

C “Review”

Part 2

November 4, 2010

Warmup Example

```
#include <assert.h>
#include <stdio.h>
int main() {
    int x = 0x40;    /* Hexadecimal value 40 */
    char y = 'A';   /* 'A' ASCII Code is 65 */
    char *z;
    z = (char*)malloc(20*sizeof(char));
    assert(z);      /* what does assert do? */
    printf("%d\n", x+y);
    printf("%d\n", y);
    printf("%c\n", y);
    strncpy(z, "Kaiser Sohse\0");
    printf("%s\n", z);
}
```

What gets printed here?

How is assert different from using an if statement to check for z's validity?

enum & typedef

- `enum days {MO, TU, WE, TH, FR, SA, SU} ;`
- `enum days lab;`
- `lab = TH;`
- enums start with 0 and increment by 1 (unless specified)
- `TU = 1, WE = 2, etc.`

- `typedef` - creates a shorthand name for a type.
- `typedef <type> <name>;`
- `typedef struct Fraction_T { . . . } Fraction_T; Note what this declaration does`
- `typedef unsigned int UINT32;`
- `typedef enum {FALSE, TRUE} bool;`

Structures

```
struct fraction_T {  
    int numerator;  
    int denominator;  
};
```

. (dot) is the access operator for accessing members of a structure

```
int main() {  
    struct fraction_T orig_fraction, dupe;  
    orig_fraction.denominator = 20;  
    orig_fraction.numerator = 10;  
    dupe = orig_fraction;  
    printf("%d / %d\n", dupe.numerator,  
           dupe.denominator);  
}
```

What will this print?

Structures and Pointers

```
typedef struct fract_T {  
    int numerator;  
    int denominator;  
} fract_T;
```

```
int main() {  
    fract_T *orig, *dupe;  
    orig=(fract_T*)malloc(sizeof(fract_T));  
    if (!orig) exit(0); /* fail silently-bad? */  
    orig->numerator = 10; What does -> do?  
    (*orig).denominator = 20;  
    dupe = orig;  
    printf("%d / %d\n", dupe->numerator,  
        (*dupe).denominator);  
}
```

Typedef can be used to save on some typing when dealing with structs, but some programmers feel typedef's decrease code readability

What is dupe?

Structures and Pointers (summary)

```
typedef struct fract_T {  
    int numerator;  
    int denominator;  
} orig, *origptr;
```

```
orig x;  
origptr y = &x;  
x.numerator = 10; /* this line is same as next*/  
y->numerator = 10; /* redundant */
```

I do not like to use typedef to hide a pointer. It is very common to use typedef to obscure the fact that you are using a struct.

What does including orig and origptr after closing french bracket instruct compiler to do?

Struct Comparisons ???

```
typedef struct fract_T {  
    int numerator;  
    int denominator;  
} fract_T ;
```

```
int main() {  
    fract_T orig, dupe;  
    orig.numerator = 10;  
    orig.denominator = 20;  
    dupe.numerator = 10;  
    dupe.denominator = 20;  
    if (orig == dupe)  
        printf ("Lucky! \n");  
}
```

What is compared in the if stmt?
Will this compile?

Exercise 1: Passing struct by Value

```

struct fract_T {
    int numerator;
    int denominator;
};
void set_number(struct fract_T myFract) {
    printf("1: %d and %d\n", myFract.numerator,
        myFract.denominator);
    myFract.numerator = 20;
    myFract.denominator = 15;
    printf("2: %d and %d\n", myFract.numerator,
        myFract.denominator);
}
int main() {
    struct fract_T demo;
    demo.numerator = 10;
    demo.denominator = 20;
    set_number(demo);
    printf("3: %d and %d\n", demo.numerator,
        demo.denominator);
}

```

Passing struct by reference

Change the parameter in `set_number` to be a pointer, and pass `&demo` instead of `demo`.

...

```
void set_number(fract_T *myFract) {  
    ...  
    myFract->numerator = 20;  
    myFract->denominator = 15;  
    ...  
}
```

...

```
set_number(&demo);
```

...

Structs: parameter passing

- Pass by value (as a parameter)
 - Copy of *struct* made for a function called
- Pass by reference (as a `struct*` (pointer))
 - Copy of *memory address* made
- When an array passed as parameter:
 - is a copy of entire array made for the function?
- `Fract_T our_fractions[100];`
- `Fract_T* our_fractions[100];`
- **What do these declarations do?**

Function Declaration Ordering

- As compiler runs through a program top to bottom, it expects to see a function declared before it is called
- Unfortunately, we might want to call a function that is below our current function in the program (Catch 22)
- To solve this problem, C allows the use of forward declarations (a.k.a. function prototyping when used on a function)

Contrived Example

```
void ping(int *a) {
    (*a) *= 2;
    printf("ping!\n");
    pong(a);
}
void pong(int *a) {
    (*a)--;
    printf("pong!\n");
    if (*a > 32) return;
    ping(a);
}

int main(int argc, char* argv[]) {
    if (argc > 1)
        ping( atoi(argv[1]) );
    else
        ping(1);
}
```

Function Catch 22 Resolution

- The “function prototype” lets the compiler know the full function is coming later in the program:

```
void pong (int *);  
void ping(int *a) {  
    (*a) *=2;  
    printf("ping!\n");  
    pong(a);  
}  
void pong(int *a) {  
    ...  
}
```

Header Files (.h)

For a given source code file (ex: `foo.c`) there typically is a header file (`foo.h`) with all the prototypes for the functions. This provides an interface showing the `foo` function prototypes without revealing the source code

```
//contents of the foo.h file  
void ping (int *a);  
void pong(int *);
```

Note: parameters of a function prototype do not need variable names. They are optional, and help with readability.

In `foo.c` you'll want to "include" the function prototypes from `foo.h` so that you don't have to prototype again.

```
//near the beginning of foo.c file  
#include "foo.h"
```

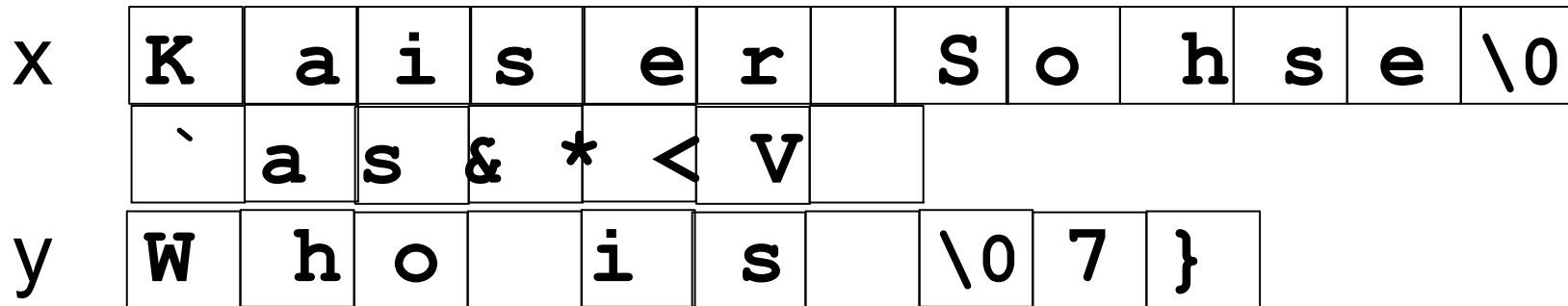
strcat and your “first” buffer overflow

```
#include <stdio.h>
int main () {
    char x[20] = "Kaiser Sohse\0";
    char y[10] = "Who is \0";
    strcat(y, x);
    printf ("%s\n", y);
}
```

- Does this compile?
- Does it run?
- Are you doing something wrong?

strcat

The Stack



1. `strcat` will find the end of string `y`

how does it do this?

2. `strcat` will start at the end of string `y` and copy characters of `x` in to `y`

when will it stop?

how does it know when to stop?

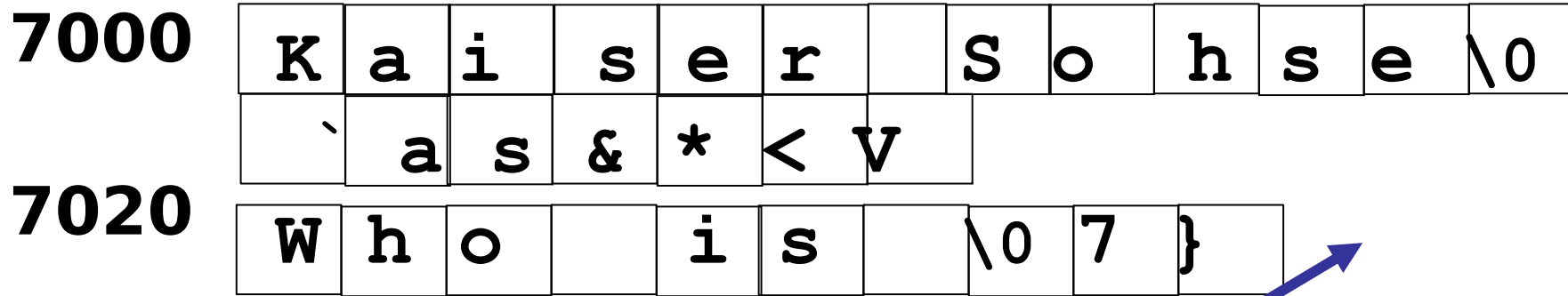
strcat heap example ...

```
int main () {  
    char *x;  
    char *y;  
    x=(char*)malloc(20*sizeof(char));  
    assert(x);  
    y=(char*)malloc(10*sizeof(char));  
    if(!y) exit(0);  
    strcpy(x, "Kaiser Sohse\0");  
    strcpy(y, "Who is \0");  
    strcat(y, x);  
    printf ("%s\n", y);  
}
```

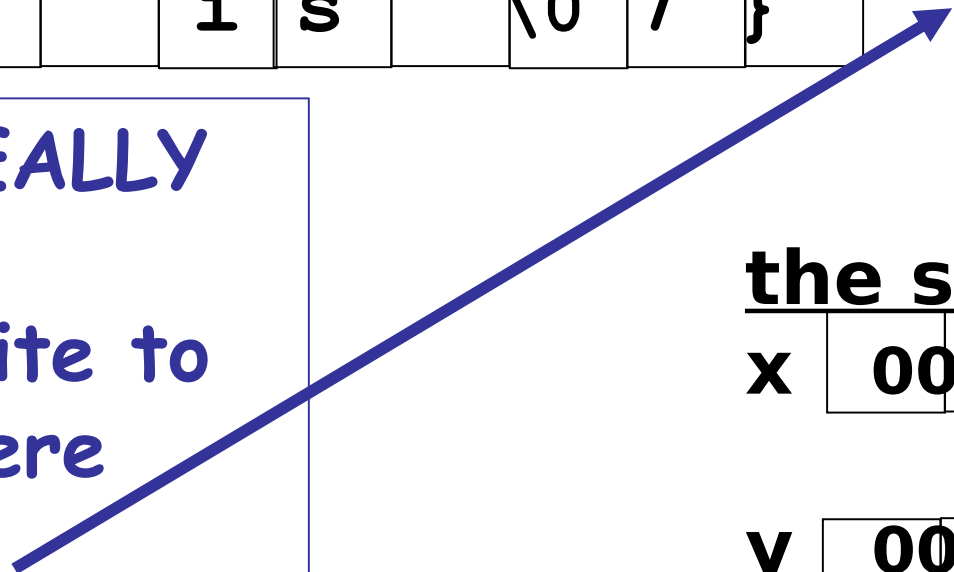
What is different from the last example?

strcat heap

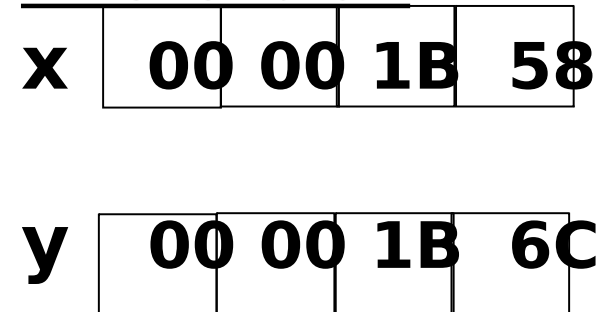
the heap



It still is a REALLY BAD IDEA to attempt to write to the memory here



the stack



strcat Solution

- **Get enough heap space ahead of time!**

```
int main () {
    char *x;
    char *y;
    x=(char*)malloc(20*sizeof(char));
    if(!x) exit(0);
    y=(char*)malloc(10*sizeof(char));
    if(!y) exit(0);
    strcpy(x, "Kaiser Sohse\0");
    strcpy(y, "Who is \0");
    y=realloc(y, strlen(x)+strlen(y)+1);
    if(!y) exit(0);
    strcat(y, x);
    printf ("%s\n", y);
}
```

Unsafe functions

- Using strcat (and most other classical C string handling routines) is outdated and typically unsafe. When available, use counted versions of the functions (strncat, strncpy) to get **some** safety, especially when dealing with user input.
- Read the specifications carefully to avoid misuse of these 'less unsafe' functions. For example, the count in strncat is the amount of remaining space left in the destination buffer, not the size of the destination buffer. If the size of the destination buffer is used as the count parameter, a buffer overflow is possible.

Exercise: C-string queue (FIFO)

- Implement the following functions, as well as define the struct(s) necessary

```
/* Add item to end of Q */
```

```
void enqueue(queue* Q,  
             char* item);
```

```
/* Get head of the queue */
```

```
char* dequeue(queue* Q);
```

```
/* Flushes (clears) the Q */
```

```
void flush_queue(queue* Q);
```

Put struct and prototypes in header file “queue.h” and function bodies in “queue.c”

Make a test file “test.c” that tests your queue implementation

How you do this is up to you

Your test file should try various tests of your enqueue, dequeue, and flush_queue functions

Test with empty queue

Test different combinations of adding, removing, and flushing