Week Seven: More on memory operations, and structures. Union, function pointer, and bit operations

Zhiyuan Li
Department of Computer Science
Purdue University, USA
Academic integrity

• **Any case of cheating will handled by the Dean of students**

• You are encouraged to discuss problems and approaches but:
  – Sharing solution is not allowed.
  – Buying solutions is not allowed.
  – Copying code from the internet is not allowed.
  – Copying code from other students is not allowed.
  – Copying partial code from other students is not allowed.

• [http://homes.cerias.purdue.edu/~spaf/cpolicy.html](http://homes.cerias.purdue.edu/~spaf/cpolicy.html)

• **Due to the persistence of cheating cases found in previous labs, we are applying a new penalty to all cheating cases for the rest of the semester, starting with Lab 4 and Midterm 1:**

• **A student found involved in a cheating case (concerning labs/projects/exams), regardless whether it is the first offense or not, will be given an "F" grade for the entire course.**
re_alloc(void* p, size_t s)

- Changes the size of the memory block pointed to by p to s bytes
- Contents unchanged for the T bytes, where T = min (old size, new size)
- *Newly alloc’d memory is uninitialized.*
- Unless p==NULL, it must come from malloc, calloc or realloc.
- If p==NULL, equivalent to malloc(size)
- If s==0, equivalent to free(ptr)
- Returns pointer to alloc’d memory, may be different from p, or NULL if the request fails or if s==0
- If fails, original block left untouched, i.e. it is not freed or moved
- **How do we know it failed?**
calloc(size_t n, size_t s)

• Allocates memory for an array of \(n\) elements of \(s\) bytes each and returns a pointer to the allocated memory.
• The memory content is set to zero
• The value returned is a pointer to the allocated memory or \texttt{NULL}

\[
p = (\text{char}*) \text{calloc}(10,1); /*\text{alloc}\ 10\ \text{bytes} */
if(p == \texttt{NULL}) \{ /*\text{panic} */ \}
\]
memcpy(void *dest, const void *src, size_t n)

• In GNU C library, declared in string.h
• Copies n bytes from src to dest
• Returns dest
• Does not check for overflow on copy

/* #include <stdlib.h> */
#include <stdio.h>
#include <string.h>
main () {
    char buf[100], *newbuf;
    char const *src = "Hi there!";
    memcpy(buf, src, 10); /* copy 10 chars */
    printf("buf is \t%s\n", buf);
}
Another example (memcpy2.c)

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

main () {
    char buf[100], *newbuf;
    int type = 'a';
    char retype = 'a';
    char const src[20] = "Hi there!";
    memcpy(buf, &type, 4); /*copy an integer */
    memcpy(buf+4, src, 10); /*copy 10 chars */
    printf("buf is \t%s\n", buf);
    memcpy(buf, &retype, 1); /*copy an integer */
    memcpy(buf+4, src, 10); /*copy 10 chars */
    printf("buf is \t%s\n", buf); /* display is different from above*/
}
```

Draw a picture to show padding
memset(void *s, int c, size_t n)

• Sets the first \( n \) bytes in \( s \) to the value of \( c \)
  – (\( c \) is converted to an \texttt{unsigned char})

• Returns \( s \)

• Does not check for overflow

• \texttt{memset(mess, 0, 100);}
Use malloc to allocate an array to store several strings of variable lengths

```c
#include <stdlib.h>                  /* StringArray.c */
#include <stdio.h>                   /* Similar to hash tables */
#include <string.h> /* in Project 1 */

main () {

    char *buf[4];
    char const *src1 = "Hi there!";
    char const *src2 = "Hi!";
    char const *src3 = "Howdy!";
    char const *src4 = "Holla!";

    buf[0] = (char *) malloc(strlen(src1));
    memcpy(buf[0], src1, strlen(src1));
    printf("buf0 is \t%s\n", buf[0]);
    buf[3] = (char *) malloc(strlen(src3));
    memcpy(buf[3], src3, strlen(src3));
    printf("buf3 is \t%s\n", buf[3]);
}
```

Draw a picture
For hash table
• We now discuss more complex issues of using structures
Typedef

- Allows us to create new data name types;
  - `typedef int len;`
  - `len l1, l2;`
  - `typedef struct { len x, y;} pos;`
  - `pos p1, p2;`

- Notice the difference. No struct needed when using the type.

- Use `typedef` to define pointer types and function types
• A structure can contain a member of another structure

    struct pos { int x; int y; };
    struct slot {
      struct pos p;
      char c;
    } s;

• Access x via: s.p.x

• The size of slot is exactly the same as if the fields of pos were written inline in slot

• In terms of memory consumption and access speed, there is no cost to nested structures

• Let us look at the next program
#include <stdio.h> /* structstruct.c */

struct pos { int x; int y; };
struct slot {
    struct pos p;
    char c;
} myslot;

main() {
    struct slot localslot;

    printf("sizeof struct slot \t%d\n", (int) sizeof(struct slot));
    printf("sizeof struct pos \t%d\n", (int) sizeof(struct pos));
    printf("sizeof local slot \t%d\n", (int) sizeof(localslot));

    printf("address of myslot \t%p\n", &myslot);
    printf("address of myslot.p \t%p\n", &myslot.p);
    printf("address of myslot.p.x \t%p\n", &myslot.p.x);
    printf("address of myslot.p.y \t%p\n", &myslot.p.y);
    printf("address of myslot.c \t%p\n", &myslot.c);
}
Structures and functions

• Structures can be initialized, copied as any other value

• **They can not be compared directly**
  – instead one must write code to compare members one by one
  – Or compare the addresses of the structures, to see whether the same structure (in the same memory location) has two aliases.

• Functions can return structure instances
  – What is the cost in terms of memory allocation, copy, and performance?
    • See the next code example
  – What’s the difference between arrays and structures in this sense?

```c
struct pt { int x, y; }

struct pt mkpt(int x, int y) {
    struct pt t; t.x = x; t.y = y; return t;
}

struct pt p1 = mkpt(0, 0);
```
Compare the locations of two structures (before and after returning)

```c
#include <stdio.h>               /* returnstruct.c */

struct pt { int x, y; };  
struct pt mkpt(int x, int y) {  
    struct pt t; t.x = x; t.y = y;
    printf("Inside mkpt\n");
    printf("address of t.x\t%p\n", &t.x);
    printf("address of t.y\t%p\n", &t.y);
    printf("exiting mkpt\n");
    return t;
}

main (){  
    struct pt p1 = mkpt(0, 0);
    printf("address of p1.x\t%p\n", &p1.x);
    printf("address of p1.y\t%p\n", &p1.y);
} 
```
Structures and functions

• What happens when a structures is passed as an argument

```c
struct Fs { int a; };
typedef struct Fs F;
F doIt(F b) {
    b.a = 13; return b;
}

int main () {
    F f = { 100 }; f = doIt(f);
    printf("%d\n", f.a);
}
```
Examine the addresses

```c
#include <stdio.h> /* passstruct.c */

struct Fs { int a; };
typedef struct Fs F;
F doIt(F b) {
    /* note the difference from using struct tag */
    printf("address of structure b\t%p\n", &b);
    b.a = 13; return b;
}

int main () {
    F f = { 100 };
    printf("address of structure f\t%p\n", &f);
    printf("f.a before calling doIt \t%d\n", f.a);
    f = doIt(f);
    printf("f.a after calling doIt \t%d\n", f.a);
    printf("address of f \t%p\n", &f);
}
```
Copying versus passing pointers

• We see that a lot of copying is involved in directly passing structures as parameters and returning structures
  – This is potentially quite expensive for large structures or frequent function calls

• Alternatively one can pass and return point to a structure
  – To save the copying cost
• Alternatively

```c
struct Fs { int a; };

typedef struct Fs F

void doIt(F* b) { b->a = 13; }

int main () {
    F f = { 100 }; doIt(&f);
    printf("%d\n", f.a);
}
```
```c
#include <stdio.h> /* passpointer.c */

struct Fs { int a; };
typedef struct Fs F;
void doIt(F* b) {
    printf("value of pointer b\t%p\n", b);
    b->a = 13;
}

int main () {
    F f = { 100 };
    printf("address of structure f\t%p\n", &f);
    printf("f.a before calling doIt \t%d\n", f.a);
    doIt(&f);
    printf("f.a after calling doIt \t%d\n", f.a);
}
```
Dangling pointer dereference

• One must be careful with *dangling pointer* dereference after a function returns
  – If a pointer points to a local variable of a function
  – After that function returns, the variable may be overwritten by new function calls
See how dead local variables may be overwritten

```c
#include <stdio.h> /* dangling.c */

struct pt { int x, y; };

struct pt * mkpt(int x, int y) {
    struct pt t; t.x = x; t.y = y;
    printf("Inside mkpt\n");
    printf("address of t.x\t%p\n", &t.x);
    printf("address of t.y\t%p\n", &t.y);
    printf("exiting mkpt\n");
    return &t;
}

int fibo(int a){
    if (a < 2 ) return 1;
    return fibo(a-1)+fibo(a-2);
}

main (){  
    struct pt *p1 = mkpt(0, 0);
    fibo(10);
    printf("p1->x\t%d\n", p1->x);
    printf("p1->y\t%d\n", p1->y);
}
```
Memory padding for structures

• Data alignment: when the processor accesses the memory reads more than one byte, usually 4 bytes on a 32-bit platform

• What if the data structure is not a multiple of 4?
  – Padding: some unused bytes are inserted in the structure by the compiler
  – Note: memory allocation also needs word alignment
  – Let us print out the memory addresses of struct members