

# **CS 211: Intro to Computer Architecture**

## ***7.2: C Generics and Library Functions***

**Minesh Patel**

Spring 2025 – Thursday 6 March

# Announcements: Exam Preparation

## 1. **Today** (*hopefully, or worst-case by tomorrow*):

- List of topics
- Practice question worksheet

## 2. **Next week:**

- No recitations, but extra office hours instead
- Part of Tuesday's lecture will go over the practice question worksheet

# Other Announcements

- This week:
  - **WA4 + WA5**: assigned through Canvas; due **next Tuesday @ 23:59**
    - **No late submissions!**
    - We will release solutions, so late submissions make no sense
  - **PA3**: released; due **Wednesday after Spring Break**
    - Encompasses pointers, arrays, and memory management
    - Please skim the doc and code ASAP so you know what you need to do

# Reference Material

- Today's lecture draws heavily from:
  - [CS 61C @ UC Berkeley](#) (Prof. Dan Garcia)

## And Various C and Linux Reference Materials

cppreference.com Create account Search

Page Discussion Standard revision: Diff View View source History

C

### C reference

C89, C95, C99, C11, C17, C23 | Compiler support C99, C23

<b>Language</b> <ul style="list-style-type: none"><li>Basic concepts</li><li>Keywords</li><li>Preprocessor</li><li>Expressions</li><li>Declaration</li><li>Initialization</li><li>Functions</li><li>Statements</li></ul>	<b>Program utilities</b> <ul style="list-style-type: none"><li><b>Variadic functions</b></li><li><b>Diagnostics library</b></li><li><b>Dynamic memory management</b></li><li><b>Strings library</b><ul style="list-style-type: none"><li>Null-terminated strings:<ul style="list-style-type: none"><li>byte – multibyte – wide</li></ul></li></ul></li><li><b>Date and time library</b></li><li><b>Localization library</b></li><li><b>Input/output library</b></li></ul>	<b>Algorithms library</b> <ul style="list-style-type: none"><li><b>Numerics library</b><ul style="list-style-type: none"><li>Common mathematical functions</li><li>Floating-point environment (C99)</li><li>Pseudo-random number generation</li><li>Complex number arithmetic (C99)</li><li>Type-generic math (C99)</li><li>Bit manipulation (C23)</li><li>Checked integer arithmetic (C23)</li></ul></li><li><b>Concurrency support library</b> (C11)</li></ul>
--	---	--

N1570 Committee Draft — April 12, 2011 ISO/IEC 9899:201x

---

**INTERNATIONAL STANDARD** ©ISO/IEC **ISO/IEC 9899:201x**

---

**Programming languages — C**

die.net

Linux Documentation

Search all of the Linux documentation available on this site:

UNTERSTÜTZT DURCH Google Search

Site Search

**Man Pages**  
Sections 1, 2, 3, 4, 5, 6, 7, 8, l, or n

Library

- linux docs
- linux man pages
- page load time

Toys

- world sunlight
- moon phase
- trace explorer

[HOWTO Collection](#)  
[Advanced Bash-Scripting Guide](#)  
[Bash Guide for Beginners](#)  
[Bugzilla Guide](#)  
[Dive Into Python](#)  
[Enterprise Volume Management System Users Guide](#)  
[Introduction to Linux](#)  
[Linux Command-Line Tools Summary](#)  
[Linux Kernel Module Programming Guide](#)  
[Linux with Mobile Devices](#)  
[System Administrators' Guide](#)

```
mp2099@ilab4: ~/cs211/expe x + v
MAN(1) Manual pager utils MAN(1)
NAME
man - an interface to the system reference manuals
Manual page man(1) line 1 (press h for help or q to quit)
```

CS50 Manual Pages

Manual pages for the C standard library, the C POSIX library, and the CS50 Library for those less comfortable.

search

frequently used in CS50

# Agenda

- **Writing General-Purpose Code**
  - Same Operation on Different Types
  - Different Operations on the Same Type
  - Different Operations on Different Types

# Generic Types and Functions

- We want to write **general-purpose code**
  - Functions that work regardless of the argument type
  - Reduce code duplication (e.g., copy/paste)

## Java

```
public class Pair<K, V> {  
    private K key;  
    private V val;  
  
    public Pair(K key, V val)  
    {  
        this.key = key;  
        this.val = val;  
    }  
}
```

## C++

```
template< typename K, typename V >  
class Pair {  
private:  
    K key;  
    V val;  
public:  
    Pair(K key, V val)  
    {  
        this.key = key;  
        this.val = val;  
    }  
};
```

# Generic Types and Functions in C

- C's version of a “placeholder type” is a **void \***

```
void *mem = malloc(N);
```

- They just point to “some location in memory”
  - No idea what is at that location
  - **Can't dereference** without type casting first

```
int i = -1;  
  
void *p = (void *)&i;
```

```
uint8_t u[10] = {0};  
  
p = (void *)&u;
```

# Generic Types and Functions in C

- C's version of a “placeholder function” is a **function pointer**

```
float (*fn)(float) = NULL;  
fn = &sin;  
fn = &cos;  
fn = &tan;
```

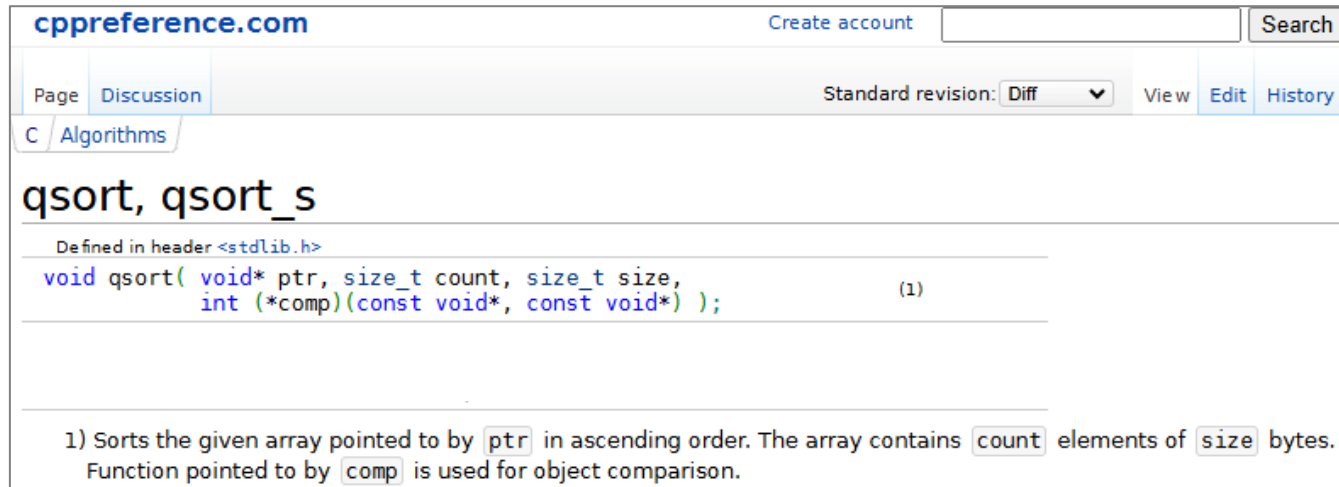
- A function pointer can point to any function with a given signature
  - Can even combine this with void pointers (we rarely do that, though 😊)

```
void *fn = (void *)&sin;
```



# Example: Generic Pointers and Functions in C

- Sort an array using a **custom comparison function**



The screenshot shows the cppreference.com website. At the top, there is a search bar and a 'Create account' link. Below that, there are tabs for 'Page' and 'Discussion', and a 'Standard revision: Diff' dropdown menu. The main content area shows the path 'C / Algorithms' and the title 'qsort, qsort\_s'. Below the title, it says 'Defined in header <stdlib.h>'. The function signature is: `void qsort( void* ptr, size_t count, size_t size, int (*comp)(const void*, const void*) );` with a '(1)' reference. At the bottom, there is a description: '1) Sorts the given array pointed to by `ptr` in ascending order. The array contains `count` elements of `size` bytes. Function pointed to by `comp` is used for object comparison.'

```
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    int arg1 = *(const int*)a;
    int arg2 = *(const int*)b;
    if (arg1 < arg2)
        return -1;
    if (arg1 > arg2)
        return 1;
    return 0;
}

int main(void)
{
    int arr[] = { 8, 33, -4, -21, 0x33, 4 };
    qsort(arr,
          sizeof(arr) / sizeof(arr[0]),
          sizeof(arr[0]),
          compare);
}
```

# Writing General-Purpose Code

## Argument Types

Same

Different

Operation

Same

*Not a problem*

```
char *alloc_mem(char a) {...}  
void *alloc_mem(uint64_t *p) {...}
```

```
void swap(float *a, float *b) {...}  
void swap(char *s0, char *s1) {...}  
void swap(struct s *s0, struct s *s1) {...}
```

**Use Void Pointers**

```
void print_hex(float f) {...}  
void print_dec(int i) {...}  
void print_str(char *str) {...}
```

```
void sort_float(float *arr) {...}  
void sort_str(char *arr) {...}  
void sort_arr(struct custom *arr) {...}
```

**Use Void + Function Pointers  
(+ encode the types)**

```
float ln(float f) {...}  
float csc(float f) {...}
```

```
void quick_sort(float *arr) {...}  
void bubble_sort(float *arr) {...}  
void insertion_sort(float *arr) {...}
```

**Use Function Pointers**

Different

# Agenda

- Writing General-Purpose Code
  - **Same Operation on Different Types**
  - Different Operations on the Same Type
  - Different Operations on Different Types

# Writing General-Purpose Code

## Argument Types

Same

Different

Operation

Same

*Not a problem*

```
char *alloc_mem(char a) {...}  
void *alloc_mem(uint64_t *p) {...}
```

```
void swap(float *a, float *b) {...}  
void swap(char *s0, char *s1) {...}  
void swap(struct s *s0, struct s *s1) {...}
```

**Use Void Pointers**

```
void print_hex(float f) {...}  
void print_dec(int i) {...}  
void print_str(char *str) {...}
```

```
void sort_float(float *arr) {...}  
void sort_str(char *arr) {...}  
void sort_arr(struct custom *arr) {...}
```

```
float ln(float f) {...}  
float csc(float f) {...}
```

```
void quick_sort(float *arr) {...}  
void bubble_sort(float *arr) {...}  
void insertion_sort(float *arr) {...}
```

**Use Function Pointers**

**Use Void + Function Pointers  
(+ encode the types)**

Different

# Same Operation; Different Types

- Operations that don't care about the type (e.g., copy, move, compare)
- Strategy: pass a **pointer** and the **size in bytes**

## Non-Generic Function

```
void swap(char a, char b) {...}
void swap(signed char a, signed char b) {...}
void swap(unsigned char a, unsigned char b) {...}

void swap(short a, short b) {...}
void swap(unsigned short a, unsigned short b) {...}

void swap(int a, int b) {...}
void swap(unsigned int a, unsigned int b) {...}

void swap(long a, long b) {...}
void swap(unsigned long a, unsigned long b) {...}
```

## Generic Function

```
void swap(void *a, void *b, int n_bytes)
{
    uint8_t *ua = (uint8_t *)a;
    uint8_t *ub = (uint8_t *)b;
    for(int i = 0; i < n_bytes; i++, ua++, ub++)
    {
        uint8_t temp = *ua;
        *ua = *ub;
        *ub = temp;
    }
}

void func(void)
{
    int a[2000] = {0},
        b[2000] = {0};
    swap(&a[8], &b[8], sizeof(a[8])); // 9th element
    swap(a, b, sizeof(a)); // entire array
}
```

# <stdlib.h>: Memory Allocation/Freeing

- Size matters, but type doesn't: **use void pointers**

**Header File**

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

**Prototype**

```
void *malloc(size_t size);
```

Think of `void *` as meaning the address of any type of value in memory. Think of `size_t` as a `long`.

---

**DESCRIPTION**

less comfortable

This function dynamically allocates `size` contiguous bytes of memory (on the heap) that can be used to store any type of values.

---

**RETURN VALUE**

less comfortable

This function returns the address of the first byte of memory allocated or `NULL` in cases of error (as when insufficient memory is available).

**Header File**

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

**Prototype**

```
void free(void *ptr);
```

Think of `void *` as meaning the address of any type of value in memory.

---

**DESCRIPTION**

less comfortable

This function frees memory that has been dynamically allocated with `malloc`. It expects as input the pointer that was returned by `malloc`.

---

**RETURN VALUE**

less comfortable

This function does not return a value.

# <string.h> Copying Objects in Memory

- Copy/move some number of bytes from address SRC to DEST
- **Does NOT** allocate memory for DEST

```
void * memcpy ( void * destination, const void * source, size_t num );
```

## Copy block of memory

Copies the values of *num* bytes from the location pointed to by *source* directly to the memory block pointed to by *destination*.

man memcpy: “The memory areas **must not overlap**.”

```
void * memmove ( void * destination, const void * source, size_t num );
```

## Move block of memory

Copies the values of *num* bytes from the location pointed by *source* to the memory block pointed by *destination*. Copying takes place as if an intermediate buffer were used, allowing the *destination* and *source* to overlap.

man memmove: “copying takes place **as though** the bytes in *src* are first copied into a **temporary array**...”

```
void func(void)
{
    char a[100];
    const char *str = “test”;
    memcpy(a, str, strlen(str) + 1);
}
```

```
void func(void)
{
    char a[100];
    const char *str = “test”;
    memmove(a, str, strlen(str) + 1);
}
```

# Revisiting “Swap” (Assuming No Overlap)

## Swap (Original)

```
void swap(void *a, void *b, int n_bytes)
{
    uint8_t *ua = (uint8_t *)a;
    uint8_t *ub = (uint8_t *)b;
    for(int i = 0; i < n_bytes; i++, ua++, ub++)
    {
        uint8_t temp = *ua;
        *ua = *ub;
        *ub = temp;
    }
}
```

## Swap (New)

```
void swap(void *a, void *b, int n_bytes)
{
    void *temp = malloc(n_bytes);
    memcpy(temp, a, n_bytes);
    memcpy(a, b, n_bytes);
    memcpy(b, temp, n_bytes);
    free(temp);
}
```



# <string.h> Manipulating Memory and Strings

## General functions for manipulating memory

### SYNOPSIS

```
#include <string.h>
int memcmp(const void s1[.n], const void s2[.n], size_t n);
```

### DESCRIPTION

The `memcmp()` function compares `n` bytes of memory starting at `s1` and `s2`. It returns an integer value greater than, equal to, or less than zero, depending on whether `s1` is greater than, equal to, or less than `s2`.

### RETURN VALUE

The `memcmp()` function returns an integer value greater than, equal to, or less than zero, depending on whether `s1` is greater than, equal to, or less than `s2`.

For a nonzero difference between `s1` and `s2`, the difference is the number of `char` (i.e., `unsigned char`) that differ.

If `n` is zero, the return value is zero.

### SYNOPSIS

```
#include <string.h>
void *memcpy(void dest[restrict .n], const void src[restrict .n], size_t n);
```

### DESCRIPTION

The `memcpy()` function copies `n` bytes from the memory area `src` to the memory area `dest`.

### RETURN VALUE

The `memcpy()` function returns a pointer to the memory area `dest`.

### SYNOPSIS

```
#include <string.h>
void *memset(void s[.n], int c, size_t n);
```

### DESCRIPTION

The `memset()` function fills the first `n` bytes of the memory area pointed to by `s` with the constant byte `c`.

### RETURN VALUE

The `memset()` function returns a pointer to the memory area `s`.

## Specialized functions for C Strings

### Prototype

```
char *strcpy(char *dest, char *src);
```

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

```
int strcmp(string s1, string s2);
```

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

**No explicit size!**  
**Null terminator instead**

```
int strlen(string s);
```

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

### DESCRIPTION

### RETURN VALUE

less comfortable

This function calculates the length of `s`.

### RETURN VALUE

less comfortable

This function returns the number of characters in `s`, excluding the terminating NUL byte (i.e., `'\0'`).

# Aside: String Library's Safe and Unsafe Operations

- The string library provides “overflow safe” versions
  - Explicit size parameters prevents accidental array-out-of-bounds access

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
int strncmp(const char s1[.n], const char s2[.n], size_t n);
```

## DESCRIPTION

The `strcmp()` function compares the two strings `s1` and `s2`. The locale is not taken into account (for a locale-aware comparison, see `strcoll(3)`). The comparison is done using unsigned characters.

`strcmp()` returns an integer indicating the result of the comparison, as follows:

- 0, if the `s1` and `s2` are equal;
- a negative value if `s1` is less than `s2`;
- a positive value if `s1` is greater than `s2`.

The `strncmp()` function is similar, except it compares only the first (at most) `n` bytes of `s1` and `s2`.

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

int main(int argc, char *argv[])
{
    const char *str0 = "test";
    const char *str1 = "test1";

    int c = strcmp(str0, str1); // unsafe
    int c = strncmp(str0, str1, 4); // safe

    return 0;
}
```

# Agenda

- Recap: Void Pointers
- Writing General-Purpose Code
  - Same Operation on Different Types
  - Different Operations on the Same Type**
  - Different Operations on Different Types

# Writing General-Purpose Code

## Argument Types

Same

Different

Operation

Same

*Not a problem*

```
char *alloc_mem(char a) {...}  
void *alloc_mem(uint64_t *p) {...}
```

```
void swap(float *a, float *b) {...}  
void swap(char *s0, char *s1) {...}  
void swap(struct s *s0, struct s *s1) {...}
```

*Use Void Pointers*

```
void print_hex(float f) {...}  
void print_dec(int i) {...}  
void print_str(char *str) {...}
```

```
void sort_float(float *arr) {...}  
void sort_str(char *arr) {...}  
void sort_arr(struct custom *arr) {...}
```

*Use Void + Function Pointers  
(+ encode the types)*

Different

```
float ln(float f) {...}  
float csc(float f) {...}
```

```
void quick_sort(float *arr) {...}  
void bubble_sort(float *arr) {...}  
void insertion_sort(float *arr) {...}
```

*Use Function Pointers*

# Different Operations; Same Type (Scalar)

Different Operations? **Pass in the Operation.**

## Non-Generic Functions

```
float sin(float x) {...}
float cos(float x) {...}
float tan(float x) {...}
float sec(float x) {...}
float csc(float x) {...}
float cot(float x) {...}

float sinh(float x) {...}
float cosh(float x) {...}
float tanh(float x) {...}
float sech(float x) {...}
float csch(float x) {...}
float coth(float x) {...}

float recip(float x)
{...}
```



## Generic Function

```
void apply(float *f, void (*fn)(float))
{
    (*fn)(f);
}

int main(int argc, char *argv[])
{
    float f = 0.0f;
    float res = apply(f, &coth);
    float res = apply(f, &sinh);
    float res = apply(f, &cosh);

    return EXIT_SUCCESS;
}
```

# Different Operations; Same Type (Array)

Different Operations? **Pass in the Operation.**

## Non-Generic Functions

```
float sin(float x) {...}
float cos(float x) {...}
float tan(float x) {...}
float sec(float x) {...}
float csc(float x) {...}
float cot(float x) {...}

float sinh(float x) {...}
float cosh(float x) {...}
float tanh(float x) {...}
float sech(float x) {...}
float csch(float x) {...}
float coth(float x) {...}

float recip(float x)
{...}
```



## Generic Function

```
void apply(float *arr, int len, void (*fn)(float))
{
    for(int i = 0; i < len; i++)
        (*fn)(arr[i]);
}

#define LEN 2000
int main(int argc, char *argv[])
{
    float arr[LEN] = {0.0f};

    apply(&arr[0], 1, &coth);
    apply(arr, LEN, &csc);
    apply(arr, LEN, &recip);

    return EXIT_SUCCESS;
}
```

# Agenda

- Recap: Void Pointers
- Writing General-Purpose Code
  - Same Operation on Different Types
  - Different Operations on the Same Type
  - Different Operations on Different Types**

# Writing General-Purpose Code

## Argument Types

Same

Different

Operation

Same

*Not a problem*

```
char *alloc_mem(char a) {...}  
void *alloc_mem(uint64_t *p) {...}
```

```
void swap(float *a, float *b) {...}  
void swap(char *s0, char *s1) {...}  
void swap(struct s *s0, struct s *s1) {...}
```

*Use Void Pointers*

```
void print_hex(float f) {...}  
void print_dec(int i) {...}  
void print_str(char *str) {...}
```

```
void sort_float(float *arr) {...}  
void sort_str(char *arr) {...}  
void sort_arr(struct custom *arr) {...}
```

*Use Void + Function Pointers  
(+ encode the types)*

Different

```
float ln(float f) {...}  
float csc(float f) {...}
```

```
void quick_sort(float *arr) {...}  
void bubble_sort(float *arr) {...}  
void insertion_sort(float *arr) {...}
```

*Use Function Pointers*



# Different Operations on Different Types

## Method 1: Encode the Type

## Method 2: Use a Callback Function

- Approach used by `printf()`
- Operations are **standardized** for each type

### Duplicated Code

```
void print_float(float f) {...}
void print_char(char c) {...}
void print_int(int i) {...}
...
```

### Generic Function: `printf()`

```
void func(void)
{
    float f = 0.0f;
    char c = '0';
    int i = 0;

    printf("%f%c%d", f, c, i);
}
```

# <stdio.h>: File I/O

The Open Group Base Specifications Issue 8  
IEEE Std 1003.1-2024  
Copyright © 2001-2024 The IEEE and The Open Group

## NAME

`fopen` — open a stream

## SYNOPSIS

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

```
FILE *fopen(const char *restrict pathname, const char *restrict mode);
```

## DESCRIPTION

[CX] ☒ Except for the "exclusive access" requirement (see below), the functionality described on this reference page is aligned with the ISO C standard. Any other conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard for all `fopen()` functionality except in relation to "exclusive access". ☒

The `fopen()` function shall open the file whose pathname is the string pointed to by `pathname`, and associates a stream with it.

The `mode` argument points to a character string. The behavior is unspecified if any character occurs more than once in the string. If the string begins with one of the following characters, then the file shall be opened in the indicated mode. Otherwise, the behavior is undefined.

'r'	Open file for reading.
'w'	Truncate to zero length or create file for writing.
'a'	Append; open or create file for writing at end-of-file.

```
void func(void)
{
    FILE *f = fopen("~/cs211/myfile", "r");
    if(f == NULL)
    {
        printf("failed to open");
        return;
    }

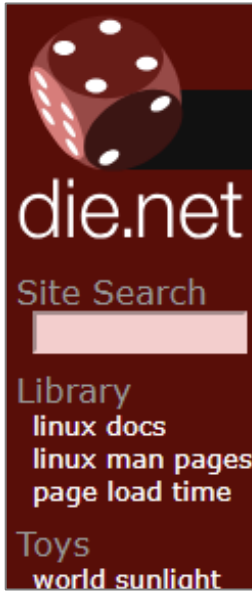
    fread(...);
    fwrite(...);

    ...

    fclose(f);
}
```

Different “types” of files  
through one interface

# <stdio.h>: printf (and friends)



## printf(3) - Linux man page

### Name

printf, fprintf, sprintf, snprintf, vprintf, fprintf, vsprintf, vsnprintf - formatted output conversion

### Synopsis

#include <stdio.h>

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
```

## printf

```
void func(void)
{
    printf("%s", "str");
    printf("%d", 2);
    ...
}
```

## fprintf

```
void func(void)
{
    FILE *f = fopen("myfile", "r");
    fprintf(f, "%s", "str");
    fprintf(f, "%d", 2);
    ...
}
```

## sprintf

```
void func(void)
{
    char str[1024];
    sprintf(str, "%s", "str");
    sprintf(str, "%d", 2);
    ...
}
```

# Aside: sprintf vs snprintf

```
int sprintf(char *str, const char *format, ...);  
int snprintf(char *str, size_t size, const char *format, ...);
```

**Safety feature:**  
stop writing after size bytes

```
void func(void)  
{  
    char str[2];  
    sprintf(str, "%d", 0x3ab5e3da); // unsafe: potential array out of bounds access  
    snprintf(str, sizeof(str), "%d", 0x3ab5e3da); // safe  
}
```

# Different Operations on Different Types

Method 1: Encode the Type

Method 2: Use a Callback Function

- Approach used by `qsort()` and `bsearch()`
- **Custom** operations on different types using function pointers

## Duplicated Code

```
void quicksort_gt(float *f) {...}
void quicksort_lt(char *str[]) {...}
void quicksort_ge(int *i) {...}
void quicksort_le(struct mytype *s) {...}
```

## Generic Function: `qsort()`

```
int compare(const void *a, const void *b)
{
    return strcmp((const char *)a, (const char *)b);
}

void func(void)
{
    char *str[] = {"abc", "bca", "cab"};
    qsort(str, 3, sizeof(char *), &compare);
}
```

# Other Uses of Callback Functions

- **Error reporting**
- **Asynchronous library calls** (e.g., network request, disk access, GUI apps)
- **Event-driven programming** (e.g., simulations)

```
void on_mouse_move(int button, int x, int y) {...}
void on_key_press(int key, int modifiers) {...}
void on_key_release(int key, int modifiers) {...}
void on_window_resize(int old_x, int old_y, int new_x, int new_y) {...}
void on_exit(void) {...}

int main(int argc, char *argv[])
{
    ...
    register_mouse_callback(&on_mouse_move);
    register_keyboard_callback(&on_key_press, KEY_PRESS);
    register_keyboard_callback(&on_key_release, KEY_RELEASE);
    register_resize_callback(&on_window_resize);
    register_close_button_press_callback(&on_exit);
    ...
}
```

# Typedef

- **User-defined alias** for an existing type (purely for convenience)

```
typedef existing_type new_type;
```

## Struct

```
struct point
{
    int x;
    int y;
};

typedef point point_t;

point_t p = {0, 0};
point_t *p_p = &p;

...
```

## Array

```
typedef int[5] arr_t;

arr_t a = {0};
int *a_as_p = &a[0];

...
```

## Pointer

```
typedef int (*fn_t)(int a);

fn_t fn = &...;
int ret = (*fn)(0);

...
```

# **CS 211: Intro to Computer Architecture**

## ***7.2: C Generics and Library Functions***

**Minesh Patel**

Spring 2025 – Thursday 6 March