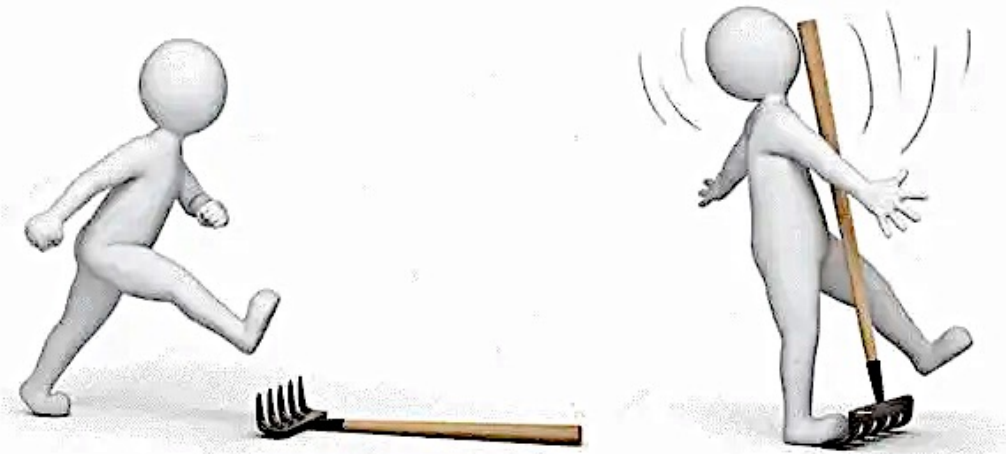
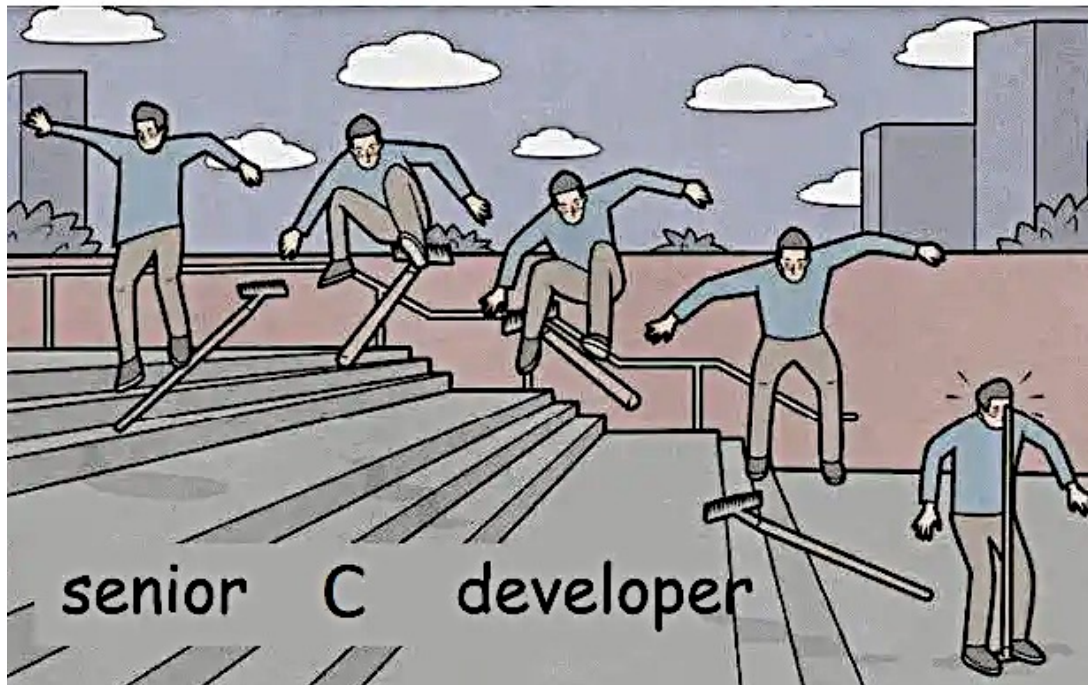
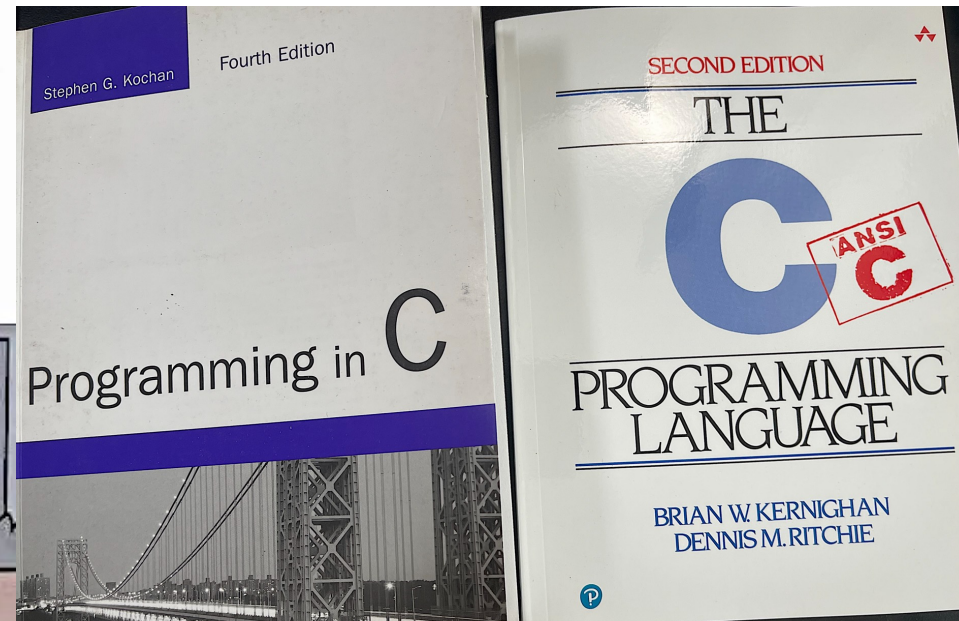


Lecture 14: C part VII



junior C developer

Suggested Literature



senior C developer

Why function pointers? Scientific Library

Excellent free library for scientific computing



GNU Operating System

Supported by the [Free Software Foundation](#)

[PHILOSOPHY](#)

[LICENSES](#)

[EDUCATION](#)

[SOFTWARE](#)

[DISTROS](#)

[DOCS](#)

[MALWARE](#)

[HELP GNU](#)

GSL - GNU Scientific Library

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License.

The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.

Unlike the licenses of proprietary numerical libraries the license of GSL does not restrict scientific cooperation. It allows you to share your programs freely with others.

Current stable version: **GSL-2.7**

The GSL Gnu Scientific Library

Integration: adaptive

How to characterize the integrand (one variable)?

With **GSL FUNCTION** struct

```
struct gsl_function_struct  
{  
    double (* function) (double x, void * params);  
    void * params;  
};  
  
typedef struct gsl_function_struct gsl_function ;
```



The **void* params** allows you to send constants to the integrand, for example, $f(x) = ax + b$ with a, b constants

GSL

Integration: adaptive

Functions have addresses

Therefore, there is a
function pointer

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f (double x, void* params)
{
    double alpha = *(double *) params;
    return log(alpha*x) / sqrt(x);
}

int main (void)
{
    const size_t max_num_subdiv = 1000;
    // adaptive integration need memory allocation of a
    // special type of struct to hold intermediate values
    gsl_integration_workspace* w
        = gsl_integration_workspace_alloc (max_num_subdiv);

    double result, error, alpha = 1.0;
    gsl_function F;
    F.function = &f;
    F.params = &alpha; // automatic cast to void*

    const double int_min = 0.0, int_max = 1.0;
    const double rel_error = 1e-7, abs_error = 0.0;

    gsl_integration_qags (&F, int_min, int_max, abs_error,
        rel_error, max_num_subdiv, w, &result, &error);

    printf ("result          = % .18f\n", result);
    printf ("exact result      = % .18f\n", -4.0);
    printf ("estimated error    = % .18f\n", error);
    printf ("actual error       = % .18f\n", result - (-4.0));

    gsl_integration_workspace_free (w);
    return 0;
}
```

```
struct gsl_function_struct
{
    double (* function) (double x, void * params);
    void * params;
};

typedef struct gsl_function_struct gsl_function ;
```



Each function type has its
unique function pointer

```

#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f (double x, void* params)
{
    double alpha = *(double *) params;
    return log(alpha*x) / sqrt(x);
}

int main (void)
{
    const size_t max_num_subdiv = 1000;
    // adaptive integration need memory allocation of a
    // special type of struct to hold intermediate values
    gsl_integration_workspace* w
        = gsl_integration_workspace_alloc (max_num_subdiv);

    double result, error, alpha = 1.0;
    gsl_function F;
    F.function = &f;
    F.params = &alpha; // automatic cast to void*

    const double int_min = 0.0, int_max = 1.0;
    const double rel_error = 1e-7, abs_error = 0.0;

    gsl_integration_qags (&F, int_min, int_max, abs_error,
        rel_error, max_num_subdiv, w, &result, &error);

    printf ("result          = % .18f\n", result);
    printf ("exact result       = % .18f\n", -4.0);
    printf ("estimated error      = % .18f\n", error);
    printf ("actual error         = % .18f\n", result - (-4.0));

    gsl_integration_workspace_free (w);
    return 0;
}

```

Why void*? GSL

Integration: adaptive

Flexibility mandates that
the **params ptr** on

gsl_function to be a **void**

void pointer is the base of
generic programming in C

Any pointer can be casted
into the **void pointer**.

The converse is also true as
a **void pointer** can be
casted into any pointer

Why some GSL functions need workspaces?

To store intermediate results

```
gsl_integration_workspace *
gsl_integration_workspace_alloc (const size_t n)
{
    gsl_integration_workspace * w ;
    if (n == 0)
    {
        GSL_ERROR_VAL ("workspace length n must be positive integer",
                        GSL_EDOM, 0);
    }

    w = (gsl_integration_workspace *)
        malloc (sizeof (gsl_integration_workspace));
    if (w == 0)
    {
        GSL_ERROR_VAL ("failed to allocate space for workspace struct",
                        GSL_ENOMEM, 0);
    }

    w->alist = (double *) malloc (n * sizeof (double));
    if (w->alist == 0)
    {
        free (w);          /* exception in constructor, avoid memory leak */

        GSL_ERROR_VAL ("failed to allocate space for alist ranges",
                        GSL_ENOMEM, 0);
    }

    w->blist = (double *) malloc (n * sizeof (double));
    if (w->blist == 0)
    {
        free (w->alist);
        free (w);          /* exception in constructor, avoid memory leak */

        GSL_ERROR_VAL ("failed to allocate space for blist ranges",
                        GSL_ENOMEM, 0);
    }

    w->rlist = (double *) malloc (n * sizeof (double));
    if (w->rlist == 0)
    {
```

typedef struct **from GSL**

```
{
    size_t limit;      source
    size_t size;       code
    size_t nrmax;
    size_t i;
    size_t maximum_level;
    double *alist;
    double *blist;
    double *rlist;
    double *elist;
    size_t *order;
    size_t *level;
}
```

gsl_integration_workspace;

GSL

Integration: non-adaptive

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f (double x, void* params)
{
    double alpha = *(double *) params;
    return alpha*x*x;
}

int main (void)
{
    size_t num_subdiv;
    const double int_min = 0.0, int_max = 1.0;
    const double rel_error = 1e-3, abs_error = 0.0;
    double result, error, alpha = 1.0;

    gsl_function F;
    F.function = &f;
    F.params = &alpha; // automatic cast to void*

    gsl_integration_qng (&F, int_min, int_max, abs_error,
        rel_error, &result, &error, &num_subdiv);

    printf ("result          = % .18f\n", result);
    printf ("exact result      = % .18f\n", 1.0/3.0);
    printf ("estimated error    = % .18f\n", error);
    printf ("actual error       = % .18f\n", result - (1./3.));

    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc -O2 gsl3.c -o gsl3 -lgsl -lgslcblas -lm
whovian@phy504:~/host/C$ ./gsl3
result          = 0.333333333333333370
exact result    = 0.333333333333333315
estimated error = 0.0000000000000003701
actual error    = 0.000000000000000056
```

- **QNG**: weaker, non-adaptive but fast integrator for smooth functions

Flexibility mandates that params on **gsl_function** is a void pointer.

Source:

<https://www.gnu.org/software/gsl/doc/html/integration.html>

Strings in C

Deeper dive on
printf command in
C: formatting

float numbers
%e, %f, %g

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    const double a = 97.4583;

    // %e and %f default: six decimal places
    printf ("%f %e\n", a, a);

    // %g is weird. Up to 10^-4, %g = %f
    // Lower than 10^-4, %g = %e
    const double b = 0.001;
    const double c = 0.0001;
    const double d = 0.00001;
    printf ("%g %g %g\n", b, c, d);

    // %g is weird. Up to 10^5, %g = %f,
    // Higher than 10^5, %g = %e
    const double e = 10000;
    const double f = 100000;
    const double g = 1000000;
    printf ("%g %g %g\n", e, f, g);

    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc char2.c -o char2
whovian@phy504:~/host/C$ ./char2
97.458300 9.745830e+01
0.001 0.0001 1e-05
10000 100000 1e+06
```

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
    const double a = 97.4583;

    printf ("x=%7.2f\n", a);
    printf ("x=%07.2f\n", a);

    printf ("\n");

    printf ("x=%.2e\n", a);
    printf ("x=%.2e\n", a);

    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc char3.c -o char3
whovian@phy504:~/host/C$ ./char3
x= 97.46
x=0097.46

x=9.75e+01
x=9.75e+01
```

Strings in C

Deeper dive on
printf command in
C: formatting

float numbers
%e, %f, %g

Strings in C

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    const int a = 300;

    printf("a=%10d\n", a);
    printf("a=%010d\n", a);

    const int b = 45;
    printf("b=%5x\n", b);
    printf("b=%05X\n", b);

    // no specification for
    // binary :(
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc char5.c -o char5
whovian@phy504:~/host/C$ ./char5
a=          300
a=0000000300
b=      2d
b=0002D
```

Deeper dive on **printf**
command in C:
formatting

Integer: **%d**
Hexadecimal: **%x** (0-9a-f) and **%X** (0-9A-F)

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
    printf("Enter the length of the string: ");

    int n;
    scanf("%d", &n);

    char* str = malloc((n + 1)*sizeof(char));
    if (str == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        return 1;
    }

    printf("Enter the string: ");
    scanf("%s", str);

    printf("The entered string is: %s\n", str);

    free(str);
    return 0;
}
```

Strings in C

scanf in C:
user defines
values on
the terminal

```
whovian@phy504:~/host/C$ gcc char1.c -o char1
whovian@phy504:~/host/C$ ./char1
Enter the length of the string: 10
Enter the string: Vivian
The entered string is: Vivian
```

Strings in C

What if you want
to specify
width/precision of
float numbers at
runtime?

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
    const double a = 97.4583;

    int width, prec;
    printf("Enter width\n");
    scanf("%d", &width);
    printf("Enter precision\n");
    scanf("%d", &prec);

    printf("x=%*.*f\n", width, prec, a);
    printf("x=%0*.*f\n", width, prec, a);
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc char4.c -o char4
```

```
whovian@phy504:~/host/C$ ./char4
```

```
Enter width
```

```
10
```

```
Enter precision
```

```
3
```

```
x=    97.458
```

```
x=000097.458
```

Literals (in any language)

Literal (computer programming)

🌐 17 languages ▾

Article [Talk](#)

[Read](#) [Edit](#) [View history](#) [Tools](#) ▾

From Wikipedia, the free encyclopedia

In [computer science](#), a **literal** is a textual representation (notation) of a [value](#) as it is written in [source code](#).^{[1][2]} Almost all programming languages have notations for atomic values such as [integers](#), [floating-point numbers](#), and [strings](#), and usually for [Booleans](#) and [characters](#); some also have notations for [elements of enumerated types](#) and compound values such as [arrays](#), [records](#), and [objects](#). An [anonymous function](#) is a literal for the [function type](#).

In contrast to literals, [variables](#) or [constants](#) are symbols that can take on one of a class of fixed values, the constant being constrained not to change. Literals are often used to initialize variables; for example, in the following, 1 is an integer literal and the three letter string in "cat" is a string literal:

```
int a = 1;  
string s = "cat";
```



This is a string literal

There are two technical question: Where are literals stored in memory? Can I access that memory via pointers?

Literals (in any language)

Literal (computer programming)

🌐 17 languages ▾

Article [Talk](#)

[Read](#) [Edit](#) [View history](#) [Tools](#) ▾

From Wikipedia, the free encyclopedia

In [computer science](#), a **literal** is a textual representation (notation) of a [value](#) as it is written in [source code](#).^{[1][2]} Almost all programming languages have notations for atomic values such as [integers](#), [floating-point numbers](#), and [strings](#), and usually for [Booleans](#) and [characters](#); some also have notations for [elements of enumerated types](#) and compound values such as [arrays](#), [records](#), and [objects](#). An [anonymous function](#) is a literal for the [function type](#).

In contrast to literals, [variables](#) or [constants](#) are symbols that can take on one of a class of fixed values, the constant being constrained not to change. Literals are often used to initialize variables; for example, in the following, 1 is an integer literal and the three letter string in "cat" is a string literal:

```
int a = 1;  
string s = "cat";
```



This is a string literal

In C++, this understand “[lvalues](#)” and “[rvalues](#)” will be crucial. Things that can be on the left side vs right side of an assignment

Literals (in C)

What is a literal string & char array in C?

C has 2 kinds of *literals*: *string literals* and *compound literals*. Both are unnamed and both can have their address taken. *string literals* can have more than 1 *null character* in them.

In the C library, a *string* is characters up to and including the first *null character*. So a *string* always has one and only one *null character*, else it is not a *string*. A *string* may be `char`, `signed char`, `unsigned char`.

```
//          v---v string literal 6 char long
char *s1 = "hello";
char *s2 = "hello\0world";
//          ^-----^ string literal 12 char long

char **s3 = &"hello"; // valid

//          v-----v compound literal
int *p1 = (int []){2, 4};
int **p2 = &(int []){2, 4}; // valid
```

C specifies the following as constants, not literals, like `123`, `'x'` and `456.7`. These *constants* can not have their address taken.

```
int *p3 = &7; // not valid
```

C++ and C differ in many of these regards.

A `char` array is an *array* of `char`. An array may consist of many *null characters*.

```
char a1[3];           // `a1` is a char array size 3
char a2[3] = "123";   // `a2` is a char array size 3 with 0 null characters
char a3[4] = "456";   // `a3` is a char array size 4
char a4[] = "789";     // `a4` is a char array size 4
char a5[4] = { 0 };    // `a5` is a char array size 4, all null characters
```

The following `t*` are *not* `char` arrays, but pointers to `char`.

```
char *t1;
char *t2 = "123";
int *t3 = (char){'x'};
```

There are two
technical questions:

Where are literals
stored in memory?

Can I access that
memory via pointers?

<https://stackoverflow.com/a/50125548/2472169>

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
```

```
// string literal is weird. Not understanding
// string literal is dangerous. "string" (with
// double quotes) is a string literal and cannot
// be modified. HOWEVER...
```

```
char s[6] = "hello"; // this copy string literal
                     // to the array. So ok to
                     // modify the array
```

```
// We need 6! Why? Because every string literal
// ends with the '\0' (NULL) character
printf ("s = %s\n", s);
```

```
// cannot use double quotes here
// weird: double quotes = string literal (X + '\0')
// string literal "X" would
// return char*
// Single quotes = just a char
s[0] = 'X'; // ok to do it
```

```
printf ("s = %s\n", s);
return 0;
```

```
}
```

```
whovian@phy504:~/host/C$ nano char6.c
```

```
whovian@phy504:~/host/C$ gcc -Wall -pedantic char6.c -o char6
```

```
whovian@phy504:~/host/C$ ./char6
```

```
s = hello
```

```
s = Xello
```

Strings in C

Introduction to

char* and
string literals

s and **&s** are
the same for

char arrays

(char array just alias to ptr)

Strings in C

Introduction to

char* and
string literals

s and **&s** are
the same for

char arrays

(char array just alias to ptr)

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    // string literal is weird. Not understanding
    // string literal is dangerous. "string" (with
    // double quotes) is a string literal and cannot
    // be modified. HOWEVER...
    char s[6] = "hello"; // this copy string literal
                        // to the array. So ok to
                        // modify the array

    // We need 6! Why? Because every string literal
    // ends with the '\0' (NULL) character
    printf ("s = %s\n", s);

    // equivalent statement
    char s2[6] = {'h', 'e', 'l', 'l', 'o', '\0'};

    printf ("s = %s\n", s2);

    return 0;
}
```

```
whovian@phy504:~/host/C$ nano char7.c
whovian@phy504:~/host/C$ gcc -Wall -pedantic char7.c -o char7
whovian@phy504:~/host/C$ ./char7
s = hello
s = hello
```

Strings in C

Introduction to **char*** and **string literals**

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    // string literal is weird. Not understanding
    // string literal is dangerous. "string" (with
    // double quotes) is a string literal and cannot
    // be modified. HOWEVER...
    char s[] = "hello"; // this copy string literal
                        // to the array. So ok to
                        // modify the array

    // When init the array with a string literal,
    // you can use char[]. C will figure out that
    // the size is 6 (remember the null char \0)
    printf ("s = %s. Size = %ld\n", s, sizeof(s));

    // PS: sizeof return long unsigned int (so %ld)
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc -Wall -pedantic char8.c -o char8
whovian@phy504:~/host/C$ ./char8
s = hello. Size = 6
```

Strings in C

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    // string literal is weird. Not understanding
    // string literal is dangerous. "string" (with
    // double quotes) is a string literal and cannot
    // be modified. HOWEVER...

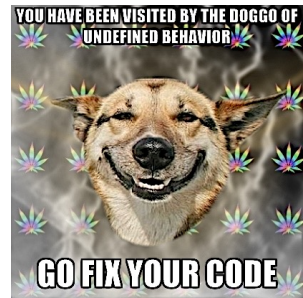
    char* s = "hello"; // string literal lives in
                       // read-only part of memory

    // so modifying a char* that points to a string
    // literal is UNDEFINED BEHAVIOR (not impossible
    // because it is not const char*, but it is UNDEFINED
    printf("s = %s\n", s);
    return 0;
}
```



```
whovian@phy504:~/host/C$ nano char9.c
whovian@phy504:~/host/C$ gcc -Wall -pedantic char9.c -o char9
whovian@phy504:~/host/C$ ./char9
s = hello
```

Strings in C



```
#include <string.h>
int main()
{
    char *str1 = "string Literal";
    const char *str2 = "string Literal";

    char source[] = "Sample string";

    // Why I really support const correctness

    strcpy(str1, source);    // No warning or error, just
                           // Undefined Behavior
    strcpy(str2, source);    // Compiler issues a warning

    return 0;
}
```

The first arg in
**strcpy(dest,
source)**
is the destination.
Both arguments in
strcpy are **char***
**strcpy = STRing
CoPY.**

```
whovian@phy504:~/host/C$ nano char11.c
whovian@phy504:~/host/C$ gcc -Wall -pedantic char11.c -o char11
char11.c: In function 'main':
char11.c:13:12: warning: passing argument 1 of 'strcpy' discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]
   13 |     strcpy(str2, source);    // Compiler issues a warning
      |           ~~~~~^
In file included from char11.c:1:
/usr/include/string.h:122:39: note: expected 'char * restrict' but argument is of type 'const char *'
   122 | extern char *strcpy (char *__restrict __dest, const char *__restrict __src)
      |                      ~~~~~^~~~~~
```

Strings in C

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

// Source: 21st century C. Author: Ben Klemens

int main()
{
    // strip will copy the str literal to a proper char[]
    char *str1 = strdup("string Literal is const");
    char source[] = "Sample string";

    strcpy(str1, source); // Not a problem anymore!
    printf("%s\n", str1);
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc char11v2.c -o char11v2
whovian@phy504:~/host/C$ ./char11v2
Sample string
```

strdup = STRing DuPlicate

An interesting way to avoid this problem is never to directly assign a string literal to **char***.

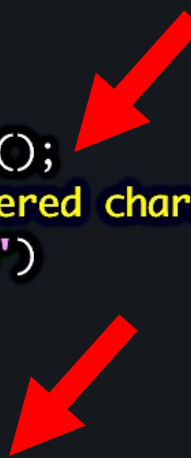
Use **char* strdup(const char* s)** to copy the string literal to **char***

```
#include <stdio.h>
#include <ctype.h>
```

```
int main()
{
    char c;
    printf("Enter some character. Enter $ to exit...\n");

    for(;;)
    {
        c = getchar();
        printf("Entered character is: ");
        if (c == '$')
        {
            break;
        }
        putchar(c);
        printf("\n");
    }

    return 0;
}
```



Strings in C

getchar/putchar

read/write a single
char from terminal

```
whovian@phy504:~/host/C$ gcc -Wall -pedantic char10.c -o char10
whovian@phy504:~/host/C$ ./char10
Enter some character. Enter $ to exit...
Vivian Miranda$
Entered character is: V
Entered character is: i
Entered character is: v
Entered character is: i
Entered character is: a
Entered character is: n
Entered character is:
Entered character is: M
Entered character is: i
Entered character is: r
Entered character is: a
Entered character is: n
Entered character is: d
Entered character is: a
```

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
    char* buffer;
    size_t bufsize = 32; // long unsigned int
    size_t chars; // long unsigned int

    buffer = (char*) malloc(bufsize*sizeof(char));
    if (buffer == NULL)
    {
        perror("Unable to allocate buffer");
        exit(1);
    }

    printf("Type something: ");
    chars = getline(&buffer, &bufsize, stdin);
    printf("%zu characters were read. \n", chars);
    printf("You type: %s \n", buffer);

    return 0;
}
```

Strings in C

getline allows
you to read
entire lines (up
to **\n**)

Be careful with
buffer
overflow!

```
whovian@phy504:~/host$ gcc getline.c -o getline
whovian@phy504:~/host$ ./getline
Type something: Interesting
12 characters were read.
You type: Interesting
```

Strings in C

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char* buffer = NULL;
    size_t bufsz = 0;
    size_t chars;

    printf("Type something: ");
    chars = getline(&buffer, &bufsz, stdin);
    printf("%zu characters were read. \n", chars);
    printf("You type: %s \n", buffer);

    return 0;
}
```

If you provide a **NULL** pointer, C will allocate sufficient storage for your input

malloc only relevant when dealing with multiple inputs (ex: double pointer for multiple lines)

```
whovian@phy504:~/host$ gcc getline2.c -o getline2
whovian@phy504:~/host$ ./getline2
Type something: Vivian Miranda
15 characters were read.
You type: Vivian Miranda

whovian@phy504:~/host$
```

Working with files

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

```
int main (void)
```

```
{
```

```
    int c;
```

```
    while ((c = getchar ()) != EOF)
```

```
    {
```

```
        putchar (c);
```

```
    }
```

```
    return 0;
```

```
}
```

C has a special keyword to denote **end-of-file: EOF**

```
whovian@phy504:~/host$ gcc eof.c -o eof
whovian@phy504:~/host$ nano eofinput.txt
whovian@phy504:~/host$ ./eof < eofinput.txt
```

```
My name is Vivian
```

```
I live in NYC
```

```
I was born in Brazil
```

```
whovian@phy504:~/host$ more eofinput.txt
```

```
My name is Vivian
```

```
I live in NYC
```

```
I was born in Brazil
```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// reverse string, swaps src & dest each iteration up to '\n'
// Code source: https://stackoverflow.com/a/66132517/2472169
char* strrev (char* str)
{ // original string is not preserved
  if (!str)
  { // validate str not NULL
    printf ("error: invalid string\n");
    return NULL;
  }
  if (!*str)
  { // check empty string
    return str;
  }
  char* begin = str;
  char* end = begin + strcspn (str, "\n") - 1;

  while (end > begin)
  {
    char tmp = *end;
    *end-- = *begin;
    *begin++ = tmp;
  }
  return str;
}

int main()
{
  char* buffer = NULL;
  size_t bufsize = 0;
  size_t chars;
  chars = getline(&buffer, &bufsize, stdin);
  printf("You type in reverse: %s \n", strrev(buffer));
  return 0;
}

```

Strings in C

We can use **strrev** to reverse lines read via **getline/scanf**

strcspan(str1, str2) returns the length of the initial part of **str1** not containing any of the chars that are part of **str2**

There are many useful functions on **string.h** header
strrev = STRing REVerse.

strcspan = STRing Complementary SPaN.

```

whovian@phy504:~/host$ gcc getline3.c -o getline3
whovian@phy504:~/host$ ./getline3
Vivian Miranda
You type in reverse: adnariM naiviV

```

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

```
// reverse string, swaps src & dest each iteration up to '\n'
// Code source: https://stackoverflow.com/a/66132517/2472169
```

```
char* strrev(char* str)
{ // original string is not preserved
  if (!str)
  { // validate str not NULL
    printf("error: invalid string\n");
    return NULL;
  }
  if (!*str)
  { // check empty string
    return str;
  }
  char* begin = str;
  char* end = begin + strcspn(str, "\n") - 1;

  while (end > begin)
  {
    char tmp = *end;
    *end-- = *begin;
    *begin++ = tmp;
  }
  return str;
}

int main()
{
  char* buffer = NULL;
  size_t bufsize = 0;
  while (getline(&buffer, &bufsize, stdin) != EOF)
  {
    printf("You type in reverse: %s \n", strrev(buffer));
  }
  return 0;
}
```

Strings in C

The use of **strcspan** is quite important when dealing with multiple lines (we want to reverse them separately)

```
whovian@phy504:~/host$ gcc getline4.c -o getline4
whovian@phy504:~/host$ ./getline4 < eofinput.txt
You type in reverse: naiviV si eman yM
```

```
You type in reverse: CYN ni evil I
```

```
You type in reverse: lizarB ni nrob saw I
```

```
whovian@phy504:~/host$
```

strcspan = STRing
Complementary SPaN.

Strings in C

There are useful functions on **string.h**

strpbrk finds the first character in the string **str1** that matches any character specified in **str2**

strpbrk = STRing Pointer BReak

```
#include <stdio.h>
#include <string.h>
// source: https://www.geeksforgeeks.org/strpbrk-in-c/
```

```
int main()
```

```
{
    char* str1 = "geeksforgeeks";
```

```
    char* str2 = "app";
```

```
    char* str3 = "kite";
```

```

    char* r = strpbrk(str1, str2);
```

```
    if (r != 0)
```

```
    {
        printf("First matching character: %c\n", *r);
    }
```

```
    else
```

```
    {
        printf("Character not found\n");
    }
```

```

    char* t = strpbrk(str1, str3);
```

```
    if (t != 0)
```

```
    {
        printf("First matching character: %c\n", *t);
    }
```

```
    else
```

```
    {
        printf("Character not found\n");
```

```
    }
    return 0;
```

```
whovian@phy504:~/host$ gcc char12.c -o char12
```

```
whovian@phy504:~/host$ ./char12
```

```
Character not found
```

```
First matching character: e
```

Strings in C

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

// source: https://www.geeksforgeeks.org/strstr-in-cpp/

int main()
{
    char* s1 = "GeeksforGeeks";
    char* s2 = "for";
    char* p = strstr(s1, s2);

    if (p != NULL) // equivalent to if(p)
    {
        printf("String found\n");
        printf("First occurrence of string '%s' in '%s' is '%s' \n",
            s2, s1, p);
    }
    else
    {
        printf("String not found\n");
    }
    return 0;
}
```

```
whovian@phy504:~/host$ gcc char13.c -o char13
whovian@phy504:~/host$ ./char13
String found
First occurrence of string 'for' in 'GeeksforGeeks' is 'forGeeks'
```

What if we want to find a string?

strstr returns a pointer to the first occurrence of string **str2** in **str1**

strstr = STRing
STRing

```

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

// char *strncat(char* dest,
//               const char* src)

// returns a pointer to the
// resulting string dest.

int main ()
{
    char str[80];
    strcpy (str, "these ");
    strcat (str, "strings ");
    strcat (str, "are ");
    strcat (str, "concatenated.");

    printf("Resulting string = \n |%s| \n", str);

    return 0;
}

```

Strings in C

How to concatenate
two strings?

Two options: **strcat** or
strncat

Pointer **dest** (destination)
should be large enough to
contain the concatenated
resulting string.

strcat: STRing CATenate

```

whovian@phy504:~/host$ gcc char14.c -o char14
whovian@phy504:~/host$ ./char14
Resulting string =
|these strings are concatenated.|

```

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

```
// char *strncat(char* dest,
//               const char* src,
//               size_t n)
```

```
// n = maximum num of chars
// to be appended.
```

```
// returns a pointer to the
// resulting string dest.
```

```
int main ()
{
    char str[80];
    strcpy (str, "these ");
    strncat (str, "strings ", 20);
    strncat (str, "are ", 20);
    strncat (str, "concatenated.", 5);

    printf("Resulting string = \n |%s| \n", str);

    return 0;
}
```



Strings in C

How to concatenate two strings?

Two options: **strcat** or **strncat**

Pointer **dest** (destination) should be large enough to contain the concatenated resulting string.

strncat: STRing N-CATenate

```
whovian@phy504:~/host$ gcc char15.c -o char15
whovian@phy504:~/host$ ./char15
Resulting string =
|these strings are conca|
```



```

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

// int strcmp(const char *leftStr,
//            const char *rightStr);

// code source: geeksforgeeks
// return < 0, 0, > 0
// > 0 -> left > right
// < 0 -> left < right

int main()
{
    char* leftStr  = "g f g";
    char* rightStr = "g f g";

    const int res =
        strcmp(leftStr, rightStr);

    if (res==0)
    {
        printf("Strings are equal\n");
    }
    else
    {
        printf("Strings are unequal\n");
    }

    printf("Value returned by strcmp() is: %d\n" , res);

    return 0;
}

```

```

whovian@phy504:~/host$ gcc char16.c -o char16
whovian@phy504:~/host$ ./char16
Strings are equal
Value returned by strcmp() is: 0

```

Strings in C

How to compare two strings (dictionary)?

Two ways: **strcmp** (locale independent) and **strcoll** (advanced as it considers language locale)

strcmp: STRing CoMPare

strncoll: STRing COLLate

(meaning to arrange data in a specific order)

Strings in C

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

int main()
{
    setlocale(LC_ALL, "en_US.UTF-8");

    char* leftStr = "é";
    char* rightStr = "f";

    const int res = strcmp(leftStr, rightStr);
    const int res2 = strcoll(leftStr, rightStr);

    printf(
        "Value returned by strcmp() is: %d\n", res);

    printf(
        "Value returned by strcoll() is: %d\n", res2);

    return 0;
}
```

Need to set the locale



How to compare two strings (dictionary)?

Two ways: **strcmp** (locale independent) and **strcoll** (advanced as it considers language locale)

Locale here? Important for comparison involving accented letters

strcmp: STRing CoMPare

strncoll: STRing COLLate

(meaning to arrange data in a specific order)

```
whovian@phy504:~/host$ gcc char17.c -o char17
whovian@phy504:~/host$ ./char17
Value returned by strcmp() is: 93
Value returned by strcoll() is: -68
```



Strings in C (Copying)

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

```
// memmove copies bytes (it does not
// care if the bytes are strings or not)
// void* memmove (void* destination,
//               const void* source,
//               size_t num );
```

```
// source: geeksforgeeks
```

```
int main()
```

```
{
    char str1[100] = "Geeks";
    char str2[10]  = "Quiz";
```

```
    printf("str1 before memmove %s \n", str1);
```

```
    // make sure that memory has enough mem
```

```
    printf("size of str1: %lu \n", sizeof(str1));
```

```
    printf("size of str2: %lu \n", sizeof(str2));
```

```
    memmove(str1, str2, sizeof(str2));
```

```
    printf("str1 after memmove %s \n", str1);
```

```
    return 0;
```

```
}
```

```
whovian@phy504:~/host$ gcc char19.c -o char19
whovian@phy504:~/host$ ./char19
str1 before memmove Geeks
size of str1: 100
size of str2: 10
str1 after memmove Quiz
```

There are important
differences between

memmove and **memcpy**

(imp difference when
strings overlap)

memmove:

MeMory MOVE

Strings in C (Copying)

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

```
// memmove copies bytes (it does not
// care if the bytes are strings or not)
// void* memmove (void* destination,
//               const void* source,
//               size_t num );
```

```
// source: geeksforgeeks
```

```
int main()
```

```
{
    char str1[100] = "Geeks";
    char str2[10]  = "Quiz";
```

```
whovian@phy504:~/host$ gcc char19.c -o char19
```

```
whovian@phy504:~/host$ ./char19
```

```
str1 before memmove Geeks
```

```
size of str1: 100
```

```
size of str2: 10
```

```
str1 after memmove Quiz
```

```
printf("str1 before memmove %s \n", str1);
```

```
// make sure that memory has enough mem
```

```
printf("size of str1: %lu \n", sizeof(str1));
```

```
printf("size of str2: %lu \n", sizeof(str2));
```

```
memmove(str1, str2, sizeof(str2));
```

```
printf("str1 after memmove %s \n", str1);
```

```
return 0;
```

```
}
```

Why **printf** only
prints “Quiz”?

memmove:

MeMory MOVE

memcpy: MeMory CoPY

Strings in C (Copying)

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

// memmove copies bytes (it does not
// care if the bytes are strings or not)
// void* memmove (void* destination,
//               const void* source,
//               size_t num );
```

```
// source: geeksforgeeks
```

```
int main()
```

```
{
    char str1[100] = "GeeksforGeeks";
    char str2[10]  = "Quiz";
```

```
    printf("str1 before memmove %s \n", str1);
```

```
    // make sure that memory has enough mem
```

```
    printf("size of str1: %lu \n", sizeof(str1));
```

```
    printf("size of str2: %lu \n", sizeof(str2));
```

```
    printf("str1 before memmove %s \n", str1);
```

```
    memmove(str1, str2, sizeof(str2));
```

```
    printf("str1 after memmove %s \n", str1);
```

```
    return 0;
```

```
}
```

```
whovian@phy504:~/host$ gcc char19.c -o char19
whovian@phy504:~/host$ ./char19
str1 before memmove GeeksforGeeks
size of str1: 100
size of str2: 10
str1 before memmove GeeksforGeeks
str1 after memmove Quiz
```

Why **printf** prints
“Quiz”?

Strings always end in the
null char. **%s** only prints
until the null char

String != char array

Strings in C (Copying)

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

```
// memmove copies bytes (it does not
// care if the bytes are strings or not)
// void* memmove (void* destination,
//               const void* source,
//               size_t num );
```

```
// source: geeksforgeeks
```

```
int main()
```

```
{
```

```
    char str1[100] = "GeeksforGeeks";
```

```
    char str2[10]  = "Quiz";
```

```
    printf("str1 before memmove %s \n", str1);
```

```
    // make sure that memory has enough mem
```

```
    printf("size of str1: %lu \n", sizeof(str1));
```

```
    printf("size of str2: %lu \n", sizeof(str2));
```

```
    memmove(str1, str2, sizeof(str2));
```

```
    printf("str1 after memmove %s \n", str1);
```

```
    printf("str1 with fwrite ");
```

```
    fwrite(str1, sizeof(char), 20, stdout);
```

```
    printf("\n");
```

```
    return 0;
```

```
}
```

```
whovian@phy504:~/host$ gcc char19v2.c -o char19v2
```

```
whovian@phy504:~/host$ ./char19v2
```

```
str1 before memmove GeeksforGeeks
```

```
size of str1: 100
```

```
size of str2: 10
```

```
str1 after memmove Quiz
```

```
str1 with fwrite Quizeks
```

Why **printf** prints “Quiz”?

Strings always end in the null char. **%s** only prints until the null char

String != char array

fwrite forces printing of the char array

Strings in C (Copying)

```
#include "stdio.h"
```

```
int main(void)
```

```
{
```

```
    int length = 5;
```

```
    char string[length];
```

```
    string[0] = 'A';
```

```
    string[1] = 'B';
```

```
    string[2] = '\0'; // null char
```

```
    string[3] = 'C';
```

```
    string[4] = 'D';
```

```
    printf("With printf: %.*s\n", length,  
          string);
```

```
    printf("With fwrite: ");
```

```
    fwrite(string, sizeof(char), length,  
           stdout);
```

```
    printf("\n");
```

```
    return 0;
```

```
}
```

```
whovian@phy504:~/host$ gcc char19v3.c -o char19v3
```

```
whovian@phy504:~/host$ ./char19v3
```

```
With printf: AB
```

```
With fwrite: ABCD
```

Strings always end in the null char. **%s** only prints until the null char

String != char array

fwrite forces printing of the char array

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

```
int main()
{
    char str[100] = "Learningisfun";

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

    // undefined behavior to use memcpy
    // memmove has an internal buffer, so
    // it is ok!
    memmove(str + 8, str, 15);
    printf("memmove overlap : %s \n", str);

    return 0;
}
```

```
whovian@phy504:~/host$ gcc char20.c -o char20
whovian@phy504:~/host$ ./char20
original str: Learningisfun
memmove overlap : LearningLearningisfun
```

Strings in C (Copying)

There are important differences between memmove and memcpy
(ex: when strings overlap)



Homework I

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

typedef struct
{
    int univ_id;
    int num_hw;
    int num_exams;
    double hwgrades[1000];
    double hwexams[1000];
} student;

// Implement the following functions

// double HwStudentMean(student x, const int num_hw);
// this function should check that the number student.num_hw > num_hw

// double ExamStudentMean(student x, const int num_exam)
// this function should check that the number of student.num_exams > num_exams

// double AverageSingleHw(const int num_students, student[], const int which_hw)

// double AverageSingleExame(const int num_students, student[], const int which_exam)

// create example showing these functions in use
```

Homework II

What is the output of the following program and why?
Does this program have a memory leak? Why?

```
#include <stdio.h>

int main()
{
    int *ptr1, *ptr2, arr[] = {10, 20, 30, 40, 50};

    ptr1 = &arr[0];
    ptr2 = &arr[3];

    printf("%d\n", ptr1[ptr2 - ptr1]);

    return 0;
}
```

Homework III

What is the output of the following program and why?
Does this program have a memory leak? Why?

```
#include <stdio.h>

int main()
{
    int *ptr, sum, arr[5] = {11, 21, 31, 41, 51};
    sum = 0;
    for(ptr = arr; ptr <= arr+4; ptr++)
    {
        --*ptr;
        sum += *ptr;
    }

    printf("Sum = %d\n", sum);

    return 0;
}
```

Homework IV

What is the output of the following program and why?

Does this program have a memory leak? Why?

```
#include <stdio.h>

int main()
{
    int a[] = {0, 0, 1, 2, 3}, b[] = {0, 0, 4, 5, 6};
    int *ptr1 = a, *ptr2 = b;

    while(*ptr1++ || *ptr2++);

    printf("%ld %ld\n", ptr1-a, ptr2-b);

    return 0;
}
```

Homework V

Write C program that sorts an integer array in ascending order using a single pointer and a pointer to a function without using qsort. Hint: use the bubble sort algorithm (https://en.wikipedia.org/wiki/Bubble_sort)

Additional Reading

- Strings in C: <https://www.geeksforgeeks.org/strings-in-c/>
- Literals in C++ and C:
<https://www.geeksforgeeks.org/literals-in-c-cpp-with-examples/>