Karlsruhe Institute of Technology

Operating Systems

Introduction to the C Programming Language
Dr.-Ing. Marc Rittinghaus | WT 2020/2021

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – ITEC – OPERATING SYSTEMS

```c
#include "malloc.h"

#include <stdio.h>
#include <assert.h>

typedef struct _Block {
    /*
     * Pointer to the header of the next free block.
     * Only valid if this block is also free.
     * This is null for the last Block of the free list.
     */
    struct _Block *next;

    /*
     * Our header should always have a size of 16 Bytes.
     * This is just for 32 bit systems.
     */
    uint8_t padding[8 - sizeof(void*)];

    /*
     * The size of this block, including the header
     * Always a multiple of 16 bytes.
```

```c
} Block;

#define HEADER_SIZE sizeof(Block)
#define INV_HEADER_SIZE_MASK ~((uint64_t)HEADER_SIZE - 1)

/*
 * This is the heap you should use.
 * 16 MiB heap space per default. The heap does not grow.
 */
#define HEAP_SIZE (HEADER_SIZE * 1024 * 1024)
uint8_t __attribute__ ((aligned(HEADER_SIZE))) _heapData[HEAP_SIZE];

/*
 * This should point to the first free block in memory.
 */
Block *_firstFreeBlock;

/*
 * Initializes the memory block. You don't need to change this.
 */
void initAllocator()
{
```

Introduction

- C
  - General-purpose, procedural language
  - Developed beginning of 1970s by Dennis Ritchie (Bell Labs)
  - Designed for implementing system software
  - Still widely used and requested programming language

Oct 2020	Oct 2019	Change	Programming Language	Ratings	Change
1	2		C	16.95%	+0.77%
2	1		Java	12.56%	-4.32%
3	3		Python	11.28%	+2.19%
4	4		C++	6.94%	+0.71%
5	5		C#	4.16%	+0.30%
6	6		Visual Basic	3.97%	+0.23%

Source: https://www.tiobe.com/tiobe-index/

Why C?

Pros

- Close to the machine
  - Low-level memory access
  - Maps efficiently to machine instructions
- Efficient
- Portable

Cons

- Limited type safety
- Error-prone
- Tedious
- Low productivity
- Not object oriented

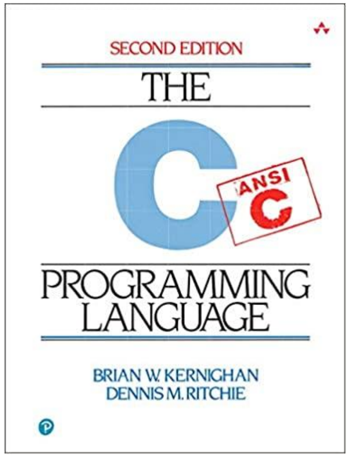
Widely used for OSes and embedded systems

➢ Anywhere where efficiency matters (space/time)

Operating Systems Group
Department of Computer Science

Goal of This Lecture

- NOT a complete reference for C
  - Refer to "The C Programming Language" by Kernighan & Ritchie (K&R)



- We discuss:
  - General structure of a C program
  - Basic and complex data types
  - Pointers and pointer arithmetic
  - Dynamic memory allocation
  - Bit arithmetic

  This lecture

  - C preprocessor
  - Calling conventions
  - Static & dynamic linking

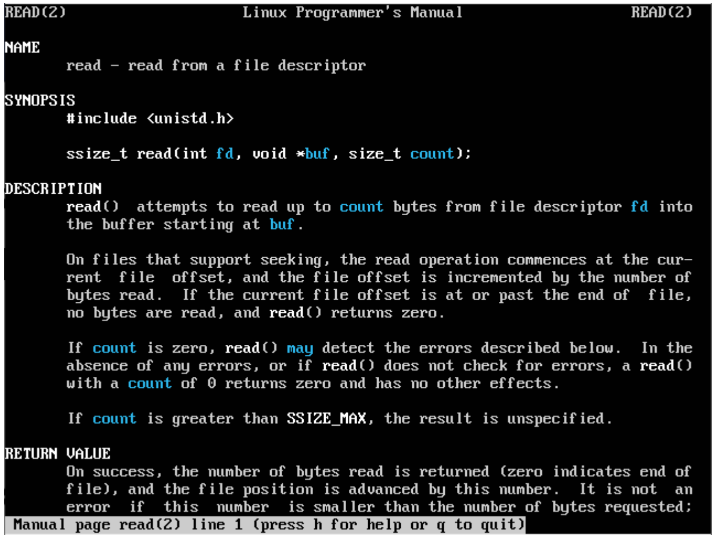
  Next lecture

➢ Provide easier start to programming assignments

Man Pages

- During assignments, get help as you need:
  - Library calls / system calls, parameters, return values
  - UNIX man(ual) page (`man`)
  - man page sections (`man 1 ls`):
    - 1 commands (`ls, gcc, gdb`)
    - 2 system calls (`read, gettimeofday`)
    - 3 library calls (`printf, scanf`)
    - 5 file formats (`passwd`)
    - 6 miscellaneous (`signal`)

```
user@host:~$ man 2 read
```



```
READ(2)                    Linux Programmer's Manual                   READ(2)

NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  attempts to read up to count bytes from file descriptor fd into
       the buffer starting at buf.

       On files that support seeking, the read operation commences at the cur-
       rent  file  offset, and the file offset is incremented by the number of
       bytes read.  If the current file offset is at or past the end of  file,
       no bytes are read, and read() returns zero.

       If count is zero, read() may detect the errors described below.  In the
       absence of any errors, or if read() does not check for errors, a read()
       with a count of 0 returns zero and has no other effects.

       If count is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE
       On success, the number of bytes read is returned (zero indicates end of
       file), and the file position is advanced by this number.  It is not  an
       error  if  this  number  is smaller than the number of bytes requested;
 Manual page read(2) line 1 (press h for help or q to quit)
```

Hello World!

helloworld.c

```c
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

- #include — preprocessor (inserts contents of file)
- stdio.h — contains the declaration of printf
- main — program starts here (entry point)
- { } — basic blocks / scope delimiters
- printf — prints to the terminal
- ‘\n’ — newline character
- return — leave function and return value (here 0)

Compiling and Running Hello World!

```
user@host:~$ gcc helloworld.c -o helloworld
user@host:~$ ./helloworld
Hello World!
```

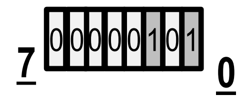
- Compilation
  - Generating binary executable from source code
  - Two main steps (besides preprocessor):
    - Generate a binary **object file (.o)** for each **source file (.c)**
    - Link object files to executable / library (resolve addresses)
- Execution
  - Operating system loads binary into memory, …
  - …may loads additional libraries,
  - …starts execution at entry point

Operating Systems Group
Department of Computer Science

Basic Data Types

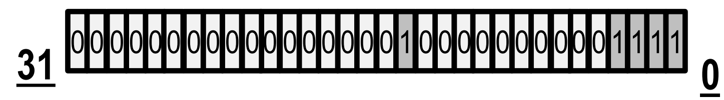
- **char**: 1 byte, usually for characters

```
char c = 5;      // Decimal
char d = 'a';    // ASCII: 97
```



- **int**: at least 2 bytes, usually 4 bytes

```
int i = 040017; // Octal (dec: 16399)
int j = 0x400f; // Hexadecimal notation
int k = 'a';
```



- **long**: at least 4 bytes
- **long long**: at least and usually 8 bytes

- Integers can be **signed** (default) or **unsigned**

```
signed int i;    // usually [−2,147,483,647 … +2,147,483,647]
unsigned int j;  // usually [0 … 4,294,967,295]
```

Ranges and bit sizes vary with platform and architecture!

sizeof, inttypes.h

- Use **sizeof** to determine size of type or variable

```
sizeof(int); // 4 (Linux,x86)
```

```
long l;
sizeof(l);   // 4 (Linux,x86) and 8 (Linux,x86-64)
```

⚠

- Use types from **inttypes.h** to be sure about sizes

```
#include <inttypes.h>
uint8_t a;  // unsigned 1 byte [0...255]
int8_t b;   // signed 1 byte [-128...127]
uint32_t c; // unsigned 4 bytes [0...4294967295]
int64_t d;  // signed 8 bytes
```

- Important for serialization
  - e.g., on-disk data structures

Compound Data Types

- **struct**ure: Collection of named variables of different types

```c
struct coord {
  uint32_t x;
  uint32_t y;
};
```

- Members accessed by name

```c
struct coord c;
c.x = 5;
c.y = 10;
```

- Initialization

```c
struct coord c = {.x = 5, .y = 10};
struct coord d = {5, 10};
struct coord e = {5}; // e.y = 0
```

```c
struct coord f; // f.x = ?, f.y = ?
```


Operating Systems Group
Department of Computer Science

typedef

- typedef: Allows declaring alias names
  - Simplify syntax of complex type

```
typedef struct coord {
  uint32_t x;
  uint32_t y;
} coord;

struct coord c1; // OK
coord c2;        // OK (w/o typedef → unknown type)
```

  - Give more descriptive name for existing type

```
typedef int length;

length l = 5;
```

Size of Compound Types

- Compiler may insert padding to ensure data alignment
  - Members are aligned to a multiple of their size (basic types)

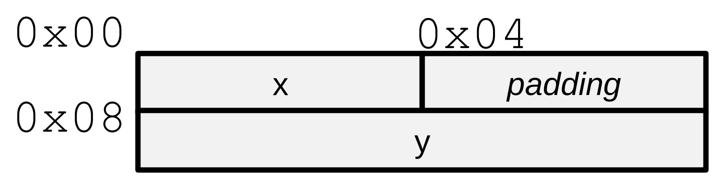
```
typedef struct coord {
  uint32_t x;
  uint32_t y;
} coord;
```

```
sizeof(coord) // 4 + 4 = 8
```


```
typedef struct coord {
  uint32_t x;
  uint64_t y;
} coord;
```

padding

```
sizeof(coord) // WRONG: 4 + 8     = 12
sizeof(coord) // RIGHT: 4 + 4 + 8 = 16
```

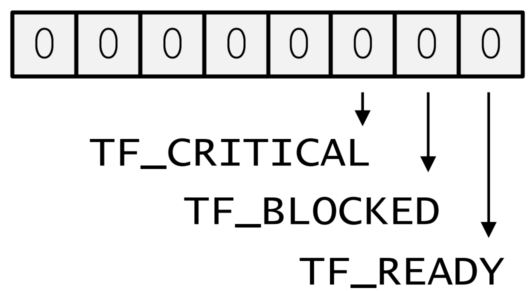


Enumerations

- **enum**eration: Collection of named values
  - Define descriptive names

```c
enum thread_priority {
  TP_NORMAL,
  TP_HIGH,
  TP_HIGHEST
};

void set_priority(enum thread_priority p);
```

```c
enum thread_flags {
  TF_READY    = 1,
  TF_BLOCKED  = 2,
  TF_CRITICAL = 4
};
```



➢
```c
set_priority(TP_HIGH);
```

- Variable treated as regular `int`
  - All integer arithmetic applies
  - Can contain arbitrary `int` values

Arrays

- Fixed number of variables laid out *continuously* in memory

```
int a[4];
```

?	?	?	?

```
int a[] = {41, 0, 0, 42};
```

41	0	0	42

```
int a[4] = {41, 42};
```

41	42	0	0

```
int a[4] = {0};
```

0	0	0	0

```
coord c[4] = {{1,1},{2,2},{3,3},{4,4}};
```

- Access elements with [ ] operator
  - NO bounds check at compile time or run time!

```
a[0] = 42;   // set first element
a[3] = 42;   // set last element
c[0].x = 42; // set x member of first coord element

a[232] = 42; // compiles and may execute fine but corrupts memory!
```

Arrays - sizeof

- **sizeof:** Returns size of (static) array in memory
  - NOT: number of elements

```
int a[5];
coord c[5];

sizeof(a)  // 5 * sizeof(int)   = 5 * 4 = 20
sizeof(c)  // 5 * sizeof(coord) = 5 * 8 = 40
```

- Get number of elements

```
sizeof(a) / sizeof(int)   // 20 / 4 = 5
sizeof(c) / sizeof(coord) // 40 / 8 = 5
```

or

```
sizeof(a) / sizeof(a[0])  // 20 / 4 = 5
sizeof(c) / sizeof(c[0])  // 40 / 8 = 5
```

Looks at type only!

Pointers

- Pointer: memory address
  - Typically typed, `void` denotes absence of type
  - Use &-operator to get address of variable (→ reference)

```
int i = 5;
int *p = &i;  // p = address of an int variable. Now pointing to i
void *r = p;
struct coord c, *d = NULL; // Convention: NULL = Invalid pointer
p = &c.x;     // change p to address of member x of struct c
```

- Dereferencing: access variable pointed to

```
int j = *p;        // j = 5
int k = *(int*)r; // k = 5, type cast required (r untyped)
int x = (*d).x;   // dereference d, get value of member x
int y = d->y;     // short for (*d).y
```

Pointer Example: Parameter Passing

- Pass-by-value

```
void setX(coord c, int x) {
  c.x = x;
}
```

  - Semantic: assignment works <u>on local copy</u>
  - Performance: have to copy value

- Pass-by-reference

```
void setX(coord *c, int x) {
  c->x = x;
}
```

  - Semantic: assignment works <u>on original data</u>
  - Performance: parameter is only memory address (32/64-bit)

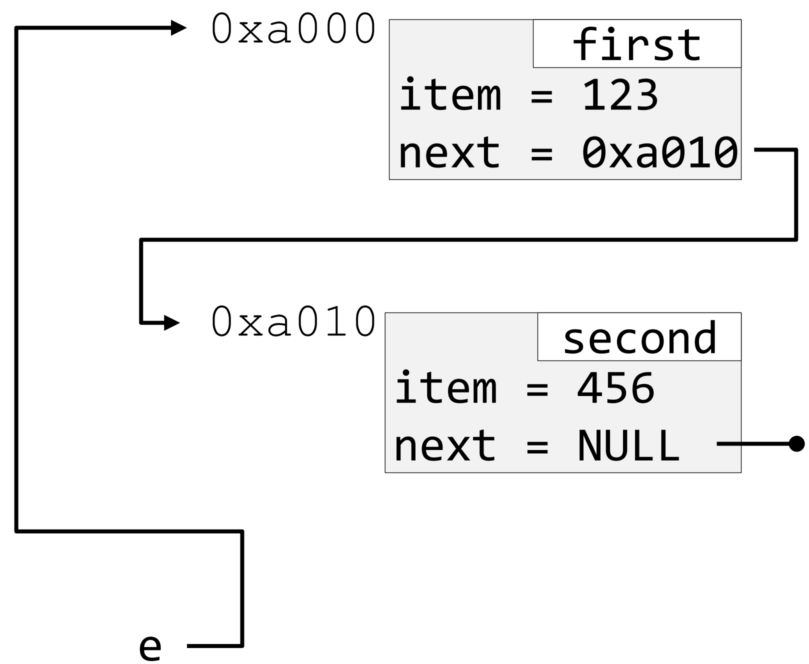
Pointer Example: Linked List

- Linked list via next-pointer

```c
struct entry {
  int item;
  struct entry *next;
} first, second;

first.item = 123;
first.next = &second;

second.item = 456;
second.next = NULL;
```

- Iterate over list

```c
struct entry *e = &first;
while (e != NULL) {
  // do something with e->item
  e = e->next;
}
```



first iteration

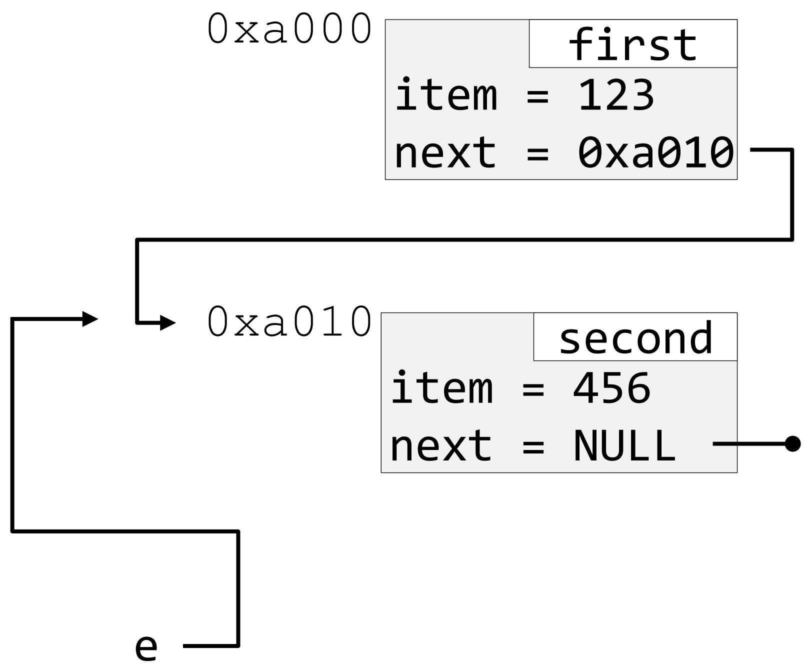
Pointer Example: Linked List

- Linked list via next-pointer

```
struct entry {
  int item;
  struct entry *next;
} first, second;

first.item = 123;
first.next = &second;

second.item = 456;
second.next = NULL;
```

- Iterate over list

```
struct entry *e = &first;
while (e != NULL) {
  // do something with e->item
  e = e->next;
}
```

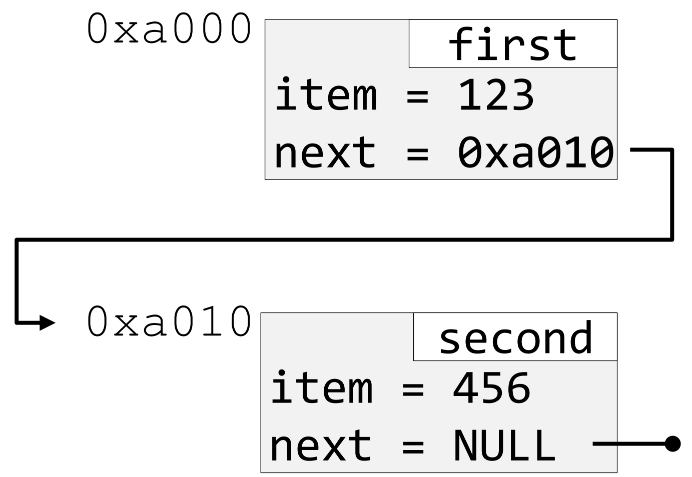


second iteration

Pointer Example: Linked List

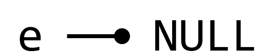
- Linked list via next-pointer

```c
struct entry {
  int item;
  struct entry *next;
} first, second;

first.item = 123;
first.next = &second;

second.item = 456;
second.next = NULL;
```



- Iterate over list

```c
struct entry *e = &first;
while (e != NULL) {
  // do something with e->item
  e = e->next;
}
```



Pointer Arithmetic

- Pointers support addition and subtraction

```c
int a[10];
int *p = a;  // OK: Array = pointer, but could also write &a[0]
             // Example: p = 0x100
int i = *p;  // Access first element: i = p[0]; ↔ i = a[0];
```

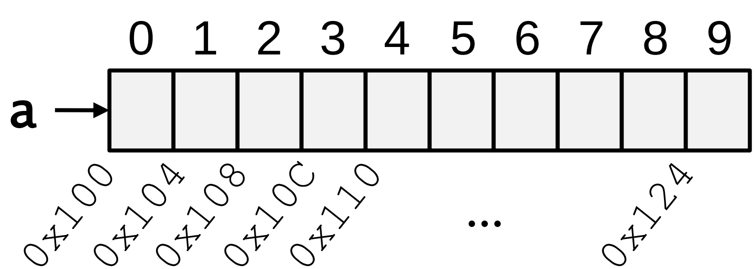
- Access second element in array:

```c
int i = p[1];
```

or

```c
p = p + 1;   // WRONG: p = 0x101
             // RIGHT: p = 0x100 + sizeof(*p) = 0x100 + 4 = 0x104
int i = *p;
```

Size of type int, not of int* ⚠



Stack vs. Heap Allocation

- Do not return pointers to local variables!
  - Local variables are allocated on the stack
  - Stack memory is released when leaving scope of declaration

```
int* myfunc() {
  int a[4] = {0,1,2,3};
  return a;
} // Memory of array will be released here!!
```

```
void main() {
  int *p = myfunc();
  // Accessing p may corrupt memory or crash
  // program!!
}
```

✖ WRONG!
DO NOT DO THIS
→ Use-after-free BUG

Stack vs. Heap Allocation

- Allocate memory on the heap

```
#include <stdlib.h>
void *malloc(size_t size); // Allocates memory on heap
void free(void *ptr);      // Frees memory on heap
```

```
int* myfunc() {
  int *a = (int*)malloc(4 * sizeof(int));
  return a;
}
```

Be careful with `sizeof`!
`sizeof(a)  = 8`
`sizeof(*a) = 4`
NOT: 12

```
void main() {
  int *p = myfunc();
  // Work with array pointed to by p
  free(p);
}
```

Do not forget to free!
→ Memory leak

Operating Systems Group
Department of Computer Science

Stack vs. Heap Allocation

- Allocate memory on the heap

```c
#include <stdlib.h>
void *malloc(size_t size); // Allocates memory on heap
void free(void *ptr);      // Frees memory on heap
```

```c
int* myfunc(int num) {
  int *a = (int*)malloc(num * sizeof(int));
  return a;
}
```

```c
void main() {
  int *p = myfunc(10);
  // Work with array pointed to by p
  free(p);
}
```

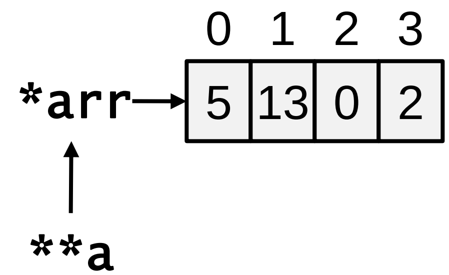
+ Dynamic memory allocation
  → Can decide size at run time

Operating Systems Group
Department of Computer Science

Double Pointers

- Sometimes a pointer to a pointer is required

```c
int *arr = Some DYNAMIC array of len 4;
int len = 4;
len = cond_append(&arr, len, 42);
```




```c
int cond_append(int **a, int len, int e) {
  int i, *p = *a;
  for (i = 0; i < len; i++, p++) {
    if (*p == e) return len; // FOUND!
  }
  p = (int*)malloc((len + 1) * sizeof(int));
  memcpy(p, *a, len * sizeof(int));
  p[len] = e;
  free(*a);
  *a = p;
  return len + 1;
}
```

Search element in array

Allocate larger array
Copy old array to new array
Set last element to e
Free old array
Update `arr` pointer
Return new length

Function Pointers

- Can also create pointers to functions

```c
typedef int (*myfunc)(int, int);

int sum(int a, int b) {
  return a + b;
}

int mul(int a, int b) {
  return a * b;
}

int main()
{
  myfunc f = mul;
  return f(1,2);
}
```


Operating Systems Group
Department of Computer Science

Characters

- Characters are just numbers (0…255)
  - ASCII table (`man ascii`) translates numbers to font glyphs
- Can calculate with characters

```
char c = 'a'; // ASCII: 97
c = c + 1;    // c = ‘b’, since ‘b’ follows ‘a’ in ASCII
```

  - e.g., allows converting from lower to upper case (‘a’ - 32 = ‘A’)
- Special characters encoded via leading backslash

\t	Tab (9)		\”	Double quote (34)
\n	Newline (10)		\’	Single quote (39)
\r	Carriage return (13)		\\	Backslash (92)

⋮

Strings

- String = array of chars + terminating null (\0) char

H	e	l	l	o	_	W	o	r	l	d	!	\0

- Initialize to **copy of** (read-only) literal constant

```
char c[] = "Hello World!";                    // sizeof(c) = 13
char c[] = {'H','e','l','l','o',' ','W','o’,'r','l','d','!','\0'};

c[1] = 'A'; // OK!
```

- Initialize as **pointer to** (read-only) literal constant
  - Use in combination with **const** keyword!

```
char *c = "Hello World!";                     // sizeof(c) = 8 (64-bit)
c[1] = 'A'; // CRASH!
```

```
const char *c = "Hello World!";
c[1] = 'A'; // DOES NOT COMPILE
```

printf (`man 3 printf`)

- Outputs a formatted string to stdout

```
#include <stdio.h>
int printf(const char *format, ...);
```

```
void myprint(const char* str, int i, unsigned long long l) {
  printf("String is %s, i is %i, and l is 0x%016llx", str, i, l);
}
```

- Format: %[flags][width][.precision][length]specifier

Flags:
− left-justify
+ force sign
0 pad with zeros

Minimum number of chars printed

Number of digits for floats

Length of data type

Data type (or interpretation thereof)

```
myprint("Test", 42, 0xF00FF7A989);
```

➤ `String is Test, i is 42, and l is 0x000000f00ff7a989`

Bit Arithmetic

- Low-level programming often requires bit arithmetic
  - Configure hardware registers
  - Densely encode information (e.g., flags, bitmaps)
  - Use with unsigned types only!

```
7 00000101 0   uint8_t a // = 5
7 00000110 0   uint8_t b // = 6

7 00000100 0   a & b  // = 4     And
7 00000111 0   a | b  // = 7     Or
7 11111010 0   ~a     // = 250   Not

7 00000010 0   a >> 1 // = 2     Rsh
7 00001010 0   a << 1 // = 10    Lsh
7 00000011 0   a ^ b  // = 4     Xor
```

Bit Arithmetic - Example

- Mask out bit number 5

```c
uint8_t bitfunc(uint8_t val) {
  uint8_t mask = ~(1 << 5);
  return val & mask;
}
```

Bits (7 … 0)	Expression
7 0 0 0 0 0 0 0 1 0	`1`
7 0 0 1 0 0 0 0 0 0	`1 << 5`
7 1 1 0 1 1 1 1 1 0	`~(1 << 5)`
7 0 0 1 1 0 0 0 1 0	`val // 49`
7 0 0 0 1 0 0 0 1 0	`val & mask // 17`

Summary

- Read man pages
- Use `inttypes.h` to be sure about sizes of basic data types
- Pointers are powerful
  - Pass-by-reference
  - Dynamic memory
  - Complex data structures
- …and prone to bugs
  - Use-after-free, memory leaks, …

Further Reading

- “The C Programming Language” by Kernighan and Ritchie
- comp.lang.c Frequently Asked Questions (http://c-faq.com/)