CS 33

Introduction to C
Part 7

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);
  // returns 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);
      // return length of initial portion of s
      // consisting entirely of bytes from accept

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

Quiz 1

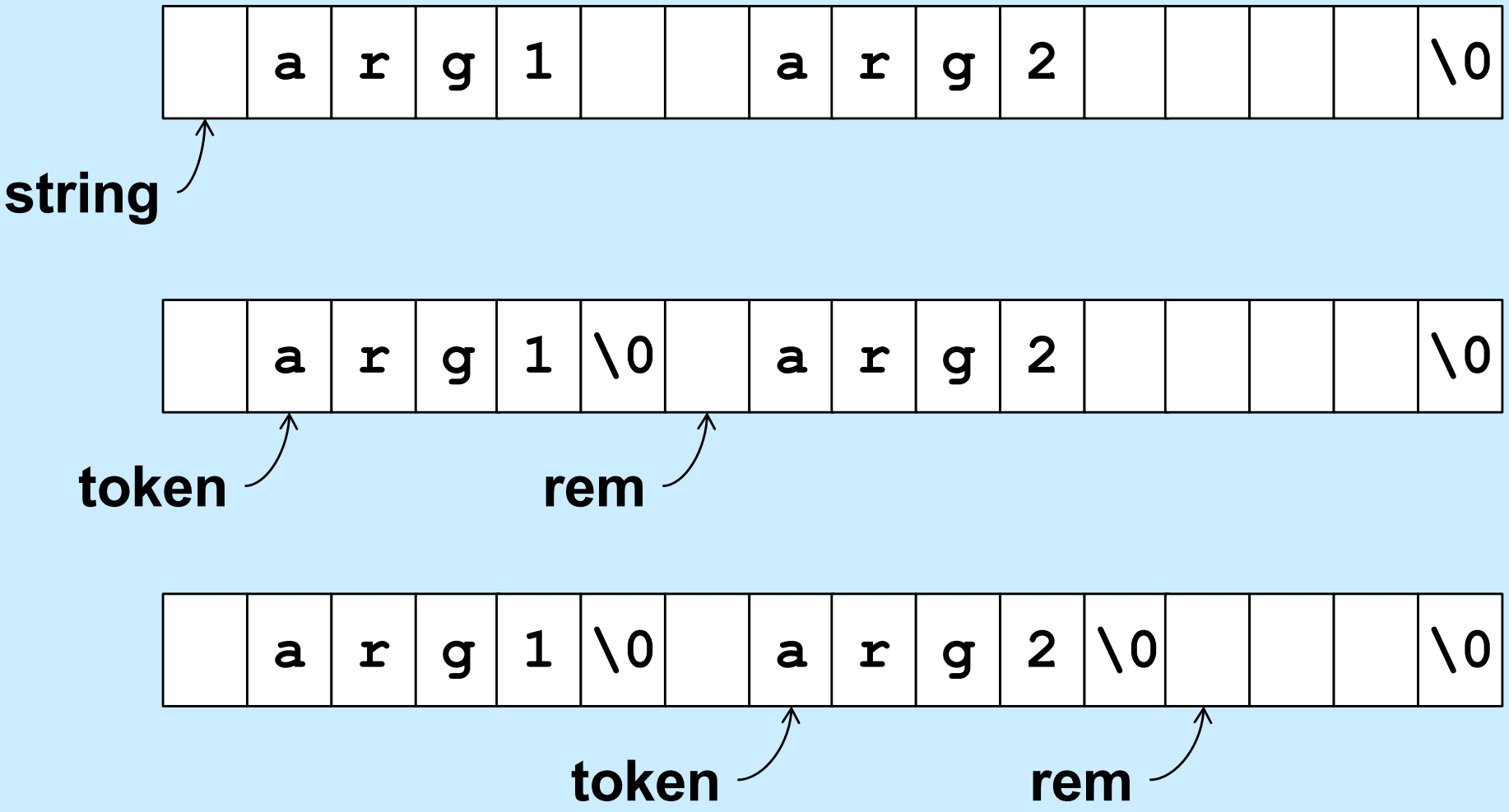
```
#include <stdio.h>
#include <string.h>

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

This code:

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

Parsing a String



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?

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* (and keeps track of where the next token would be)**
  - **if *string* is NULL, *strtok* returns a pointer to the token just after the one 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***

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(string, 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 */
```

Implicit Conversions (1)

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

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

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? */
```

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? */
```

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!)

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);
```

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**

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 */
```

Swap, Revisited

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

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;
  }
}
```

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]);
}
```

Fun with Functions (3)

```
int triple(int arg) {
  return 3*arg;
}

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

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;
```

Not a Quiz

- **What's A?**

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

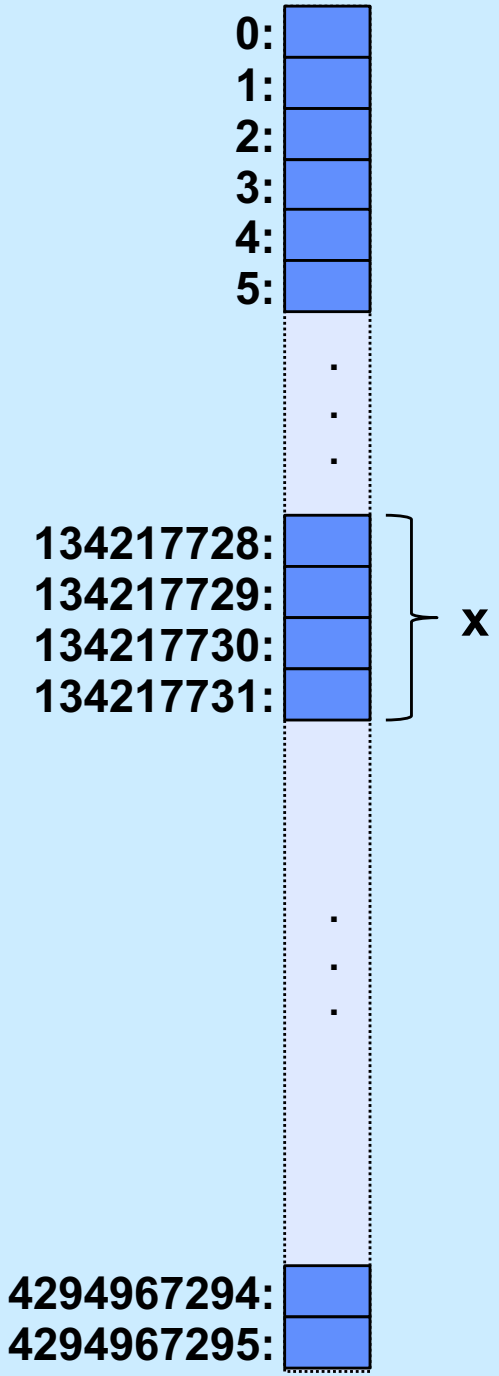
a) an array of M doubles
b) an MxN array of doubles
c) an NxM array of doubles
d) a syntax error

CS 33

Data Representation, Part 1

Representing Data in Memory

- x is a 4-byte integer
  - how do the 32 bits represent its value?

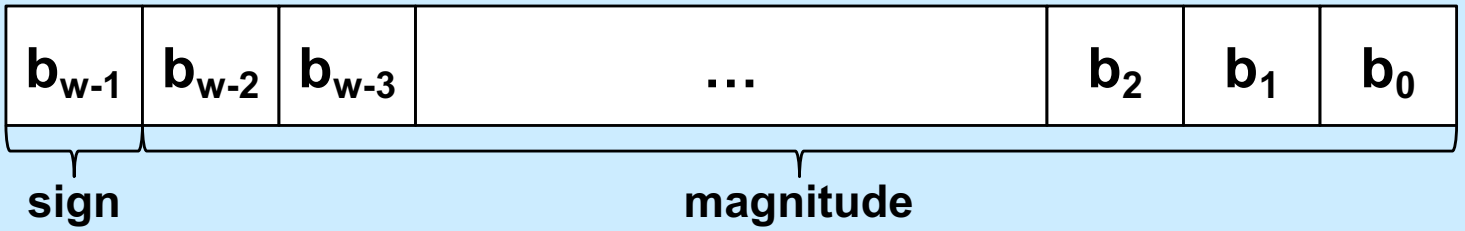


Unsigned Integers

$$\text{value} = \sum_{i=0}^{w-1} b_i \cdot 2^i$$

Signed Integers

- Sign-magnitude



$$\text{value} = (-1)^{b_{w-1}} \cdot \sum_{i=0}^{w-2} b_i \cdot 2^i$$

- two representations of zero!
  - computer must have two sets of instructions
    - one for signed arithmetic, one for unsigned

Signed Integers

- **Ones' complement**
  - **negate a number by forming its bit-wise complement**
    - » **e.g., (-1)·01101011 = 10010100**

b_{w-1} = 0 ⇒ non-negative number

$$\text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i$$

b_{w-1} = 1 ⇒ negative number

$$\text{value} = \sum_{i=0}^{w-2} (b_i - 1) \cdot 2^i$$

two zeros!

Signed Integers

- **Two's complement**

  $b_{w-1} = 0 \Rightarrow$ non-negative number

  value = $\sum_{i=0}^{w-2} b_i \cdot 2^i$

  $b_{w-1} = 1 \Rightarrow$ negative number

  value = $(-1) \cdot 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$

  one zero!

Example

- **w = 4**

```
0000: 0        1000: -8
0001: 1        1001: -7
0010: 2        1010: -6
0011: 3        1011: -5
0100: 4        1100: -4
0101: 5        1101: -3
0110: 6        1110: -2
0111: 7        1111: -1
```

Signed Integers

- **Negating two's complement**

$$value = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

  - **how to compute –*value*?**

    (~value)+1

Signed Integers

- **Negating two's complement (continued)**

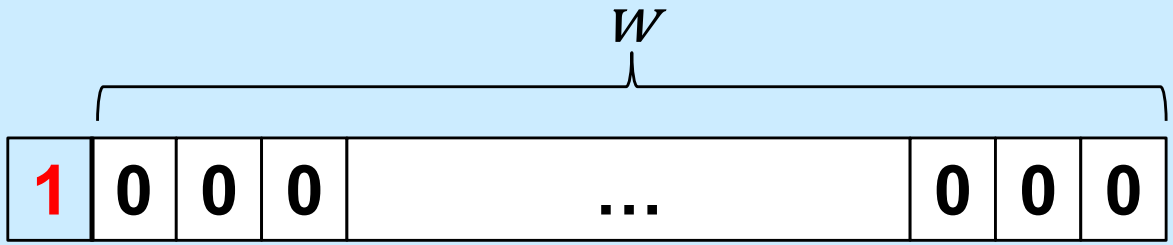
$$value + (\sim value + 1)$$

$$= (value + \sim value) + 1$$

$$= (2^w - 1) + 1$$

$$= 2^w$$

$=$



Quiz 3

- **We have a computer with 4-bit words that uses two's complement to represent signed integers. What is the result of subtracting 0010 (2) from 0001 (1)?**
  - **a) 1110**
  - **b) 1001**
  - **c) 0111**
  - **d) 1111**

Signed vs. Unsigned in C

- char, short, int, and long
  - signed integer types
  - right shift (>>) is arithmetic
- unsigned char, unsigned short, unsigned int, unsigned long
  - unsigned integer types
  - right shift (>>) is logical

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1

- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- **Other Values**
  - Minus 1
    111…1

Values for *W* = 16

	Decimal	Hex	Binary
UMax	**65535**	**FF FF**	**11111111 11111111**
TMax	**32767**	**7F FF**	**01111111 11111111**
TMin	**-32768**	**80 00**	**10000000 00000000**
-1	**-1**	**FF FF**	**11111111 11111111**
0	**0**	**00 00**	**00000000 00000000**

Values for Different Word Sizes

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **Observations**
  - $|TMin| = TMax + 1$
    - » Asymmetric range
  - $UMax = 2 * TMax + 1$

- **C Programming**
  - **#include** <limits.h>
  - declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - values platform-specific

Quiz 4

- **What is –TMin (assuming two's complement signed integers)?**
  a) **TMin**
  b) **TMax**
  c) **0**
  d) **1**