CSCI-UA.0201

Computer Systems Organization

C Programming – Pointers, Structs, Arrays

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
Pointers:
Very powerful but also dangerous concept!
Can a function modify its arguments?

What if we wanted to implement a function `pow_assign()` that *modified* its argument, so that these are equivalent:

```c
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

```c
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* p is 32.0 here */
```

Would this work?

```c
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}
```
NO!

Remember the stack!

```c
void pow_assign(float x, unsigned int exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

main()
{
    float p=2.0;
    pow_assign(p, 5);
}
```

In C, all arguments are passed by value

But, what if the argument is the *address* of a variable?

<table>
<thead>
<tr>
<th>float x</th>
<th>32.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32_t exp</td>
<td>5</td>
</tr>
<tr>
<td>float result</td>
<td>32.0</td>
</tr>
<tr>
<td>float p</td>
<td>2.0</td>
</tr>
</tbody>
</table>
# Passing Addresses

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char x</td>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>char y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

address of x: 4
memory content at address 4: 72
This is exactly how “pointers” work.

• address of x: &x
• if y is an address, the content of the memory at that address: *y

A “pointer type”: pointer to char

```c
void f(char * p)
{
    *p = *p - 32;
}
```

```c
char y = 101; /* y is 101 */
f(&y); /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

Pointers are used in C for many other purposes:
• Passing large objects without copying them
• Accessing dynamically allocated memory
• Passing functions to other functions
• Implement functions with multiple return values
A valid pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior, and will often cause your OS to kill your process (SEGV or Segmentation Fault).

There are two general causes for these errors:
• Program errors that set the pointer value to an invalid address
• Use of a pointer that was at one time valid, but later became invalid

Will ptr be valid or invalid?

```c
char * get_pointer() {
    char x=0;
    return &x;
}

void foo() {
    char * ptr = get_pointer();
    *ptr = 12;  /* valid? */
}
```
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer()
{
    char x=0;
    return &x;
}
void foo()
{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
    other_function();
}
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!
Now that we know about pointers, let’s go back to types.
## More on Types

We’ve seen a few types at this point: char, int, float, char *

Types are important because:
- They allow your program to impose logical structure on memory
- They help the compiler tell when you’re making a mistake

In the next slides we will discuss:
- How to create logical layouts of different types (structs)
- How to use arrays
- How to parse C type names (there is a logic to it!)
- How to create new types using typedef
Structures

• a collection of related data items
• possibly of different types
• defined using the keyword `struct`
• The members of a struct type variable are accessed with the dot (.) operator:
  `<struct-variable>..<member_name>`
struct basics

• Definition of a structure:

```c
struct <struct-name> {  
    <type> <identifier_list>;  
    <type> <identifier_list>;  
    ...  
} ;
```

Each identifier defines a member of the structure.
struct basics

• Example:

```c
struct Address {
    int zip;
    char street[50];
    char city[20];
} ;
```

```c
main()
{
    struct Address addrs;
    ...
    addrs.zip = 10012;
}
```

Example of initializing a structure

```c
struct Address addr = {10012, “Mercer”, “New York”};
```
Arrays

Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

/* define an array of 5 chars */
char x[5] = {'t','e','s','t','\0'};

/* accessing element 0 */
x[0] = 'T';

/* pointer arithmetic to get elt 3 */
char elt3 = *(x+3); /* x[3] */

/* x[0] evaluates to the first element;
 * x evaluates to the address of the
 * first element, or &(x[0]) */

/* 0-indexed for loop idiom */
#define COUNT 10
char y[COUNT];
int i;
for (i=0; i<COUNT; i++) {
    /* process y[i] */
    printf("%c\n", y[i]);
}

Brackets specify the count of elements. Initial values optionally set in braces.

Arrays in C are 0-indexed (here, 0..4)

x[3] == *(x+3) == ‘t’  (NOT ‘s’!)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char x [0]</td>
<td>100</td>
<td>‘t’</td>
</tr>
<tr>
<td>char x [1]</td>
<td>101</td>
<td>‘e’</td>
</tr>
<tr>
<td>char x [2]</td>
<td>102</td>
<td>‘s’</td>
</tr>
<tr>
<td>char x [3]</td>
<td>103</td>
<td>‘t’</td>
</tr>
<tr>
<td>char x [4]</td>
<td>104</td>
<td>‘\0’</td>
</tr>
</tbody>
</table>

For loop that iterates from 0 to COUNT-1.
Memorize it!
Pointers and Arrays in C

• An array name by itself is an address, or pointer in C.

• When an array is declared, the compiler allocates sufficient space beginning with some base address to accommodate every element in the array.

• The base address of the array is the address of the first element in the array (index position 0).

  – Example: int num[10];
    &num[0] is the same as num
Pointers and Arrays in C

• Suppose we define the following array and pointer:

\[
\text{int } a[100]; \quad \text{int* } \text{ptr};
\]

Assume that the system allocates memory at addresses 400, 404, 408, ..., 796 to the array.

\[
\text{int values are allocated 32 bits = 4 bytes.}
\]

  – The two statements: \( \text{ptr} = a; \) and \( \text{ptr} = \&a[0]; \) are equivalent and would assign the value of 400 to \( \text{ptr} \).

• Pointer arithmetic provides an alternative to array indexing in C.

  – The two statements: \( \text{ptr} = a + 1; \) and \( \text{ptr} = \&a[1]; \) are equivalent and would assign the value of 404 to \( \text{ptr} \).
Pointers and Arrays in C

• Assuming the elements of the array of integers have been assigned values, the following code would sum the elements of the array:

```c
int sum = 0;
for (ptr = a; ptr < &a[100]; ++ptr)
    sum += *ptr;
```

• Here is another way to sum the array:

```c
int sum = 0;
for (i = 0; i < 100; ++i)
    sum += *(a + i);
```

`a[b]` is just syntactic sugar for `*(a + b)`
Strings

• Series of characters treated as a single unit
• Can include letters, digits, and certain special characters (*, /, $)
• String literal (string constant) - written in double quotes
  – "Hello"
• Strings are arrays of characters (type char[])
• String literals are implicitly terminated by a '\0'.
• Each character is represented in numerical code called ASCII code.
• Example:
  – char greeting[] = “Hello”;
  – size of greeting is 6 (length of “Hello” + 1 for '\0').
  – address of the above string can be expressed in two ways:
    • &greeting[0]
    • greeting
## ASCII code

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Html ; Chr</th>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Html ; Chr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>NUL</td>
<td>32</td>
<td>20</td>
<td>040</td>
<td>&amp; #32 ; Space</td>
<td>64</td>
<td>40</td>
<td>100</td>
<td>&amp; #64 ; ß</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td>SOH</td>
<td>33</td>
<td>21</td>
<td>041</td>
<td>&amp; #33 ; !</td>
<td>65</td>
<td>41</td>
<td>101</td>
<td>&amp; #65 ; A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>002</td>
<td>STX</td>
<td>34</td>
<td>22</td>
<td>042</td>
<td>&amp; #34 ; &quot;</td>
<td>66</td>
<td>42</td>
<td>102</td>
<td>&amp; #66 ; B</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>003</td>
<td>ETX</td>
<td>35</td>
<td>23</td>
<td>043</td>
<td>&amp; #35 ; #</td>
<td>67</td>
<td>43</td>
<td>103</td>
<td>&amp; #67 ; C</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>004</td>
<td>EOT</td>
<td>36</td>
<td>24</td>
<td>044</td>
<td>&amp; #36 ; $</td>
<td>68</td>
<td>44</td>
<td>104</td>
<td>&amp; #68 ; D</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>005</td>
<td>ENQ</td>
<td>37</td>
<td>25</td>
<td>045</td>
<td>&amp; #37 ; %</td>
<td>69</td>
<td>45</td>
<td>105</td>
<td>&amp; #69 ; E</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>006</td>
<td>ACK</td>
<td>38</td>
<td>26</td>
<td>046</td>
<td>&amp; #38 ; &amp;</td>
<td>70</td>
<td>46</td>
<td>106</td>
<td>&amp; #70 ; F</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>007</td>
<td>BEL</td>
<td>39</td>
<td>27</td>
<td>047</td>
<td>&amp; #39 ; '</td>
<td>71</td>
<td>47</td>
<td>107</td>
<td>&amp; #71 ; G</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>010</td>
<td>BS</td>
<td>40</td>
<td>28</td>
<td>050</td>
<td>&amp; #40 ; (</td>
<td>72</td>
<td>48</td>
<td>110</td>
<td>&amp; #72 ; H</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>011</td>
<td>TAB</td>
<td>41</td>
<td>29</td>
<td>051</td>
<td>&amp; #41 ; )</td>
<td>73</td>
<td>49</td>
<td>111</td>
<td>&amp; #73 ; I</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>012</td>
<td>LF</td>
<td>42</td>
<td>2A</td>
<td>052</td>
<td>&amp; #42 ; *</td>
<td>74</td>
<td>4A</td>
<td>112</td>
<td>&amp; #74 ; J</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>013</td>
<td>VT</td>
<td>43</td>
<td>2B</td>
<td>053</td>
<td>&amp; #43 ; +</td>
<td>75</td>
<td>4B</td>
<td>113</td>
<td>&amp; #75 ; K</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>014</td>
<td>FF</td>
<td>44</td>
<td>2C</td>
<td>054</td>
<td>&amp; #44 ; ,</td>
<td>76</td>
<td>4C</td>
<td>114</td>
<td>&amp; #76 ; L</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>015</td>
<td>CR</td>
<td>45</td>
<td>2D</td>
<td>055</td>
<td>&amp; #45 ; -</td>
<td>77</td>
<td>4D</td>
<td>115</td>
<td>&amp; #77 ; M</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>016</td>
<td>SO</td>
<td>46</td>
<td>2E</td>
<td>056</td>
<td>&amp; #46 ; .</td>
<td>78</td>
<td>4E</td>
<td>116</td>
<td>&amp; #78 ; N</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>017</td>
<td>SI</td>
<td>47</td>
<td>2F</td>
<td>057</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>117</td>
<td>&amp; #79 ; O</td>
</tr>
<tr>
<td>16</td>
<td>G</td>
<td>020</td>
<td>DLE</td>
<td>48</td>
<td>30</td>
<td>060</td>
<td>&amp; #48 ; 0</td>
<td>80</td>
<td>50</td>
<td>120</td>
<td>&amp; #80 ; P</td>
</tr>
<tr>
<td>17</td>
<td>H</td>
<td>021</td>
<td>DC1</td>
<td>49</td>
<td>31</td>
<td>061</td>
<td>&amp; #49 ; 1</td>
<td>81</td>
<td>51</td>
<td>121</td>
<td>&amp; #81 ; Q</td>
</tr>
<tr>
<td>18</td>
<td>I</td>
<td>022</td>
<td>DC2</td>
<td>50</td>
<td>32</td>
<td>062</td>
<td>&amp; #50 ; 2</td>
<td>82</td>
<td>52</td>
<td>122</td>
<td>&amp; #82 ; R</td>
</tr>
<tr>
<td>19</td>
<td>J</td>
<td>023</td>
<td>DC3</td>
<td>51</td>
<td>33</td>
<td>063</td>
<td>&amp; #51 ; 3</td>
<td>83</td>
<td>53</td>
<td>123</td>
<td>&amp; #83 ; S</td>
</tr>
<tr>
<td>20</td>
<td>K</td>
<td>024</td>
<td>DC4</td>
<td>52</td>
<td>34</td>
<td>064</td>
<td>&amp; #52 ; 4</td>
<td>84</td>
<td>54</td>
<td>124</td>
<td>&amp; #84 ; T</td>
</tr>
<tr>
<td>21</td>
<td>L</td>
<td>025</td>
<td>NAK</td>
<td>53</td>
<td>35</td>
<td>065</td>
<td>&amp; #53 ; 5</td>
<td>85</td>
<td>55</td>
<td>125</td>
<td>&amp; #85 ; U</td>
</tr>
<tr>
<td>22</td>
<td>M</td>
<td>026</td>
<td>SYN</td>
<td>54</td>
<td>36</td>
<td>066</td>
<td>&amp; #54 ; 6</td>
<td>86</td>
<td>56</td>
<td>126</td>
<td>&amp; #86 ; V</td>
</tr>
<tr>
<td>23</td>
<td>N</td>
<td>027</td>
<td>ETB</td>
<td>55</td>
<td>37</td>
<td>067</td>
<td>&amp; #55 ; 7</td>
<td>87</td>
<td>57</td>
<td>127</td>
<td>&amp; #87 ; W</td>
</tr>
<tr>
<td>24</td>
<td>O</td>
<td>030</td>
<td>CAN</td>
<td>56</td>
<td>38</td>
<td>068</td>
<td>&amp; #56 ; 8</td>
<td>88</td>
<td>58</td>
<td>130</td>
<td>&amp; #88 ; X</td>
</tr>
<tr>
<td>25</td>
<td>P</td>
<td>031</td>
<td>EM</td>
<td>57</td>
<td>39</td>
<td>071</td>
<td>&amp; #57 ; 9</td>
<td>89</td>
<td>59</td>
<td>131</td>
<td>&amp; #89 ; Y</td>
</tr>
<tr>
<td>26</td>
<td>Q</td>
<td>032</td>
<td>SUB</td>
<td>58</td>
<td>3A</td>
<td>072</td>
<td>&amp; #58 ; :</td>
<td>90</td>
<td>5A</td>
<td>132</td>
<td>&amp; #90 ; Z</td>
</tr>
<tr>
<td>27</td>
<td>R</td>
<td>033</td>
<td>ESC</td>
<td>59</td>
<td>3B</td>
<td>073</td>
<td>&amp; #59 ; ;</td>
<td>91</td>
<td>5B</td>
<td>133</td>
<td>&amp; #91 ; [</td>
</tr>
<tr>
<td>28</td>
<td>S</td>
<td>034</td>
<td>FS</td>
<td>60</td>
<td>3C</td>
<td>074</td>
<td>&amp; #60 ; &lt;</td>
<td>92</td>
<td>5C</td>
<td>134</td>
<td>&amp; #92 ; \</td>
</tr>
<tr>
<td>29</td>
<td>T</td>
<td>035</td>
<td>GS</td>
<td>61</td>
<td>3D</td>
<td>075</td>
<td>&amp; #61 ; =</td>
<td>93</td>
<td>5D</td>
<td>135</td>
<td>&amp; #93 ; ]</td>
</tr>
<tr>
<td>30</td>
<td>U</td>
<td>036</td>
<td>RS</td>
<td>62</td>
<td>3E</td>
<td>076</td>
<td>&amp; #62 ; &gt;</td>
<td>94</td>
<td>5E</td>
<td>136</td>
<td>&amp; #94 ; ^</td>
</tr>
<tr>
<td>31</td>
<td>V</td>
<td>037</td>
<td>US</td>
<td>63</td>
<td>3F</td>
<td>077</td>
<td>&amp; #63 ; _</td>
<td>95</td>
<td>5F</td>
<td>137</td>
<td>&amp; #95 ; `</td>
</tr>
</tbody>
</table>

Source: www.LookupTables.com
Strings

- **String declarations**
  - Declare as a character array or a variable of type `char *`
    ```
    char color[] = "blue";
    char *colorPtr = "blue";
    ```
  - Remember that strings represented as character arrays end with `\0`
    - `color` has 5 elements

- **Inputting strings**
  - Use `scanf`
    ```
    scanf("%s", word);
    ```
    - Copies input into `word[]`, which does not need `&` (because a string is a pointer)
  - Remember to leave space for `\0`
## Character Handling Library

- In `<ctype.h>`

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int isdigit( int c )</code></td>
<td>Returns <strong>true</strong> if <code>c</code> is a digit and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td><code>int isalpha( int c )</code></td>
<td>Returns <strong>true</strong> if <code>c</code> is a letter and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td><code>int isalnum( int c )</code></td>
<td>Returns <strong>true</strong> if <code>c</code> is a digit or a letter and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td><code>int isxdigit( int c )</code></td>
<td>Returns <strong>true</strong> if <code>c</code> is a hexadecimal digit character and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td><code>int islower( int c )</code></td>
<td>Returns <strong>true</strong> if <code>c</code> is a lowercase letter and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td><code>int isupper( int c )</code></td>
<td>Returns <strong>true</strong> if <code>c</code> is an uppercase letter; <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td><code>int tolower( int c )</code></td>
<td>If <code>c</code> is an uppercase letter, <code>tolower</code> returns <code>c</code> as a lowercase letter. Otherwise, <code>tolower</code> returns the argument unchanged.</td>
</tr>
<tr>
<td><code>int toupper( int c )</code></td>
<td>If <code>c</code> is a lowercase letter, <code>toupper</code> returns <code>c</code> as an uppercase letter. Otherwise, <code>toupper</code> returns the argument unchanged.</td>
</tr>
<tr>
<td><code>int isspace( int c )</code></td>
<td>Returns <strong>true</strong> if <code>c</code> is a white-space character—newline (<code>'\n'</code>), space (<code>' '</code>), form feed (<code>'\f'</code>), carriage return (<code>'\r'</code>), horizontal tab (<code>'\t'</code>), or vertical tab (<code>'\v'</code>)—and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td><code>int iscntrl( int c )</code></td>
<td>Returns <strong>true</strong> if <code>c</code> is a control character and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td><code>int ispunct( int c )</code></td>
<td>Returns <strong>true</strong> if <code>c</code> is a printing character other than a space, a digit, or a letter and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td><code>int isprint( int c )</code></td>
<td>Returns <strong>true</strong> value if <code>c</code> is a printing character including space (<code>' '</code>) and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td><code>int isgraph( int c )</code></td>
<td>Returns <strong>true</strong> if <code>c</code> is a printing character other than space (<code>' '</code>) and <strong>false</strong> otherwise.</td>
</tr>
</tbody>
</table>

Each function receives a character (an `int`) or **EOF** as an argument
String Conversion Functions

- Conversion functions
  - In `<stdlib.h>` (general utilities library)
  - Convert strings of digits to integer and floating-point values

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double atof( const char *nPtr )</td>
<td>Converts the string nPtr to double.</td>
</tr>
<tr>
<td>int atoi( const char *nPtr )</td>
<td>Converts the string nPtr to int.</td>
</tr>
<tr>
<td>long atol( const char *nPtr )</td>
<td>Converts the string nPtr to long int.</td>
</tr>
<tr>
<td>double strtod( const char *nPtr, char **endPtr )</td>
<td>Converts the string nPtr to double.</td>
</tr>
<tr>
<td>long strtol( const char *nPtr, char **endPtr, int base )</td>
<td>Converts the string nPtr to long.</td>
</tr>
<tr>
<td>unsigned long strtoul( const char *nPtr, char **endPtr, int base )</td>
<td>Converts the string nPtr to unsigned long.</td>
</tr>
</tbody>
</table>
String Manipulation Functions

• In `<string.h>`

• String handling library has functions to
  – Manipulate string data
  – Search strings
  – Determine string length

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *strcpy( char *s1, const char *s2 )</td>
<td>Copies string s2 into array s1. The value of s1 is returned.</td>
</tr>
<tr>
<td>char *strncpy( char *s1, const char *s2, size_t n )</td>
<td>Copies at most n characters of string s2 into array s1. The value of s1 is returned.</td>
</tr>
<tr>
<td>char *strcat( char *s1, const char *s2 )</td>
<td>Appends string s2 to array s1. The first character of s2 overwrites the terminating null character of s1. The value of s1 is returned.</td>
</tr>
<tr>
<td>char *strncat( char *s1, const char *s2, size_t n )</td>
<td>Appends at most n characters of string s2 to array s1. The first character of s2 overwrites the terminating null character of s1. The value of s1 is returned.</td>
</tr>
</tbody>
</table>
String Manipulation Functions

int strcmp ( const char * str1,
            const char * str2 )

return value | indicates
--|---
<0 | the first character that does not match has a lower value in `ptr1` than in `ptr2`
0 | the contents of both strings are equal
>0 | the first character that does not match has a greater value in `ptr1` than in `ptr2`