CS 33

Introduction to C
Part 6
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
#include <stdio.h>
#include <string.h>

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

```
arg1  arg2  \0
```

```
arg1  \0  arg2  \0
```

```
arg1  \0  arg2  \0
```

```
arg1  \0  arg2  \0  \0
```
Design of `getfirstword`

- `char *getfirstword(char **rem_p)`
  - returns
    » pointer to null-terminated first word in `*rem_p`
    or
    » `NULL`, if `*rem_p` is a string entirely of whitespace
  - `*rem_p` modified to
    » point to character following first word in `*rem_p` if within bounds of string
    or
    » `NULL` if next character not within bounds
Using `getfirstword`

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

**Output:**
- arg0
- arg1
- arg2
- arg3
char *getfirstword(char **rem_p) {
    char *str = *rem_p;
    if (str == NULL)
        return NULL;
    int len = strlen(str);
    int wlen =
        strspn(str, "\t\n");
    // initial whitespace
    if (wlen == len) {
        // string is all whitespace
        return NULL;
    } else {
        // no more words
        *rem_p = NULL;
        return str;
    }
    str = &str[wlen];
    // skip over whitespace
    len -= wlen;
    // length of first word
    if (wlen < len) {
        // word ends before end of string: terminate it with null
        str[wlen] = '\0';
        *rem_p = &str[wlen+1];
    } else {
        // word ends after end of string: return it
        *rem_p = &str[wlen+1];
        return str;
    }
}

int wlen =
    strcspn(str, "\t\n");
    // length of first word
if (wlen < len) {
    // word ends before end of string: terminate it with null
    str[wlen] = '\0';
    *rem_p = &str[wlen+1];
} else {
    // no more words
    *rem_p = NULL;
}
return str;
Numeric Conversions

```c
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)

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

x = i/j + y;
/* what's the value of x? */
```
Implicit Conversions (2)

```c
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

```c
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

```c
int i, j;
float a; // sizeof(float) == 4
a = (float)i/(float)j;
```

• Intimidation

```c
float x, y;
swap((int*)&x, (int*)&y);
```
Quiz 2

• Will this work?

```c
double x, y; // sizeof(double) == 8
...
swap((int *)&x, (int *)&y);
```

a) yes
b) no
Nothing, and More ...

• *void* means, literally, nothing:
  ```c
  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

```c
void swap(int *i, int *j) {
    int tmp;
    tmp = *j; *j = *i; *i = tmp;
}
/* can we make this generic? */
```
An Application: Generic Swap

```c
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

```c
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)

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

```c
void ArrayBop(int A[],
    int len,
    int (*func)(int)) {
    int i;
    for (i=0; i<len; i++)
}
```
Fun with Functions (3)

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

int main() {  
    int A[20];  
    ... /* initialize A */  
    ArrayBop(A, 20, triple);
    return 0;
}
```
For Our Next Trick ...

What’s my type?

```c
int *f0(int *a) {
    ...
}

double *f1(double *a) {
    ...
}

char *f2(char *a) {
    ...
}
```
Working Our Way There …

• An array of 3 ints
  - `int A[3];`

• An array of 3 int *s
  - `int *A[3];`

• A func returning an int *, taking an int *
  - `int *f(int *);`

• A pointer to such a func
  - `int *(*pf)(int *);`
There …

• An array of func pointers
  - \texttt{int (**pf[3])(int *)};

• An array of generic func pointers
  - \texttt{void (**pf[3])(void *)};
Using It

```c
int *f0(int *a) { *a += 1; return a; }
double *f1(double *a) { *a += 1; return a; }
char *f2(char *a) { *a += 1; return a; }
int main() {
  int x = 1;
  int *p;
  void *(*pf[3])(void *);
  pf[0] = (void *(*(*)(void *))f0);
  pf[1] = (void *(*(*)(void *))f1);
  pf[2] = (void *(*(*)(void *))f2);
  p = pf[0](&x);
  printf("%d\n", *p);
  return 0;
}
```

$ ./funcptr
2
$
Casts, Yet Again

• They tell the C compiler: “Shut up, I know what I’m doing!”

• Sometimes true

\[
pf[0] = (\text{void } (*)(\text{void } *))f0;
\]

• Sometimes false

\[
\text{long } f = 7;
\]
\[
(\text{void}(*)(\text{int}))*f(2);
\]
Laziness …

• Why type the declaration
  
  \[ \text{void } *(\ast f)(\text{void } *, \text{ void } *); \]

• You could, instead, type
  
  \[ \text{MyType } f; \]

• (If, of course, you can somehow define \textit{MyType} to mean the right thing)
typedef

• Allows one to create new names for existing types
  
  \[
  \text{typedef int } *\text{IntP}_t;
  \]

\[
\text{IntP}_t \ x;
\]

– means the same as

\[
\text{int } *x;
\]
More typedefs

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

complex_t i, *ip;
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
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

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., \((-1) \cdot 01101011 = 10010100\)

\[ b_{w-1} = 0 \Rightarrow \text{non-negative number} \]

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

\[ b_{w-1} = 1 \Rightarrow \text{negative number} \]

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

two zeros!
Signed Integers

- Two’s complement

  \[ b_{w-1} = 0 \implies \text{non-negative number} \]

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

  \[ b_{w-1} = 1 \implies \text{negative number} \]

  \[ \text{value} = (-1) \cdot 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]
Example

• \( w = 4 \)

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
</tr>
</tbody>
</table>