amazon unbox is launched, promising to change to the way movies are delivered. DVD-quality movies, but no Mac support. Sell netflix stock…
### Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

1. Kim’s melodious giddiness terrifies people, excepting zealous yodelers
2. Kirby Messed Gigglypuff Terribly, (then) Perfectly Exterminated Zelda and Yoshi
3. Killed meat gives teeth peace except zebra yogurt
4. Kind Men Give Tense People Extra Zeal (for) Yoga
5. Killing melee gives terror; peace exhibits Zen yoga
6. Killing messengers gives terrible people exactly zero, yo
7. Kindergarten means giving teachers perfect examples (of) zeal (&) youth
8. Kissing mediocre girls/guys teaches people (to) expect zero (from) you
9. Kinky Mean Girls Teach Penis-Extending Zen Yoga

---

1. Young Zebras Exclaim, “People Teach {Giraffes, Girls} Messy Kissing!” – Omar Akkawi
2. “King me,” Gina tells Perry, expert zebra yodeler – Diana Ko
3. Kirk met Gibson’s team, perilously expecting zealous youngsters – Diana Ko
4. Kind Men Give Ten Percent Expressly Zee Yoorphans – Daniel Gallagher
5. King Mel Gibson Tells People “Examine Ze Yoodle!” – Daniel Gallagher
7. Killer Mechanical { Giraffe / Giant } Teaches Pet, Extinct Zebra, to Yodel – Larry Ly
8. Kilted Men Given Testosterone Perform Exceedingly Zealous Yoga – David Wu
Pointer Arithmetic Summary

• \( x = *(p+1) \) ?
  \[ \Rightarrow x = *(p+1) ; \]

• \( x = *p+1 \) ?
  \[ \Rightarrow x = (*p) + 1 ; \]

• \( x = (*p)++ \) ?
  \[ \Rightarrow x = *p ; \quad *p = *p + 1 ; \]

• \( x = *p++ \) ? \( (*p++) \) ? \( *(p)++ \) ? \( *(p++) \) ?
  \[ \Rightarrow x = *p ; \quad p = p + 1 ; \]

• \( x = *++p \) ?
  \[ \Rightarrow p = p + 1 ; \quad x = *p ; \]

• Lesson?

• These cause more problems than they solve!
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 `src` to the memory at `dst`. The caller must ensure that `dst` has enough memory to hold the data to be copied.
• 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 (2/4)

• 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);
```

...review...
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);
```
Pointers (4/4)

- Solution! Pass a pointer to a pointer, 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);
```

\[
\begin{array}{ccc}
50 & 60 & 70 \\
\end{array}
\]
Dynamic Memory Allocation (1/4)

- C has operator `sizeof()` which gives size in bytes (of type or variable)
- Assume size of objects can be misleading and is bad style, so use `sizeof(type)`
  - Many years ago an `int` was 16 bits, and programs were written with this assumption.
  - What is the size of integers now?
- “`sizeof`” is not a real function – it does its work at compile time.
  - So the following IS valid:
    ```c
    char foo[3*sizeof(int)]
    ```
  - But neither of these is valid in C:
    ```c
    char foo[3*myfunction(int)]
    char foo[3*myfunction(7)]
    ```
Dynamic Memory Allocation (2/4)

• To allocate room for something new to point to, use `malloc()` (with the help of a typecast and `sizeof`):

\[
\text{ptr} = (\text{int} *) \text{malloc} (\text{sizeof(int)});
\]

  • 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

\[
\text{ptr} = (\text{int} *) \text{malloc} (n*\text{sizeof(int)});
\]

  • This allocates an array of `n` integers.
Dynamic Memory Allocation (3/4)

• 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.

  • Even though the program frees all memory on `exit` (or when `main` returns), don’t be lazy!

  • You never know when your `main` will get transformed into a subroutine!
Dynamic Memory Allocation (4/4)

• The following two things will cause your program to crash or behave strangely later on, and cause VERY VERY hard to figure out bugs:
  • `free()`ing the same piece of memory twice
  • calling `free()` on something you didn’t get back from `malloc()`

• The runtime **does not** check for these mistakes
  • Memory allocation is so performance-critical that there just isn’t time to do this
  • The usual result is that you corrupt the memory allocator’s internal structure
  • You won’t find out until much later on, in a totally unrelated part of your code!
Binky Pointer Video (thanks to NP @ SU)

Pointer Fun with Binky

by Nick Parlante
This is document 104 in the Stanford CS Education Library — please see cslibrary.stanford.edu for this video, its associated documents, and other free educational materials.

Copyright © 1999 Nick Parlante. See copyright panel for redistribution terms.
Carpe Post Meridiem!
C structures : Overview

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

```c
struct point {   /* type definition */
    int x;
    int y;
};

void PrintPoint(struct point p)
{
    As always in C, the argument is passed by “value” – a copy is made.
    printf("(%d,%d)”, p.x, p.y);
}

struct point p1 = {0,10}; /* x=0, y=10 */
PrintPoint(p1);
```
C structures: Pointers to them

• Usually, more efficient to pass a pointer to the struct.

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

• The following are equivalent:

```
struct point *p;
/* code to assign to pointer */
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`
Linked List Example

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

```c
/* node structure for linked list */
struct Node {
    char *value;
    struct Node *next;
};
```

Recursive definition!
typedef simplifies the code

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

/* "typedef" means define a new type */
typedef struct Node NodeStruct;

... OR ...

typedef struct Node {
    char *value;
    struct Node *next;
} NodeStruct;

... THEN ...

typedef NodeStruct *List;
typedef char *String;
```

```c
/* Note similarity! */
/* To define 2 nodes */
struct Node {
    char *value;
    struct Node *next;
} node1, node2;
```
Linked List Example

/* Add a string to an existing list */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));
    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}

{  
    String s1 = "abc", s2 = "cde";
    List theList = NULL;
    theList = cons(s2, theList);
    theList = cons(s1, theList);
    /* or, just like (cons s1 (cons s2 nil)) */
    theList = cons(s1, cons(s2, NULL));
}
Linked List Example

/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}

node:
  ?

list:

s:
  "abc"

list:
  ... ↳ ...
  ↳ NULL
Linked List Example

/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}

node:

list:

s:

"abc"
Linked List Example

/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));
    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));
    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));
    
    node->value = (String) malloc(strlen(s) + 1);  
    strcpy(node->value, s);
    node->next = list;
    return node;
}

node:

```
+-------------------+
|                  |
|                  |
| "abc"            |
|                  |
+-------------------+-------------------+
|                  |
|                  |
| "abc"            |
|                  |
+-------------------+
```

s:

```
+-------------------+
|                  |
|                  |
| NULL             |
|                  |
+-------------------+-------------------+
```

Garcia, Fall 2006 © UCB
“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.
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>II</th>
<th>III</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>YES</td>
</tr>
<tr>
<td>3</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>5</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>8</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
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 printf causes an ERROR
6: I surrender!