
c =16*/
c = b >> 3; /* After this statement */
c =1*/
```

<table>
<thead>
<tr>
<th>Dec</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0000 0100</td>
</tr>
<tr>
<td>b</td>
<td>0000 1000</td>
</tr>
</tbody>
</table>

### What is this stuff good for?

° StressEraser: Biofeedback device promises to reduce stress
  • aid for deep breathing exercises, (prescribed to alleviate stress.)
  • The device tells when to inhale and when to stop.
  • It does this by divining the state of your nervous system by some clever analysis of your heart rate
° device is a pulse oximeter integrated with a display and a microprocessor.

- Pulse oximeters identify heartbeats by the variation in the amount of light absorbed through the skin of your finger as fresh blood pulses through it.
- It monitors heart rate to identify the activity level of the vagus nerve, one of 12 nerves that emanate directly from your brain rather than through your spinal cord.

### World’s Last C Bug

° If you remember nothing else, remember this:

```
while (1) {
    status = GetRadarInfo();
    if (status = 1) {
        LaunchMissiles();
    }
}
```

= is used instead of ==
Example #1:

How many bugs in this code?

```c
#include <stdio.h>
int main ( ) {
    int numAs; /* counts A's in the input */
    char c;
    numAs = 0;
    while ((c = getchar ( )) != EOF) { /* getchar returns EOF if no more chars to read. */
        if (c == 'A') {
            numAs++;
        }
    }
    printf ("%d A's in the input\n", numAs);
    return 0;
}
```

Choices:
1 or none,
2,
3,
4,
5 or more

Example #1 (Solution):

How many bugs in this code?

```c
#include <stdio.h>
int main ( ) {
    int numAs; /* counts A's in the input */
    char c;
    numAs = 0;
    while ((c = getchar ( )) != EOF) { /* getchar returns EOF if no more chars to read. */
        if (c == 'A') {
            numAs++;
        }
    }
    printf ("%d A's in the input\n", numAs);
    return 0;
}
```

3 Bugs

Bug Symptoms

- A value that’s wildly out of range suggests an uninitialized variable or function return value.
- A loop that’s executed 0 times or the maximum number of times when it shouldn’t be suggests misuse of = in a test, or misparentheses.
Example #2

What output is produced by this code?

```c
void addOne (int x) {
    x = x + 1;
}
int main ( ) {
    int y = 3;
    addOne (y);
    printf (%d, y);
    return 0;
}
```

Choices: 3, 4, unknown

Example #2 (Solution)

What output is produced by this code?

```c
void addOne (int x) {
    x = x + 1;
}
int main ( ) {
    int y = 3;
    addOne (y);
    printf (%d, y);
    return 0;
}
```

3 is Produced

Passed by value:
Make a copy of the original argument.
The original won’t change

Example #3

What choice for the blank ensures that 7 = 7 is printed?

```c
#include <stdio.h>
int *f (int varNotToSet, int varToSet) {
    int n = 7;
    varToSet = 7;
    return _____ ;
    /* &n, &varNotToSet, or &varToSet could go here */
}
int main ( ) {
    int *ptr;
    int k1 = 7, k2 = 0;
    ptr = f(k1, k2);
    printf ("7 = ");
    printf (%d, *ptr);
    return 0;
}
```

Choices: 1) &n, 2) &varNotToSet, 3) &varToSet

Example #3 (Solution)

Variables and function parameters are allocated on the system stack

When a function exits, its allocated space gets reallocated to another function.

Answer: none
Example #4

What choice for the blank ensures that values[0] = x

```c
int main() {
    int values[20];
    int x;
    ...
    assign(_____, x);
    ...
}
void assign(int *y, int x) {
    *y = x;
}
```

Answer:
1. values
2. &values[0]

Example #5

How to copy one array to another array

```c
int main() {
    char sFrom[6], sTo[6];
    copy(sTo[6], sFrom[6]);
}
void copy(char sTo[], char sFrom[]) {
    ----
    sTo[0] 'h' --> 'h' sFrom[0]
    sTo[1] 'e' --> 'e' sFrom[1]
    sTo[3] 'l' --> 'l' sFrom[3]
    sTo[5] 0    --> 0 sFrom[5]
}
```

Example #5 (Solution #1/4)

```c
void copy(char sTo[], char sFrom[]) {
    sTo = sFrom;
}
```

Example #5 (Solution #2/4)

Straight Forward Array Version

```c
void copy(char sTo[], char sFrom[]) {
    int k = 0;
    while (sFrom[k] != '\0') {
        sTo[k] = sFrom[k];
        k++;
    }
    /* copy terminating 0 */
    sTo[k] = sFrom[k];
}
```

Similarly you don’t compare two string using ==
Example #5 (Solution #3/4)

° Array Version (Taking advantage of value returned by assignment operator)

```c
void copy (char sTo[], char sFrom[]) {
    int k = 0;
    while ((sTo[k] = sFrom[k]) != '\0') {
        k++;
    }
}
```

° Pointer Version

```c
void copy (char sTo[], char sFrom[]) {
    while (*sTo = *sFrom) {/* pointer arithmetic (K&R 5.4) */
        sFrom++;
        sTo++;
    }
}
```

---

Things to Remember (#1/2)

° Use handles to change pointers
° Create abstractions with structures
° A **struct** is a data structure composed for simpler data types.
° We can change the value of a pointer in a function by pointer to pointer feature.

---

Things to Remember (#2/2)

° Pointers and arrays are virtually same
° C knows how to increment pointers
° C is an efficient language, with little protection
  • Array bounds not checked
  • Variables not automatically initialized
° (Beware) The cost of efficiency is more overhead for the programmer.
  • “C gives you a lot of extra rope but be careful not to hang yourself with it!”