C Characters and Strings

CS 2060

Prof. Jonathan Ventura

Character handling

- The C Standard Library provides many functions for testing characters in `ctype.h`.

```
int isdigit(int c); // is c a digit (0-9) ?
int isalpha(int c); // is c a letter ?
int isalnum(int c); // is c a digit or a letter ?
int isspace(int c); // is c a whitespace character?
int islower(int c); // is c a lowercase letter ?
int isupper(int c); // is c an uppercase letter ?
int tolower(int c); // convert c to lowercase
int toupper(int c); // convert c to uppercase
```

Character handling

```
char string[] = "Hello, world!";
puts(string);
{
char *ptr = string;
while ( *ptr ) { *ptr = toupper(*ptr); ptr++; }
puts(string);
}
{
char *ptr = string;
while ( *ptr ) { *ptr = tolower(*ptr); ptr++; }
puts(string);
}
```

String conversion functions

- `stdlib.h` has functions to extract numbers from strings:

```
double strtod( char *nPtr, char **endPtr ); // double
long strtol( char *nPtr, char **endPtr ); // long
```

- The function sets `*endPtr` to the location after the end of the number.

```
char string[] = "10 ducks";
char *endPtr;
long num = strtol(string,&endPtr);
// num = 10
// endPtr points to the third element of string.
```

- If you don't need `endPtr` just pass `NULL`:

```
long num = strtol(string,NULL);
```

Printing to a string

- We can use `sprintf` to format a string rather than printing to the console:

```
int num_ducks = 10;
char string[1024];
sprintf("%d ducks were found.",num_ducks);
// string contains "10 ducks were found."
```

- Why is this method insecure?

Printing to a string

- `sprintf` might overflow the array, leading to a code vulnerability:

```
int num_ducks = 10;
char string[10];
sprintf(string,"%d ducks were found.",num_ducks);
// string is not long enough -- overflow
```

- Instead we can use `snprintf` which includes the size of the string:

```
int num_ducks = 10;
char string[10];
snprintf(string,10,"%d ducks were found.",num_ducks);
// string contains "10 ducks "
```

Reading from a string

- Similarly we can read values from a string using `sscanf`:

```
char string[] = "10 ducks were found.";
int num_ducks;
char string2[1024];
sscanf(string,"%d %s",&num_ducks,string2);
// num_ducks = 10, string2 = "ducks were found."
```

- Notice any vulnerabilties here?

Reading from a string

- Again it is safer to include the length of the string in the format specifier:

```
char string[] = "10 ducks were found.";
int num_ducks;
char string2[6];
sscanf(string,"%d %5s",&num_ducks,string2);
// num_ducks = 10, string2 = "ducks"
```

- Remember to leave room for the null terminator at the end of the string.

Copying a string

- We can copy strings using `strcpy` or `strncpy`:

```
char *strcpy( char *to, const char *from );
char *strncpy( char *to, const char *from, size_t n );
```

- Remember that the first argument is the destination and the second argument is the source – this is typical in the C Standard Library.

Copying a string

- We can copy strings using `strcpy` or `strncpy`:

```
char *strcpy( char *to, const char *from );
char *strncpy( char *to, const char *from, size_t n );
```

- Remember that the first argument is the destination and the second argument is the source – this is typical in the C Standard Library.

Concatenating strings

- We can *concatenate* strings using `strcat` or `strncat`:

```
char *strcat( char *to, const char *from );
char *strncat( char *to, const char *from, size_t n );
```

- The `from` string will be appended to the end of the `to` string.

```
char result[1024];
char s1[] = "Hello";
char s2[] = ", ";
char s3[] = "World!";
strcpy(result,s1);
strcat(result,s2);
strcat(result,s3);
printf("%s\n",result);
```

Comparing strings

- We can compare strings using `strcmp` or `strncmp`:

```
int strcmp( char *s1, const char *s2 );
int strncmp( char *s1, const char *s2, size_t n );
```

- These functions return:
  - 0 if the strings are equal
  - a negative value if s1 $<$ s2
  - a positive value if s1 $>$ s2
- Here $<$ and $>$ refer to phone book ordering:

`"Aardvark"` $<$ `"Apple"` $<$ `"Banana"`

Comparing strings

```
int red_count = 0, blue_count = 0, green_count = 0;
char color[16];
printf("Enter color: ");
scanf(" %15s",color);
if ( strcmp( color, "red" ) == 0 ) {
    red_count++;
} else if ( strcmp( color, "blue" ) == 0 ) {
    blue_count++;
} else if ( strcmp( color, "green" ) == 0 ) {
    green_count++;
}
puts("");
```

String search functions

- There are many string search functions available; we will discuss a couple here.

```
char *strstr(const char *s1, const char *s2);
```

- `strstr` finds the *first occurence* of `s2` in `s1` and returns a pointer to the location in `s1`.
- If `s2` is not found, the function returns `NULL`.

String search functions

```
void printLocation( const char *string, const char *key ){
    char *loc = strstr(string,key);
    if ( loc1 != NULL )
        printf("%s found at index %d\n",key1,loc1-string);
    else
        printf("%s not found\n",key);
}
int main() {
char string[] = "String to search";
char key1[] = "search";
char key2[] = "duck";
printLocation(string,key1);
printLocation(string,key2);
}
```

String tokenization

- The C Standard Library provides a *tokenizer* function `strtotok`:

```
char *strtok( char *string, const char *separators );
```

- The string is broken into tokens separated by any character in `separators`.
- The return value is a pointer to a string containing the next token.
- The second and later calls to `strtok` should pass `NULL` as the first argument.
- The function actually modifies `string` by inserting null terminators!

String tokenization

```
char string[] = "Hello world, it's me the computer.";
char separators[] = " ,.";

// tokenize string by space or punctuation
// first call to strtok: pass string
char *tokenPtr = strtok(string, separators);

while ( tokenPtr != NULL ) {
    printf( "%s\n", tokenPtr );
    // get next token: pass NULL
    tokenPtr = strtok(NULL, separators);
}
```

String length

- `strlen` finds the length of a string:

```
size_t strlen( const char *str );
```

- How does it calculate the length of the string?

Other string functions

- `strlen` finds the length of a string:

```
size_t strlen( const char *str );
```

- How does it calculate the length of the string?
- It searches for the null terminator, starting from the pointer.

Memory functions

- `string.h` also contains functions for manipulation of any kind of data.

```
void *memcpy( void *to, void *from, size_t num_bytes );
```

- `memcpy` copies `num_bytes` bytes starting from `from` to `to`.
- What is the difference from `strcpy` ?

Memory functions

- `string.h` also contains functions for manipulation of any kind of data.

```
void *memcpy( void *to, void *from, size_t num_bytes );
```

- `memcpy` copies `num_bytes` bytes starting from `from` to `to`.
- It does not look for the null-terminator (instead, size of array is given).
- This is more efficient when the number of bytes to copied is known.

memcpy example

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

int main()
{
    char array1[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    char array2[] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };

    memcpy( array1, array2, 5 );

    for ( int i = 0; i < 10; i++ ) printf("%d ",array1[i]);
    puts("");
}
```

What will this output?

memcpy example

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

int main()
{
    int array1[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int array2[] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };

    // memcpy ?

    for ( int i = 0; i < 10; i++ ) printf("%d ",array1[i]);
    puts("");
}
```

How can we copy the first five integers from array2 to array1?

memcpy example

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

int main()
{
    int array1[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int array2[] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };

    memcpy( array1, array2, 5*sizeof(int) );

    for ( int i = 0; i < 10; i++ ) printf("%d ",array1[i]);
    puts("");
}
```

Copy 5 $\times$ (number of bytes in one integer).

memcpy example

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

int main()
{
    int array1[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // undefined behavior:
    memcpy( array1, array1+6, 6*sizeof(int) );

    for ( int i = 0; i < 10; i++ ) printf("%d ",array1[i]);
    puts("");
}
```

What is the issue here?

Memory functions

- `memcpy` causes undefined behavior when the two pieces of memory overlap.
- In this case we need to use `memmove`:

```
void *memmove( void *to, void *from, size_t num_bytes )
```

- `memmove` behaves as if we first copy `from` to a temporary block of memory, and then copy it to `to`.
- It is safe to use when the two blocks of memory overlap.

Memory functions

- `memset` sets an entire block of memory to the same (byte) value:

```
void *memset( void *to, int value, size_t num_bytes );
```

- `value` is converted to a char and then copied to `num_bytes` starting from `to`.

memset example

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

int main()
{
    char str[] = "The password is secret";

    memset( str+16, '*', 6 );

    printf("%s\n",str);
}
```

What will the output be?