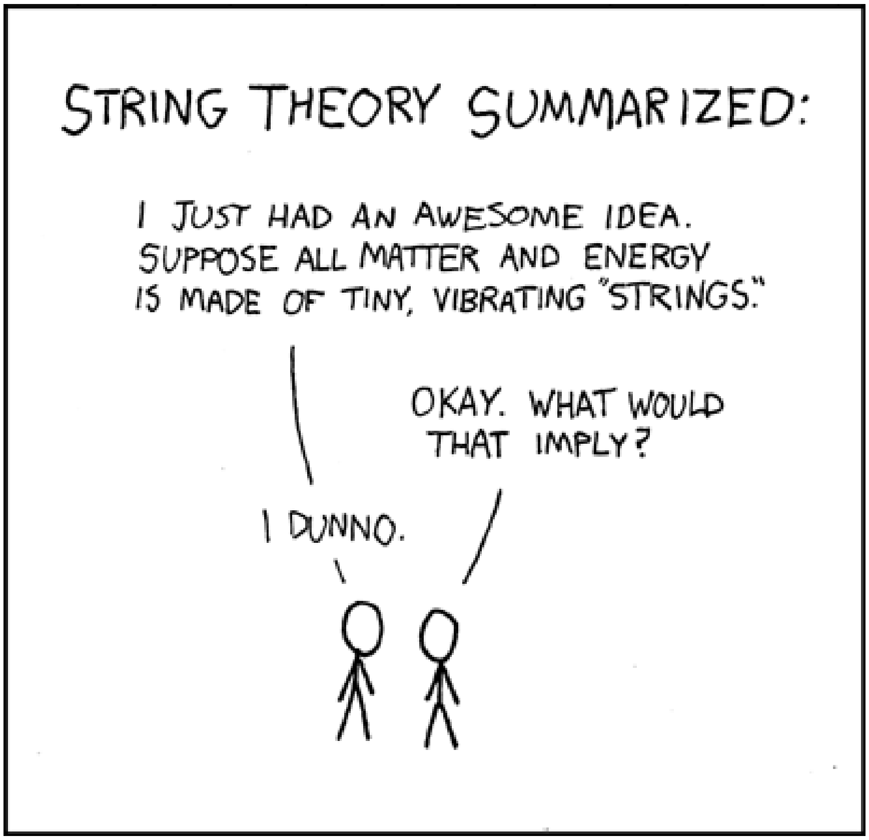
Strings

Marcus Birkenkrahe

November 26, 2024


STRING THEORY SUMMARIZED:
I JUST HAD AN AWESOME IDEA. SUPPOSE ALL MATTER AND ENERGY IS MADE OF TINY, VIBRATING "STRINGS."
OKAY. WHAT WOULD THAT IMPLY?
I DUNNO.

Motivation

- We have used `char` values and arrays but not strings.
- We cover string constants (aka literals) and variables:

1. Rules for string literals
2. Declaring string variables
3. Reading and writing strings
4. Writing functions that process strings
5. Understanding string-handling functions in the C library
6. Setting up pointer arrays to strings of different length

- Focus on examples with short explanations
- Review questions
- Exercises
- Programming assignments
- Video playlist

Escape Sequences in String Literals

- **Definition.** A *string literal* is a sequence of characters enclosed within double quotes
- **Example:** Formatting strings in `printf` or `scanf` statement

```
printf("hello\n"); // without format specifier & escape sequence ‘\n‘
printf("%d is smaller than %g\n", 100, 100.1); // with format specifier
```

```
hello
100 is smaller than 100.1
```

- **Example** - different escape sequences:

```
printf("\"The world will never starve for want of wonders;\nbut only for want of w
```

```
"The world will never starve for want of wonders;
but only for want of wonder."
G.K. Chesterton
```

Continuing a String Literal

- **Example**: Continuing a string literal on the next line

```
printf("\"Man only likes to count his troubles; \
he doesn't calculate his happiness.\" \
 \nFyodor Dostoyevsky, Notes from Underground.\n");
```

```
"Man only likes to count his troubles; he doesn't calculate his happiness."
Fyodor Dostoyevsky, Notes from Underground.
```

- A better way to join two string literals: When the compiler finds white space between string literals, it will join them.

```
printf("\"To know and love one other human being"
 " is the root of all wisdom.\"\n"
 "Evelyn Waugh, Brideshead Revisited");
```

```
"To know and love one other human being is the root of all wisdom."
Evelyn Waugh, Brideshead Revisited
```

How String Literals are Stored

- C treats string literals as character arrays: When the compiler encounters a string literal of length `n`, it sets aside `n+1` bytes of memory for the string - the characters of the string plus the *null* character.

- The null character is a byte whose bits are all zero, so it's represented by the `\0` escape sequence.

- The empty string `""` is stored as a single null character.

- Example: When `printf("abc")` is called, it is passed the address of `"abc"`, a pointer to where the leetter `a` is stored in memory.

Operations on String Literals

- We can use a string literal wherever C allows a `char *` pointer. The assignment below does not copy `"abc"`. It makes p point to the first character of the string:

  ```
  char *p;
  p = "abc";
  ```

- Pointers can be subscripted, and so can string literals.

  ```
  char ch;
  ch = "abc"[1];
  printf("%c\n",ch);
  ```

  ```
  b
  ```

- In the example, `ch` is assigned the character 'b'. As a char it can only hold one character. If if were declared as a `char *` pointer, it would point to a character rather than store one.

- Other possible subscripts are 0 ('a'), 2 ('c'), and 3 (the null character). This is used in the following function:

  ```
  // convert number in (0,15) to a character that represents a
  // hexadecimal digit
  char digit_to_hex_char(int digit)
  {
    return "0123456789ABCDEF"[digit];
  }

  int main(void)
  {
    int digit = 15;
    char hex = digit_to_hex_char(digit);
    printf("%d to %c\n", digit, hex);
    return 0;
  }
  ```

```
15 to F
```

- What happens when you try to modify a string literal?

  ```
  char *p = "abc"; // p points to ’a’

  printf("%c", p[1+1]);


  c
  ```

- This leads to a segmentation fault

```
char *p = "abc"; // p points to ’a’
*p = ’d’; // segmentation fault!
```

String Literals versus Character Constants

- A string literal containing a single character is not the same as a character constant.
- The string literal "a" is represented by a **pointer** to a memory location that contains the character ’a’ followed by ’\0’:

  ```
  printf("\n"); // printf expects a pointer as its first argument
  ```

- The character constant ’a’ is represented by an **integer** (the numerical code for the character).

  ```
  printf("%d",’\n’); // prints ASCII code for ’\n’
  printf("\n%d",’\0’); // prints ASCII code for ’\0’


  10
  0
  ```

Initializing String Variables

- C does not have a `string` type for declaring string variables.
- In C, any one-dimensional array of characters can be used to store a string, which is terminated by a *null* character.
- To store a string of up to 80 characters, declare the variable to be an array of 81 characters:

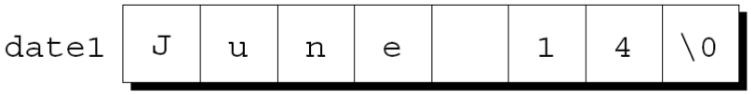
```
#define STR_LEN 80
char str[STR_LEN+1];
```

- A string variable can be initialized when it’s declared:

```
char date1[8] = "June 14"; // storing 7 characters in an 8 char array
printf("%s\n", date1);
```

```
June 14
```

- `date1` looks like this:



- `"June 14"` **is not a string literal**! It is an abbreviation for an array initializer. The following statement is equivalent:

```
char date1[8] = {’J’, ’u’, ’n’, ’e’, ’ ’, ’1’, ’4’, ’\0’};
printf("%s\n", date1);
```

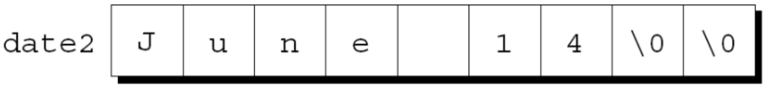
```
June 14
```

- If the initializer is too short to fill the string variable, the compiler adds extra null characters:

```
char date2[9]="June 14"; // storing 7 characters in a 9 char array
printf("%s\n", date2);
```

```
June 14
```

- `date2` looks like this:

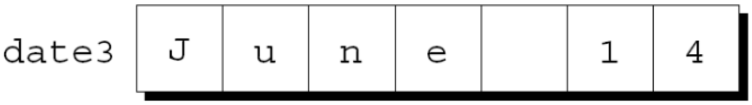


- If the initializer is exactly as long as the string variable, the null character is omitted, and the array cannot be used as a string.
- In this example, you can not guarantee that the printout will work because `printf` expects the next character to be null but it may not

```
char date3[7]="June 14"; // storing 7 characters in a 7 char array
printf("%s\n", date3);
```

```
June 14
```

- `date3` looks like this:



- Here is a longer program to test this:

```
char date3[7] = "June 14"; // No null terminator
char following_data[] = "Extra";

// Print ‘date3‘ normally (this might print garbage after "June 14" due to lack of
printf("Output without null terminator: %s\n", date3);

// Manually insert a ‘\0‘ in ‘following_data‘ to control the behavior
```

```
following_data[0] = '\0';

// Now print again to see if output changes
printf("Output after inserting a null character nearby: %s\n", date3);

// Use memory addresses to directly inspect contents
for (int i = 0; i < 15; i++) {
  printf("Byte %d after date3: %c (0x%X)\n", i, *(date3 + i), *(unsigned char *)(c
 }
```

```
Output without null terminator: June 14
Output after inserting a null character nearby: June 14
Byte 0 after date3: J (0x4A)
Byte 1 after date3: u (0x75)
Byte 2 after date3: n (0x6E)
Byte 3 after date3: e (0x65)
Byte 4 after date3:   (0x20)
Byte 5 after date3: 1 (0x31)
Byte 6 after date3: 4 (0x34)
Byte 7 after date3:  (0x0)
Byte 8 after date3: D (0x44)
Byte 9 after date3: u (0x75)
Byte 10 after date3:  (0xB9)
Byte 11 after date3: (0xC9)
Byte 12 after date3:  (0x6)
Byte 13 after date3:  (0x86)
Byte 14 after date3: (0xD9)
```

- If the initializer is longer than the string variable, we have undefined behavior just like with any array.

```
char date[6]="June 14"; // storing 7 characters in a 6 char array
printf("%s\n", date);
printf("%c\n", date[6]);
```

```
June 1
```

- If the declaration omits the length, the compiler computes it:

```
char date4[] = "June 14";
printf("%s\n", date4);
printf("%zu\n", sizeof(date4)); // length of string = 7 + 1 = 8
printf("%zu\n", sizeof(date4[0])); // length of 1 char = 1
```

```
June 14
8
1
```

Exercises: Initializing String Variables

Exercise 1: Basic Initialization

- Declare a character array called `greeting` with a length of 10 and initialize it with the string `"Hello"`. Use `printf` to print the string.

```
char greeting[10] = "Hello";
printf("%s\n", greeting);
```

```
Hello
```

Exercise 2: Exact Fit without Null Character

- Declare a character array `month` with a length of 5 and initialize it with the string `"June"`. Print the string and observe the result.

- Modify the code to add a `\0` at the end of `month` and print it again.

```
char month[5] = "June";
printf("%s\n", month); // Modify here to add \0 if needed
```

```
June
```

Exercise 3: Overfilled Array

- Declare a character array `day` with length 3 and try initializing it with the string `"Mon"`. Print the array using `printf`.

```
char day[3] = "Mon";
printf("%s\n", day);
```

```
Mon
```

Exercise 4: Using `sizeof` with Strings

- Declare `char name[] = "Student";`. Print the size of `name` using `sizeof` and compare it with the length of the string.
- What is the relationship between `sizeof(name)` and the number of characters in `"Student"`?

```
char name[] = "Student";
printf("Size of name: %zu\n", sizeof(name));
```

```
Size of name: 8
```

Character Arrays versus Character Pointers

- Strings can be defined either as arrays or as pointers:

```
char date_a[] = "June 14";
char *date_p  = "June 14";

printf("%s\n", date_a); // printing array
printf("%s\n", date_p); // printing pointer
```

```
June 14
June 14
```

- What are the differences?

1. In the array version, the characters stored in `date_a` can be modified as elements of an array.
2. In the pointer version, `date_p` points to a string literal, and cannot be modified.
3. In the array version, `date_a` is an array name.
4. In the pointer version, `date_p` is a variable that can be made to point to other strings during program execution.

- The declaration `char *p;` sets aside memory for a pointer variable, but not for a string. To do that, we must write:

```
#define STR_LEN 7

char str[STR_LEN+1]; // declare string for STR_LEN characters
char *p; // declare pointer that points to a character

p = str; // p now points at str[0]
```

- With a string:

```
char str[8] = "June 14", *p;

p = str; // p now points at str[0]

printf("%c %c %s\n", p[0], *p, p);


J J June 14
```

- Using an uninitialized pointer variable as a string is an error:

```
char *p; // pointer points nowhere in particular
// p[0] = 'a'; // segmentation fault
```

- Explanation:

  > Since `p` has not been initialized, we don't know where it is pointing, and using it to write characters into memory causes undefined behavior.

Exercises: Character Arrays versus Character Pointers

Exercise 1: String Array vs. Pointer Modification Attempt

- Declare `char message_a[] = "Welcome";` and `char *message_p = "Welcome";`.
- Try modifying the first character of each: `message_a[0] = 'w';` and `message_p[0] = 'w';`.

```
char message_a[] = "Welcome";
char *message_p = "Welcome";
message_a[0] = 'w'; // Should work
// message_p[0] = 'w'; // Uncomment to observe behavior
```

Exercise 2: Changing the Pointer Target

- Initialize two character arrays `char date1[] = "June 14";` and `char date2[] = "July 15";`.
- Declare a character pointer `char *p` and make it point to `date1`. Print the string using `p`, then change `p` to point to `date2` and print again.

```
char date1[] = "June 14";
char date2[] = "July 15";
char *p = date1;
printf("%s\n", p);
p = date2;
printf("%s\n", p);
```

```
June 14
July 15
```

Exercise 3: Pointer Initialization and Dereferencing

- Declare a character array `char city[] = "Paris";` and a character pointer `char *ptr`.
- Initialize `ptr` to point to `city` and print the first character of `city` by dereferencing `ptr` with `*ptr` and `ptr[0]`.

```
char city[] = "Paris";
char *ptr = city;
printf("First character: %c %c\n", *ptr, ptr[0]);
```

```
First character: P P
```

Exercise 4: Uninitialized Pointer Error Simulation

- Declare a character pointer `char *uninitialized_ptr;`. Attempt to assign `uninitialized_ptr[0] = 'X';`.

```
char *uninitialized_ptr;
// uninitialized_ptr[0] = 'X'; // Uncomment to see runtime error
```

Writing Strings Using `printf` and `puts`

- Writing strings is easy with `printf` or `puts`

```
char str[]= "Are we having fun yet?";
printf("%s\n",str);
```

```
Are we having fun yet?
```

- `printf` will continue to print until it finds a null character in memory.
- To print just part of a string on a field of size `m`, use the conversion spec `%m.ps` where `p` is the number of characters to be displayed.

```
char str[]= "Are we having fun yet?";

// print part of string
printf("%.6s\n", str);

puts("|----|----|----|"); // `puts` always ends with an \n
// print part of string on field of length 10 (right-aligned)
printf("%10.6s\n", str);
// print part of string on field of length 10 (left-aligned)
printf("%-10.6s\n", str);
```

```
Are we
|----|----|----|
    Are we
Are we
```

Exercises: Writing strings

Exercise 5: Simple String Output

- **Problem**: Write a program that stores the string `"Learning C is fun!"` in a variable and prints it to the console.
- **Solution**:

```
char str[] = "Learning C is fun!";
printf("%s\n", str);
```

```
Learning C is fun!
```

Exercise 6: Partial String Output

- **Problem**: Modify the previous program to only print the first 10 characters of the string.
- **Solution**:

```
char str[] = "Learning C is fun!";
printf("%.10s\n", str);  // Prints "Learning C"
```

```
Learning C
```

Exercise 7: String Formatting with Field Width

- **Problem**: Use the string `"C programming"` and:

  1. Print only the first 5 characters in a field of width 8, right-aligned.
  2. Print only the first 5 characters in a field of width 8, left-aligned.
  3. Print a ruler (`|....|....|....|`) to check your results.

- **Solution**:

```
char str[] = "C programming";
puts("|....|....|....|");
printf("%8.5s\n", str);  // Right-aligned
printf("%-8.5s\n", str); // Left-aligned
```

```
|....|....|....|
   C pro
C pro
```

Exercise 8: Safe String Input with fgets

- **Problem**: Write a program that reads a line of text using `fgets` and then prints it. Use the `:cmdline < strinput` header argument to stream the data to the program.

- String input:

  ```
  echo "To C or not to C that is the question" > strinput
  cat strinput
  ```

- **Solution**:

  ```
  char buffer[80];  // Buffer to store input
  fgets(buffer, sizeof(buffer), stdin);
  printf("You entered: %s", buffer);
  ```

  ```
  You entered: Are we having fun?
  ```

Reading strings using `scanf` and `gets`

- Reading a string is harder because input string and string variable must have the same length.

- To read, we can use `scanf` or `gets`, or read strings one character at a time.

- There's no need for an address-of operator `&` in front of `str` in the call, because it is treated like as a pointer when passed to a function, and it always stores a null character at the end of the string:

```
char str[19];
scanf("%s", str);
printf("%s\n", str);
```

```
Are
```

- Input file

```
echo "Are we having fun?" > strinput
cat strinput
```

```
Are we having fun?
```

- Explanation: When `scanf` is called, it skips white space, then reads characters and stores them in `str` until it encounters white space. It cannot read a full line of input with white space.
- To do this, use `gets`: It does not skip whitespace at the start, and it reads until it finds a `\n` (newline) character - it discards it and replaces it by `\0`.

```
char str[19];
gets(str); // unsafe: use ‘fgets‘
printf("%s\n", str);
```

```
Are we having fun?
```

- Both `scanf` and `gets` are inherently *unsafe* because they may store characters past the end of the array:

  1. `scanf` can be made safer with `%ns` where `n` is the maximum number of characters to be stored.
  2. `fgets` is a safe alternative to `gets`, but it requires us to understand file streams (check the Linux `man` page for info).

- Example:

```
char str[19];  // Str to store the input

fgets(str, sizeof(str), stdin);
printf("%s", str);
```

```
Are we having fun?
```

Reading strings character by character

- Regular subscripting:

```
char str[]="Are we having fun?";
int i;

for (i=0; i < 18; i++) // goes to N-1 to avoid printing \0
  printf("%c", str[i]);
```

```
Are we having fun?
```

- More elegant: use the knowledge about strings!

```
char str[]="Are we having fun?";
int i;

for (i=0; str[i] != ’\0’; i++) // loop until first ’\0’
  printf("%c", str[i]);
```

```
Are we having fun?
```

Accessing the characters in a string (counting spaces)

- Write a function that counts the number of spaces in a string

```
// count spaces in a string - return integer count value
int count_spaces(const char s[])
{
  int count=0, i; // counter and loop variables
  for (i=0; s[i] != '\0'; i++)
    if (s[i] == ' ')
count++;
  return count;
}

int main(void)
{
  char str[]="Are we having                fun?";

  // str decays to pointer to str[0] upon function call
  printf("Spaces in \"%s\" = %d\n", str, count_spaces(str));

  return 0;
}
```

```
Spaces in "Are we having                fun?" = 18
```

- Using pointer arithmetic is especially convenient for strings:

```
int count_spaces(const char *s)
{
  int count = 0;

  for( ; *s != '\0'; s++ )
    if (*s == '\0')
count++;
  return count;
}
int main(void)
{
  char str[]="Are we having fun?";

  // str decays to pointer to str[0] upon function call
  printf("Spaces in \"%s\" = %d\n", str, count_spaces(str));

  return 0;
}
```

```
Spaces in "Are we having fun?" = 0
```

Using the C string library `<string.h>`

- Straight copying won't work!

  ```
  char str1[10], str2[10];

  str1 = "abc"; // wrong: array cannot be an lvalue
  str2 = str1; // wrong
  ```

- This will compile but with nonsense results:

  ```
  if (str1 == str2) // wrong: no vectorization
  ```

- Get length with `strlen`
- Copy with `strcpy` (safe: `strncpy`)
- Compare with `strcmp`
- Concatenate with `strcat` (safe: `strncat`)

Example: `strcpy`

- Example prototype for `strcpy`

  ```
  // function: strcpy
  // returns a char pointer
  // copies string s2 into destination string s1
  // it does not change the source string s2
  char *strcpy(char *s1, const char *s2);
  ```

- Quick example:

  ```
  char str[80];
  printf("%s\n",str);

  strcpy(str,"Hello");

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

  @#[
  Hello
  ```

Command-line arguments

- Programs often need to be supplied with information from `stdin` - a file name, or a switch/flag/option that modifies the programs behavior:

  ```
  ls -l [7]*
  ```

  ```
  -rw-rw-r-- 1 aletheia aletheia   9079 Nov 14 14:48 7_strings_codealong_copy.org
  -rw-rw-r-- 1 aletheia aletheia   6959 Nov 14 00:07 7_strings_codealong.org
  -rw-rw-r-- 1 aletheia aletheia  25793 Nov 26 19:58 7_strings.org
  -rw-rw-r-- 1 aletheia aletheia 159651 Nov 26 19:57 7_strings.pdf
  -rw-rw-r-- 1 aletheia aletheia  31160 Nov 26 19:57 7_strings.tex
  ```

  ```
  which ls
  ```

- In the example,

  1. `ls` is the C program (executable for file listing with PATH),
  2. `-l` is the flag (long listing), and
  3. `[7]*` is a regular expression (all files beginning with 7).

- Command-line information is available to all files not just OS commands. To access these *command-line arguments*, we define `main` as a function not without (`void`) but with two parameters, `argc` and `argv`:

  ```
  int main(int argc, char *argv[])
  {
    return 0;
  }
  ```

- `argc` is the "argument count": The number of command-line arguments (including the name of the program itself.

- `argv` is the "argument vector": An array of pointers to the command-line arguments, stored in string form.

- `argv[0]` points to the name of the program, `argv[1]` to `argv[argc-1]` point to the remaining command-line arguments, while the last element, `argv[argc]` is always a null pointer that points nowhere.

- The preprocessor macro NULL represents a null pointer (cannot be dereferenced - has no value equivalent):

```
int *ptr = NULL; // NULL macro used to define a null pointer

if (ptr == NULL) {
  printf("Pointer is null.\n");
 } else {
  printf("Pointer is not null.\n");
 }


Pointer is null.
```

- Example: If the user enters `gcc main.c -o main`, `argv` points to the following values:

Index	Value	Explanation
0	"gcc"	The name of the program being executed (with path)
1	"main.c"	The first argument, the input source file + \0
2	"-o"	The option flag specifying output file name + \0
3	"main"	The name of the output file + \0
4	NULL	Null pointer

- If the program name is not available, `argv[0]` points to an empty string.

- You can print the command-line arguments with array subscripting:

```
int main(int argc, char *argv[])
{
  int i;
  for(i=1;i<argc;i++)
    printf("%s ", argv[i]);
  puts("");
  return 0;
}
```

- To test this program, compile it and run it on any command-line input.

```
gcc argv.c -o CMD
./CMD foo bar baz
./CMD ls -l main.c


foo bar baz
ls -l main.c
```

- You can also print the command-line arguments with pointer arithmetic:

  ```
  int main(int argc, char *argv[])
  {
    char **p; // p is a pointer to a pointer to a character

    // start: p points to argv[1] which points to the first string after
    // the program name
    // stop: when the pointer is the NULL pointer
    // increment: go to next pointer (element of argv)
    for (p = &argv[1]; *p != NULL; p++)
      printf("%s ", *p); // print dereferenced pointer = stored string
    puts("");
    return 0;
  }
  ```

- Testing:

  ```
  gcc argv2.c -o CMD
  ./CMD foo bar baz
  ./CMD ls -l main.c


  foo bar baz
  ls -l main.c
  ```

- Why would you not just use `main(void)`?

You might want to use `main` like a function, and not just like a program block.

- Using `main` with the full parameter list also simplifies code testing. Check e.g. this program to sum two numbers.

  1. Check if the number of arguments is correct & print usage.
  2. Convert command-line arguments to integers.
  3. Print the integers and their sum.

```
int main(int argc, char *argv[])
{
  if (argc != 3) { // argv[0] points to the executable, argv[1],
     // argv[2] point to the two numbers to be added
    printf("Usage: %s <num1> <num2>\n", argv[0]); // name of program
    return 1;
  }
  // convert command-line arguments to integers
  int num1 = atoi(argv[1]);
  int num2 = atoi(argv[2]);

  // print the sum of the two numbers
  printf("The sum of %d and %d is %d\n", num1, num2, num1 + num2);
  return 0;
}
```

Test: When the command fails, you now get a return code 1 and usage information.

```
gcc sum.c -o sum
./sum 10 20 # this works
echo 10 20 | ./sum  # this does not work
```

```
The sum of 10 and 20 is 30
Usage: ./sum <num1> <num2>
```