C Programming, Part 2: Text Input And Output

jakebailey edited this page on Dec 16, 2014 · 3 revisions

How do I print strings, ints, chars to the standard output stream?

Use `printf`. The first parameter is a format string that includes placeholders for the data to be printed. Common format specifiers are `%s` treat the argument as a c string pointer, keep printing all characters until the NULL-character is reached; `%d` print the argument as an integer; `%p` print the argument as a memory address.

A simple example is shown below:

```
char *name = ... ; int score = ...;
printf("Hello %s, your result is %d\n", name, score);
printf("Debug: The string and int are stored at: %p and %p\n", name, &score );
// name already is a char pointer and points to the start of the array.
// We need "&" to get the address of the int variable
```

By default, for performance, `printf` does not actually write anything out (by calling write) until its buffer is full or a newline is printed.

How else can I print strings and single characters?

Use `puts( name );` and `putchar( c )` where name is a pointer to a C string and c is just a `char`

How do I print to other file streams?

Use `fprintf( _file_ , "Hello %s, score: %d", name, score);` Where _file_ is either predefined 'stdout' 'stderr' or a FILE pointer that was returned by `fopen` or `fdopen`

How do I print data into a C string?

Use `sprintf` or better `snprintf`.

```
char result[200];
int len = snprintf(result, sizeof(result), "%s:%d", name, score);
```

snprintf returns the number of characters written excluding the terminating byte. In the above example this would be a maximum of 199.

How do I parse input using `scanf` into parameters?

Use `scanf` (or `fscanf` or `sscanf`) to get input from the default input stream, an arbitrary file stream or a C string respectively. It's a good idea to check the return value to see how many items were parsed. `scanf` functions require valid pointers. It's a common source of error to pass in an incorrect pointer value. For example,

```
int *data = (int *) malloc(sizeof(int));
char *line = "v 10";
char type;
// Good practice: Check scanf parsed the line and read two values:
int ok = 2 == sscanf(line, "%c %d", &type, &data); // pointer error
```

We wanted to write the character value into c and the integer value into the malloc'd memory. However we passed the address of the data pointer, not what the pointer is pointing to! So `sscanf` will change the pointer itself. i.e. the pointer will now point to address 10 so this code will later fail e.g. when free(data) is called.

How do I stop scanf from causing a buffer overflow?

The following code assumes the scanf won't read more than 10 characters (including the terminating byte) into the buffer.

```
char buffer[10];
scanf("%s",buffer);
```

You can include an optional integer to specify how many characters EXCLUDING the terminating byte:

```
char buffer[10];
scanf("%9s", buffer); // reads upto 9 charactes from input (leave room for the 10
```

Why is `gets` dangerous? What should I use instead?

The following code is vulnerable to buffer overflow. It assumes or trusts that the input line will be no more than 10 characters, including the terminating byte.

```
char buf[10];
gets(buf); // Remember the array name means the first byte of the array
```

`gets` is deprecated and will be removed in future versions of the C standard. Programs should use `fgets` or `getline` instead.

Where each have the following structure respectively:

```
char *fgets (char *str, int num, FILE *stream);

ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Here's a simple, safe way to read a single line. Lines longer than 9 characters will be truncated:

```
char buffer[10];
char *result = fgets(buffer, sizeof(buffer), stdin);
```

The result is NULL if there was an error or the end of the file is reached. Note, unlike `gets`, `fgets` copies the newline into the buffer, which you may want to discard-

```
if (!result) { return; /* no data - don't read the buffer contents */}

int i= strlen(buffer) -1;
if (buffer[i] == '\n') buffer[i] = '\0'
```

How do I use `getline`?

One of the advantages of `getline` is that will automatically (re-) allocate a buffer on the heap of sufficient size.

```
// ssize_t getline(char **lineptr, size_t *n, FILE *stream);

 /* set buffer and size to 0; they will be changed by getline*/
char *buffer = NULL;
size_t size = 0;

ssize_t chars = getline(&buffer, &size, stdin);

// Discard newline character if it is present,
if (chars > 0 && buffer[chars -1] =='\n') buffer[chars-1] = '\0';

// Read another line.
// The existing buffer will be re-used, or, if necessary,
// It will be `free`'d and a new larger buffer will `malloc`'d
chars = getline(&buffer, &size, stdin);

// Later... don't forget to free the buffer!
free(buffer);
```