

CS 33

Introduction to C Part 6

Copying Strings (1)

```
char s1[] = "abcd";
char s2[5];

s2 = s1;    // does this do anything useful?

// correct code for copying a string
for (i=0; s1[i] != '\0'; i++)
    s2[i] = s1[i];
s2[i] = '\0';

// would it work if s2 were declared:
char *s2;
// ?
```

The answer to the first question is no: the assignment will be considered a syntax error, since the value of `s2` is constant. What we really want to do is copy the array pointed to by `s1` into the array pointed to by `s2`, which we do explicitly in the following lines of code. Note that, while we could have copied the null byte within the for loop, we copy it explicitly outside the for loop. This is to emphasize that the null must be copied.

Our code would not work if `s2` were declared simply as a pointer. The original `s2`, declared as an array, has 5 bytes of memory associated with it, which is sufficient space to hold the string that's being copied. Thus the original `s2` points to an area of memory suitable for holding a copy of the string. The second `s2`, being declared as simply a pointer and not given an initial value, points to an unknown location in memory. Copying the string into what `s2` points to will probably lead to disaster (otherwise known as a segmentation fault).

Copying Strings (2)

```
char s1[] = "abcdefghijklmnopqrstuvwxyz";  
char s2[5];
```

```
for (i=0; s1[i] != '\0'; i++)  
    s2[i] = s1[i];  
s2[i] = '\0';
```

} Does this work?

```
for (i=0; (i<4) && (s1[i] != '\0'); i++)  
    s2[i] = s1[i];  
s2[i] = '\0';
```

} Works!

The answer, of course, is that the first for loop doesn't work, since there's not enough room in the array referred to by s2 to hold the contents of the array referred to by s1. Note that "&&" is the AND operator in C.

The correct way to copy a string is shown in the code beginning with the second for loop, which checks the length of the target: it copies no more than 4 bytes plus a null byte into s2, whose size is 5 bytes.

String Length

```
char *s1;

s1 = produce_a_string();
// how long is the string?

sizeof(s1); // doesn't yield the length!!

for (i=0; s1[i] != '\0'; i++)
    ;
// number of non-null characters in s1 is now in i
```

`sizeof(s1)` yields the size of the variable `s1`, which, on a 64-bit architecture, is 8 bytes.

Size

```
int main() {  
    char s[] = "1234";  
    printf("%d\n", sizeof(s));  
    proc(s, 5);  
    return 0;  
}
```

```
void proc(char s1[], int len) {  
    char s2[12];  
    printf("%d\n", sizeof(s1));  
    printf("%d\n", sizeof(s2));  
}
```

```
$ gcc -o size size.c  
$ ./size  
5  
8  
12  
$
```

`sizeof(s)` is 5 because 5 bytes of storage were allocated to hold its value (including the null).

`sizeof(s1)` is 8 because it's a pointer to a char, and pointers occupy 8 bytes.

`sizeof(s2)` is 12 because 12 bytes of storage were allocated for it.

Not a Quiz

```
void proc(char s[16]) {  
    printf("%d\n", sizeof(s));  
}
```

What's printed?

- a) 8
- b) 15
- c) 16
- d) 17

Comparing Strings (1)

```
char *s1;
char *s2;

s1 = produce_a_string();
s2 = produce_another_string();
// how can we tell if the strings are the same?

if (s1 == s2) {
    // does this mean the strings are the same?
} else {
    // does this mean the strings are different?
}
```

Note that comparing `s1` and `s2` simply compares their numeric values as pointers, it doesn't take into account what they point to.

Comparing Strings (2)

```
int strcmp(char *s1, char *s2) {
    int i;
    for (i=0;
        (s1[i] == s2[i]) && (s1[i] != 0) && (s2[i] != 0);
        i++)
        ; // an empty statement
    if (s1[i] == 0) {
        if (s2[i] == 0) return 0; // strings are identical
        else return -1; // s1 < s2
    } else if (s2[i] == 0) return 1; // s1 > s2
    if (s1[i] < s2[i]) return -1; // s1 < s2
    else return 1; // s2 < s1;
}
```

The for loop finds the first position at which the two strings differ. The rest of the code then determines whether the two strings are identical (if so, they must be of the same length), and if not, it determines which is less than the other. The function returns -1 if s1 precedes s2, 0 if they are identical, and 1 if s2 precedes s1.

The comparisons are done using the ordering implied by the ASCII encoding of strings. From the previous lecture, we know that the encoding of 'a' is 97 (in decimal) and the encoding of 'b' is 98. Thus 'a' < 'b'. This sort of results in a lexicographical (dictionary) ordering of strings, but it has the peculiar property that all uppercase letters are less than all lowercase letters. Thus, while "abc" is less than (comes before) "bcd", "abc" is greater than (goes after) "Bbc".

The String Library

```
#include <string.h>

char *strcpy(char *dest, char *src);
    // copy src to dest, returns ptr to dest
char *strncpy(char *dest, char *src, int n);
    // copy at most n bytes from src to dest
int strlen(char *s);
    // return the length of s (not counting the null)
int strcmp(char *s1, char *s2);
    // returns -1, 0, or 1 depending on whether s1 is
    // less than, the same as, or greater than s2
int strncmp(char *s1, char *s2, int n);
    // do the same, but for at most n bytes
```

The String Library (more)

```
size_t strspn(const char *s, const char *accept);  
    // returns length of initial portion of s  
    // consisting entirely of bytes from accept  
  
size_t strcspn(const char *s, const char *reject);  
    // returns length of initial portion of s  
    // consisting entirely of bytes not from  
    // reject
```

These will be useful in upcoming assignments.

Quiz 1

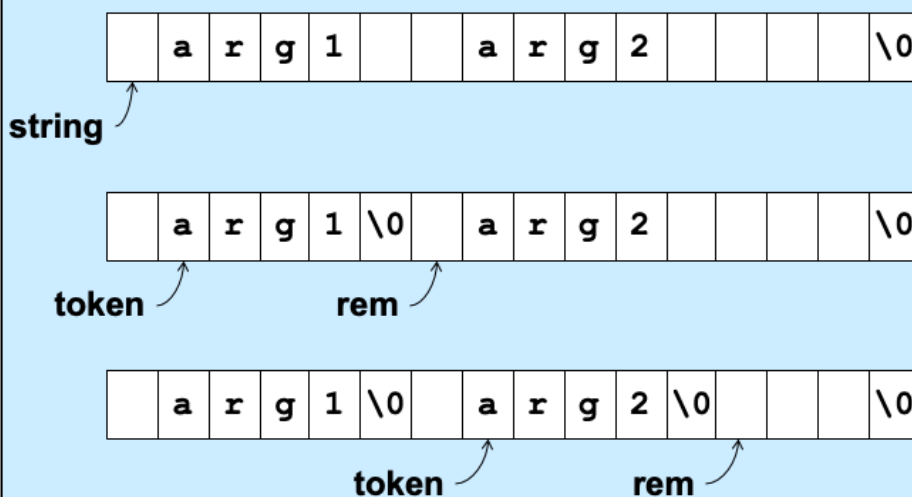
```
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "Hello World!\n";
    char *s2;
    strcpy(s2, s1);
    printf("%s", s2);
    return 0;
}
```

This code:

- a) is a great example of well written C code**
- b) has syntax problems**
- c) might seg fault**

Parsing a String



Suppose we have a string of characters (perhaps typed into the command line of a shell). We'd like to parse this string to pull out individual words or "tokens" (to be used as arguments to a command); these tokens are separated by one or more characters of white space. Starting with a pointer to this string, we call a function that null-terminates the first token and returns a pointer to that word (*token*) and sets *rem* to point to the remainder of the string. We call it again to get the second token, etc.

Designing the Parse Function

- It modifies the string being parsed
 - puts nulls at the end of each token
- Each call returns a pointer to the next token
 - how does it know where it left off the last time?
 - » how is *rem* dealt with?

The parse function must keep track of where it left off after each call to it. One way of doing this is via the use of a static local variable.

Design of *strtok*

- **char *strtok(char *string,
 const char *sep)**
 - if *string* is non-NULL, *strtok* returns a pointer to the first token in *string*
 - if *string* is NULL, *strtok* returns a pointer to the next token in *string* (the one after that returned in the previous call), or NULL if there are no more tokens
 - tokens are separated by any non-empty combination of characters in *sep*

strtok is a standard function in the C strings library. Note that, since the second argument is declared to be a pointer to a constant, there's a promise that *strtok* will not modify what its second argument points to.

Using *strtok*

```
int main() {  
    char line[] = "  arg0  arg1 arg2  arg3  ";  
    char *str = line;  
    char *token;  
    while ((token = strtok(str, " \\t\\n")) != NULL) {  
        printf("%s\\n", token);  
        str = NULL;  
    }  
    return 0;  
}
```

Output:
arg0
arg1
arg2
arg3

***strtok* Code** part 1

```
char *strtok(char *string, const char *sep) {
    static char *rem = NULL;
    if (string == NULL) {
        if (rem == NULL) return NULL;
        string = rem;
    }
    int len = strlen(string);
    int slen = strspn(str, sep);
        // initial separators
    if (slen == len) {
        // string is all separators
        rem = NULL;
        return NULL;
    }
}
```


***strtok* Code** part 2

```
string = &string[slen]; // skip over separators
len -= slen;
int tlen = strcspn(string, sep); // length of first token
if (tlen < len) {
    // token ends before end of string: terminate it with 0
    string[tlen] = '\0';
    rem = &string[tlen+1];
} else {
    // there's nothing after this token
    rem = NULL;
}
return string;
}
```

Numeric Conversions

```
short a;  
int b;  
float c;  
  
b = a;    /* always works */  
a = b;    /* sometimes works */  
c = b;    /* sort of works */  
b = c;    /* sometimes works */
```

Assigning a short to an int will always work, since all possible values of a short can be represented by an int. The reverse doesn't always work, since there are many more values an int can take on than can be represented by a short.

A float can represent an int in the sense that the smallest and largest ints fall well within the range of the smallest (most negative) and largest floats. However, floats have few significant digits than do ints and thus, when converting from an int to a float, there may well be a loss of precision.

When converting from a float to an int there will not be any loss of precision, but large floats and small (most negative) floats cannot be represented by ints.

Implicit Conversions (1)

```
float x, y=2.0;
int i=1, j=2;

x = i/j + y;
/* what's the value of x? */
```

x's value will be 2, since the result of the (integer) division of i by j will be 0.

Implicit Conversions (2)

```
float x, y=2.0;
int i=1, j=2;
float a, b;

a = i;
b = j;
x = a/b + y;
/* now what's the value of x? */
```

Here the values of *i* and *j* are converted to float before being assigned to *a* and *b*, thus the value assigned to *x* is 2.5.

Explicit Conversions: Casts

```
float x, y=2.0;
int i=1, j=2;

x = (float)i/(float)j + y;
/* and now what's the value of x? */
```

Here we do the int-to-float conversion explicitly; x's value will be 2.5.

Purposes of Casts

- **Coercion**

```
int i, j;  
float a;  
a = (float)i / (float)j;
```

modify the
value
appropriately

- **Intimidation**

```
float x, y;  
    // sizeof(float) == 4  
swap((int *)&x, (int *)&y);
```

it's ok as is
(trust me!)

“Coercion” is a commonly accepted term for one use of casts. “Intimidation” is not. The concept is more commonly known as a “sidecast”. Coercion means to convert something of one datatype to another. Intimidation (or sidecasting) means to treat an instance one datatype as being another datatype without doing any conversion of the actual data. Intimidation works only for pointer datatypes.

Quiz 2

- Will this work?

```
double x, y; //sizeof(double) == 8
```

```
...
```

```
swap((int *) &x, (int *) &y);
```

a) yes

b) no

Caveat Emptor

- **Casts tell the C compiler:**
“Shut up, I know what I’m doing!”

- **Sometimes true**

```
float x, y;  
swap((int *) &x, (int *) &y);
```

- **Sometimes false**

```
double x, y;  
swap((int *) &x, (int *) &y);
```

The call to *swap* makes sense as long as what *x* and *y* point to are the same size as *int*'s.

The moral is to be careful with casting, particularly intimidation casts, since they effectively turn off type checking.

Nothing, and More ...

- ***void* means, literally, nothing:**

```
void NotMuch(void) {  
    printf("I return nothing\n");  
}
```

- **What does *void ** mean?**
 - it's a pointer to anything you feel like
 - » a generic pointer

The `void *` type is an exception to the rule that the type of the target of a pointer must be known.

Rules

- **Use with other pointers**

```
int *x;  
void *y;  
x = y; /* legal */  
y = x; /* legal */
```

- **Dereferencing**

```
void *z;  
func(*z); /* illegal! */  
func(*(int *)z); /* legal */
```

Dereferencing a pointer must result in a value with a useful type. “void” is not a useful type.

Swap, Revisited

```
void swap(int *i, int *j) {  
    int tmp;  
    tmp = *j; *j = *i; *i = tmp;  
}  
/* can we make this generic? */
```

Can we write a version of *swap* that handles a variety of data types?

An Application: Generic Swap

```
void gswap (void *p1, void *p2,
           int size) {
    int i;
    for (i=0; i < size; i++) {
        char tmp;
        tmp = ((char *)p1)[i];
        ((char *)p1)[i] = ((char *)p2)[i];
        ((char *)p2)[i] = tmp;
    }
}
```

Note that there is a function in the C library that one may use to copy arbitrary amounts of data — it's called *memcpy*. To see its documentation, use the Linux command “man memcpy”.

Using Generic Swap

```
short a=1, b=2;  
gswap(&a, &b, sizeof(short));
```

```
int x=6, y=7;  
gswap(&x, &y, sizeof(int));
```

```
int A[] = {1, 2, 3}, B[] = {7, 8, 9};  
gswap(A, B, sizeof(A));
```

Fun with Functions (1)

```
void ArrayDouble(int A[], int len) {  
    int i;  
    for (i=0; i<len; i++)  
        A[i] = 2*A[i];  
}
```

Fun with Functions (2)

```
void ArrayBop(int A[],
             int len,
             int (*func)(int)) {
    int i;
    for (i=0; i<len; i++)
        A[i] = (*func)(A[i]);
}
```

Here *func* is declared to be a pointer to a function that takes an *int* as an argument and returns an *int*.

What's the difference between a pointer to a function and a function? A pointer to a function is, of course, the address of the function. The function itself is the code comprising the function. Thus, strictly speaking, if *func* is the name assigned to a function, *func* really represents the address of the function. You might think that we should invoke the function by saying “**func*”, but it's understood that this is what we mean when we say “*func*”. Thus when one calls *ArrayBop*, one supplies the name of the desired function as the third argument, without prepending “&”.

Fun with Functions (3)

```
int triple(int arg) {  
    return 3*arg;  
}  
  
int main() {  
    int A[20];  
    ... /* initialize A */  
    ArrayBop(A, 20, triple);  
    return 0;  
}
```

Here we define another function that takes a single *int* and returns an *int*, and pass it to *ArrayBop*.

Laziness ...

- Why type the declaration

```
void * (*f) (void *, void *);
```

- You could, instead, type

```
MyType f;
```

- (If, of course, you can somehow define *MyType* to mean the right thing)

typedef

- **Allows one to create new names for existing types**

```
typedef int *IntP_t;
```

```
IntP_t x;
```

– means the same as

```
int *x;
```

More typedefs

```
typedef struct complex {  
    float real;  
    float imag;  
} complex_t;  
  
complex_t i, *ip;
```

A standard convention for C is that names of datatypes end with “_t”. Note that it’s not necessary to give the struct a name in this example (we could have omitted the “complex” following “struct”). It’s also not necessary for the name of the type to be different from the name of the struct. Though it’s a bit confusing, we could have coded the above as:

```
typedef struct complex {  
    float real;  
    float imag;  
} complex;
```

```
complex i, *ip;
```

After doing this, “struct complex” and “complex” would mean exactly the same thing.

And ...

```
typedef void *(*MyFunc_t)(void *, void *);

MyFunc_t f;

// you must do its definition the long way

void *f(void *a1, void *a2) {
    ...
}
```

Quiz 3

- What's A?

```
typedef double X_t[M];  
X_t A[N];
```

- a) an array of N doubles
- b) an MxN array of doubles
- c) an NxM array of doubles
- d) a syntax error