

CS 211: Intro to Computer Architecture

5.1: C Data Representations

Minesh Patel

Spring 2025 – Tuesday 18 February

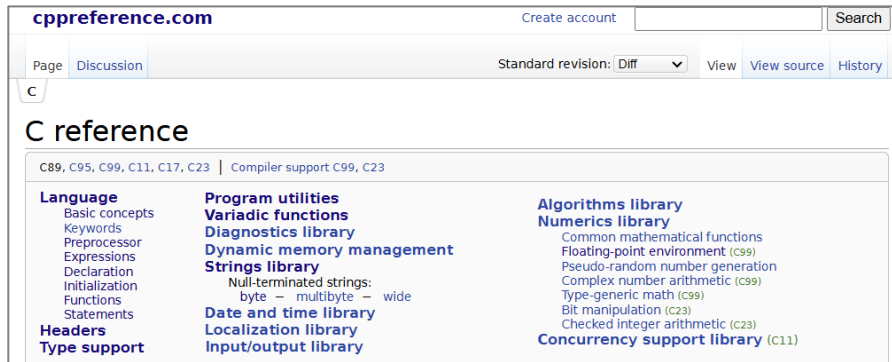
Announcements

- **PA2** due at the end of the week (Sunday, Feb 23 @ 23:59)
- **WA3** due next Monday night (Monday, Feb 24 @ 23:59)
 - **On Canvas this time** instead of Gradescope
 - We should have probably used Canvas from the get-go

Reference Material

- Today's lecture partially draws inspiration from:
 - [CS 122 @ CMU](#) (Profs. Iliano Cervesato and Anne Kohlbrenner)
 - [CS 61C @ UC Berkeley](#) (Prof. Dan Garcia)

And Various C and Linux Reference Materials



cppreference.com

Create account Search

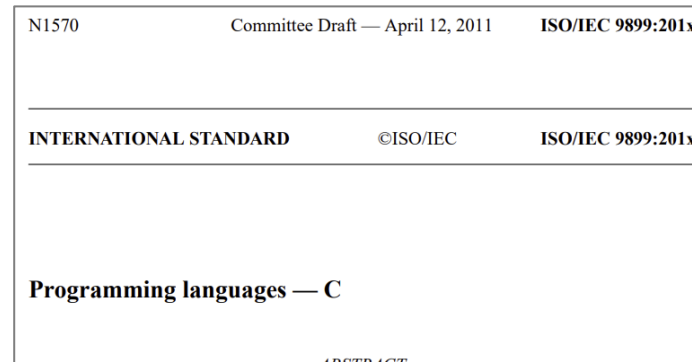
Page Discussion Standard revision: Diff View View source History

C

C reference

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

Language <ul style="list-style-type: none">Basic conceptsKeywordsPreprocessorExpressionsDeclarationInitializationFunctionsStatements	Program utilities <ul style="list-style-type: none">Variadic functionsDiagnostics libraryDynamic memory managementStrings library<ul style="list-style-type: none">Null-terminated strings:<ul style="list-style-type: none">byte – multibyte – wideDate and time libraryLocalization libraryInput/output library	Algorithms library <ul style="list-style-type: none">Numerics library<ul style="list-style-type: none">Common mathematical functionsFloating-point environment (C99)Pseudo-random number generationComplex number arithmetic (C99)Type-generic math (C99)Bit manipulation (C23)Checked integer arithmetic (C23)Concurrency support library (C11)
--	---	--

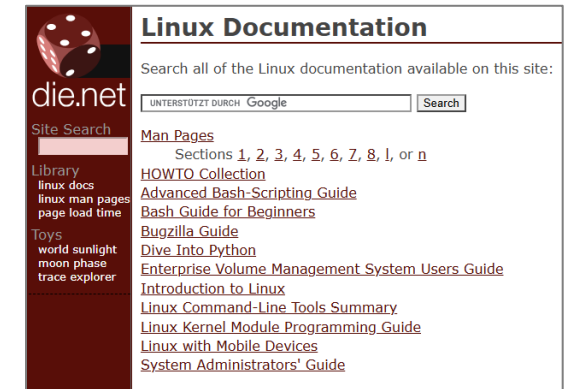


N1570 Committee Draft — April 12, 2011 ISO/IEC 9899:201x

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

Programming languages — C

ABSTRACT



die.net

Linux Documentation

Search all of the Linux documentation available on this site:

UNTERSTÜTZT DURCH Google Search

Site Search

Library

- linux docs
- linux man pages
- page load time

Toys

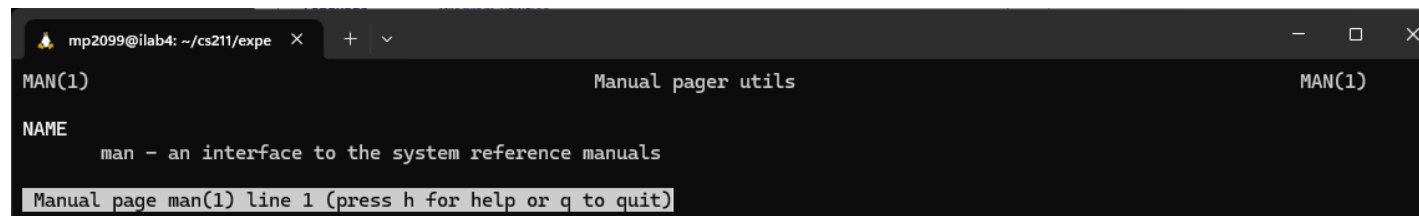
- world sunlight
- moon phase
- trace explorer

Man Pages

- Sections 1, 2, 3, 4, 5, 6, 7, 8, 1, or n

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 + - 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)
```

Foreword

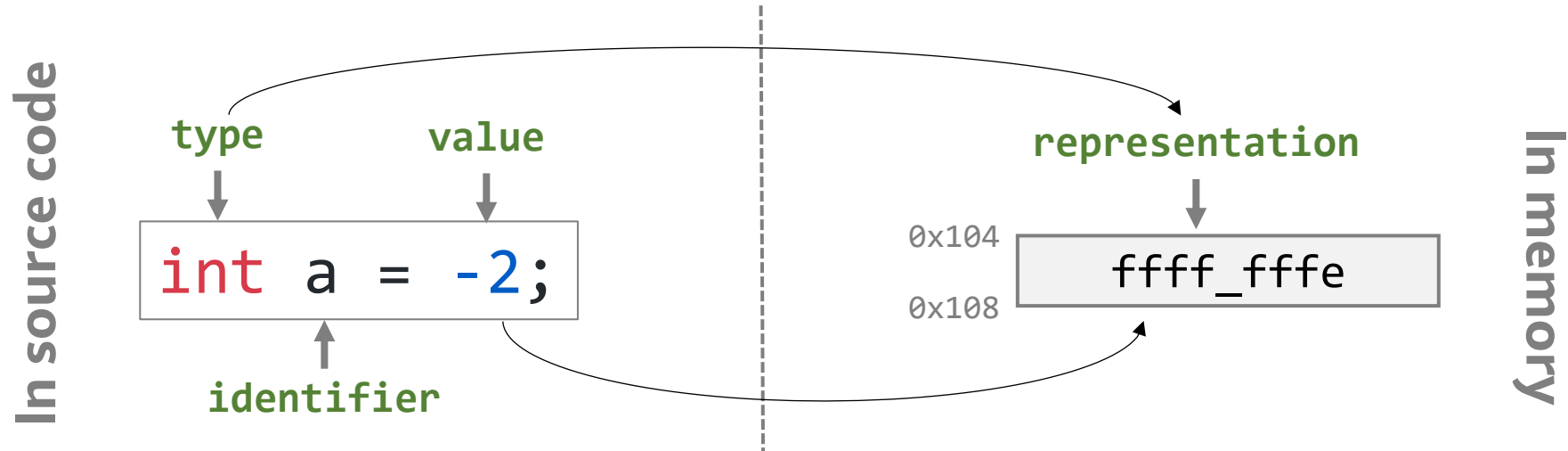
- We **do not** expect you to memorize obscure C programming quirks
 - **Goal of lecture:** know what concepts to look up 😊
- We **do** expect you to:
 - Look up reference material as needed (e.g., when you run into problems)
 - Go over examples yourself after lecture

Agenda

- **C's Type System**

Why Does C Have Types?

- **Types** tell the compiler what to do with your code
 - How to represent **identifiers** as **objects in memory**
 - Defines an object's **size** and **supported operations**



6.2.5 Types

- 1 The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it.

ISO Standard 9899:201x

cppreference.com [Create account](#)

Page Discussion Standard revision: Diff

C C language Basic Concepts

Type

(See also [arithmetic types](#) for the details on most built-in types and [the list of type-related utilities](#) that are provided by the C library.)

[Objects](#), [functions](#), and [expressions](#) have a property called *type*, which determines the interpretation of the binary value stored in an object or evaluated by the expression.

Void Types

- **void**
- basic types
 - char
 - signed integers
 - unsigned integers
 - floating-point
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

6.2.5 Types

1 The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it.

an object type may be *incomplete* (lacking sufficient information to determine the size of objects of that type) or *complete* (having sufficient information).³⁷⁾

19 The **void** type comprises an empty set of values; it is an incomplete object type that cannot be completed.

ISO Standard 9899:201x

~~void a;~~

```
mp2099@ilab2:~/cs211/experiment$ /common/system/riscvi/b
void.c: In function 'main':
void.c:3:14: error: variable or field 'a' declared void
   3 |     void a;
     |           ^
mp2099@ilab2:~/cs211/experiment$
```

identifier

void function(void);

↑
type

compiler sees:
“no return value”

↑
type

compiler sees:
“no arguments”

Void vs. Unknown Function Arguments

Caution: never leave your argument list empty

`void function(void);`

not the same

`void function();`

compiler sees:
“expect no arguments”

```
mp2099@ilab2: ~/cs211/expe x + v - □ x
mp2099@ilab2:~/cs211/experiment$ cat bad_arg.c
#include <stdlib.h>

void empty(void) {}
int main(int argc, char *argv[])
{
    empty(0.1);
    return EXIT_SUCCESS;
}
mp2099@ilab2:~/cs211/experiment$ /common/system/risc
vi/bin/riscv64-unknown-elf-gcc -o bad_arg bad_arg.c
bad_arg.c: In function 'main':
bad_arg.c:6:5: error: too many arguments to function
'empty'
   6 |     empty(0.1);
     |           ^~~~~~
bad_arg.c:3:6: note: declared here
   3 | void empty(void) {}
     |           ^~~~~~
mp2099@ilab2:~/cs211/experiment$ |
```

compiler sees:
“arguments not known”

```
mp2099@ilab2: ~/cs211/expe x + v - □ x
mp2099@ilab2:~/cs211/experiment$ cat bad_arg.c
#include <stdlib.h>

void empty() {}
int main(int argc, char *argv[])
{
    empty(0.1);
    return EXIT_SUCCESS;
}
mp2099@ilab2:~/cs211/experiment$ /common/system/risc
vi/bin/riscv64-unknown-elf-gcc -o bad_arg bad_arg.c
mp2099@ilab2:~/cs211/experiment$ ./bad_arg
mp2099@ilab2:~/cs211/experiment$ echo $?
0
mp2099@ilab2:~/cs211/experiment$ |
```

↓
compilers disables
argument checking

↓
no warnings or errors

↓
potential bugs

Basic Types: Agenda

- `void`
- **basic types**
 - `char`
 - `signed integers`
 - `unsigned integers`
 - `floating-point`
- `enumerated types`
- `derived types`
 - `arrays`
 - `structures`
 - `unions`
 - `functions`
 - `pointers`

- **Overview**
- `Format strings`
- `Initialization`
- `Arithmetic`
- `Type casting`

Basic Types

- **Types** connect **objects** with **representations**

Programmer **declares** the identifier/type

```
void func(void)
{
    char c = 'c'; // ASCII codepoint 99
    signed int s = -0x2;
    unsigned int u = 0xc0decafe;
    int64_t i = -683;
    float f = -0.000001;
}
```

Compiler **represents** it as
an object of the **type**

0x100	6300_0000
0x104	ffff_fffe
0x108	c0de_cafe
0x10c	ffff_ffff _ffff_fd55
0x114	bd37_86b5
0x118	

Character Types

- void
- basic types
 - char
 - signed integers
 - unsigned integers
 - floating-point
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

Platform Dependent

15 The three types `char`, `signed char`, and `unsigned char` are collectively called the *character types*. **The implementation shall define** `char` to have the same range, representation, and behavior as either `signed char` or `unsigned char`.⁴⁵⁾

ISO Standard 9899:201x

`char` → Meant for text (e.g., 'a')
`signed char`
`unsigned char` } → Meant for numbers

ASCII
(Codepoints 32-63)

Dec	Hex	Dec	Hex
32	20	48	30 0
33	21	49	31 1
34	22	50	32 2
35	23	51	33 3
36	24	52	34 4
37	25	53	35 5
38	26	54	36 6
39	27	55	37 7
40	28	56	38 8
41	29	57	39 9
42	2A	58	3A :
43	2B	59	3B ;
44	2C	60	3C <
45	2D	61	3D =
46	2E	62	3E >
47	2F	63	3F ?

```
int is_binary_digit(char c)
{
    return (c == '0' || c == '1');
    return (c >= '0' && c <= '1');
    return (c > '/' && c <= '2');
    return (c >= 48 && c <= 49);
    return (c > 0x2f && c <= 0x32);
}
```

Integer Types

- void
- basic types
 - char
 - **signed integers**
 - **unsigned integers**
 - floating-point
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

Type specifier	Equivalent type	Width in bits
		C standard
char	char	at least 8
signed char	signed char	
unsigned char	unsigned char	
short	short int	at least 16
short int		
signed short		
signed short int		
unsigned short	unsigned short int	
unsigned short int		
int		
signed	int	at least 16
signed int		
unsigned	unsigned int	
unsigned int		
long	long int	at least 32
long int		
signed long		
signed long int		
unsigned long	unsigned long int	
unsigned long int		
long long	long long int (C99)	at least 64
long long int		
signed long long		
signed long long int		
unsigned long long	unsigned long long int (C99)	
unsigned long long int		

Besides the minimal bit counts, the C Standard guarantees that

$$1 == \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$$

Platform Dependent

=

“Implementation Defined”
in C terminology

Aside: Undefined vs. Implementation-Defined Behavior

- C standards leave many decisions to the platform (i.e., compiler)

N1548	Committee Draft — December 2, 2010	ISO/IEC 9899:201x
Annex I (informative)	Common warnings	548
Annex J (informative)	Portability issues	550
J.1	Unspecified behavior	550
J.2	Undefined behavior	553
J.3	Implementation-defined behavior	566
J.4	Locale-specific behavior	574
J.5	Common extensions	575

Aside: Undefined vs. Implementation-Defined Behavior

Undefined Behavior

- Unpredictable: may differ across executions
- Compiler may or may not warn you

```
#include <limits.h>

int f0(void)
{
    return 1 / 0; // div/mod by zero
}

int f1(void)
{
    int a;
    return a; // uninitialized object
}

int f2(void)
{
    return INT_MAX + 1; // signed overflow
}
```

Implementation Defined Behavior

- Each compiler makes its own decision
- Consistent on the same platform

..... If the sign bit is one, the value shall be modified in one of the following ways:

- the corresponding value with sign bit 0 is negated (*sign and magnitude*);
- the sign bit has the value -2^M (*two's complement*);
- the sign bit has the value $-(2^M - 1)$ (*ones' complement*).

Which of these applies is implementation-defined,

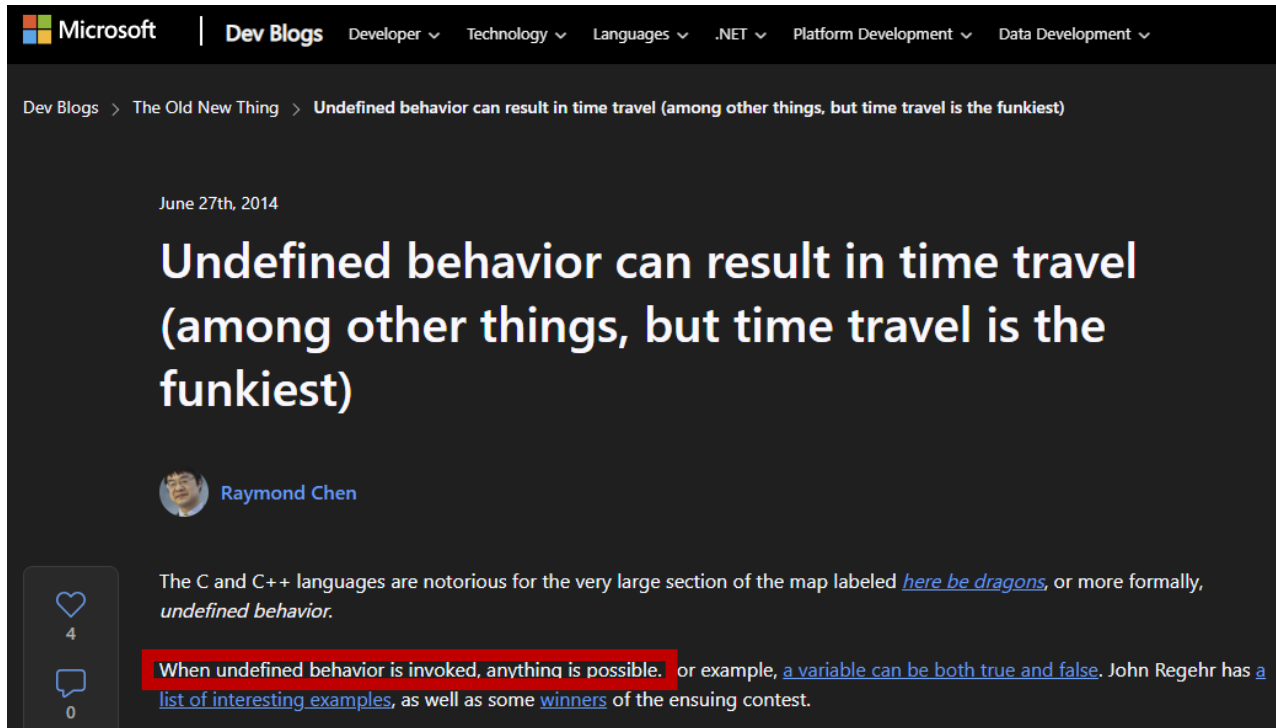
ISO Standard 9899:201x

```
#include <limits.h>

int f0(void)
{
    return UINT_MAX; // conversion exceeds int's range
}
```

Aside: Avoid Undefined Behavior

- Just say no!



Microsoft | Dev Blogs | Developer | Technology | Languages | .NET | Platform Development | Data Development

Dev Blogs > The Old New Thing > Undefined behavior can result in time travel (among other things, but time travel is the funkiest)

June 27th, 2014

Undefined behavior can result in time travel (among other things, but time travel is the funkiest)

Raymond Chen

The C and C++ languages are notorious for the very large section of the map labeled *here be dragons*, or more formally, *undefined behavior*.

When undefined behavior is invoked, anything is possible. or example, a variable can be both true and false. John Regehr has a [list of interesting examples](#), as well as some [winners](#) of the ensuing contest.

<https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633>

The LLVM Project Blog

LLVM Project News and Details from the Trenches



About Posts Tags llvm.org

What Every C Programmer Should Know About Undefined Behavior #1/3

By Chris Lattner

May 13, 2011

[#optimization](#), [#Clang](#)

9 minute read

People occasionally ask why LLVM-compiled code sometimes generates SIGTRAP signals when the optimizer is turned on. After digging in, they find that Clang generated a "ud2" instruction (assuming X86 code) - the same as is generated by `__builtin_trap()`. There are several issues at work here, all centering around undefined behavior in C code and how LLVM handles it.

<https://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>

Aside: Avoid Implementation-Defined Behavior

- Standard library macros/types typically tell you the compiler's choice

```
#include <limits.h>
#include <stdint.h>
```

```
int16_t fixed(void)
{
    int16_t a = INT16_MIN;
    int16_t b = INT16_MAX;
    return a + b;
}
```

```
int native(void)
{
    int a = INT_MIN;
    int b = INT_MAX;
    return a + b;
}
```

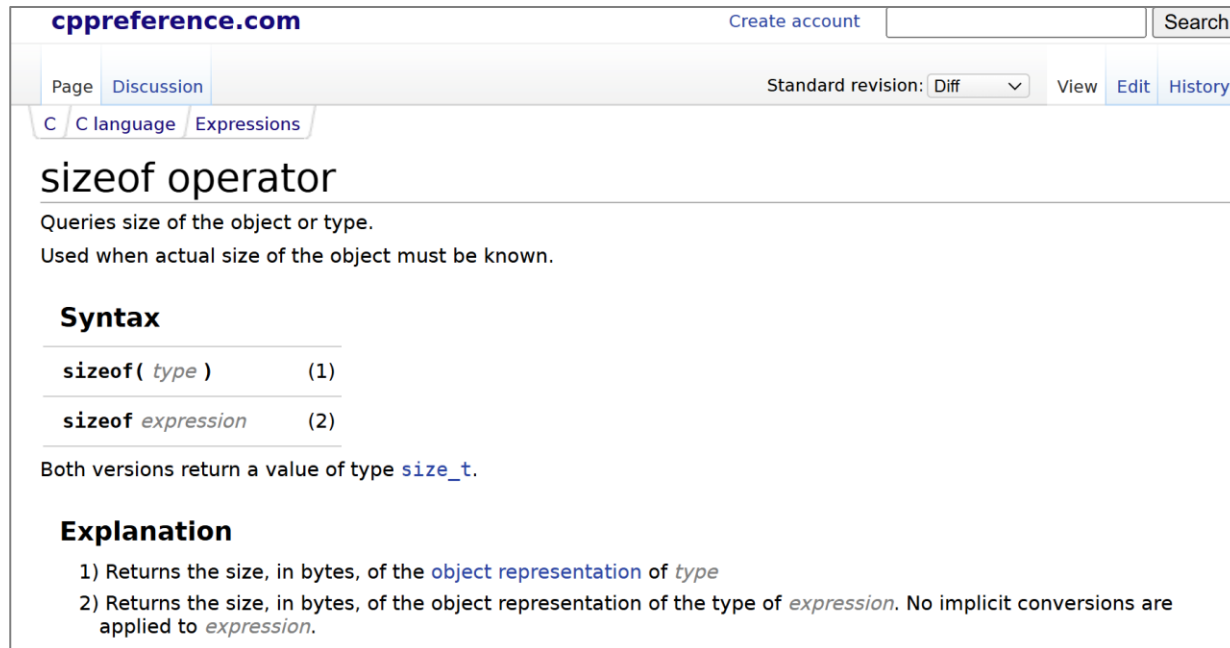
```
int impl_defined(void)
{
    int a = -2147483648;
    int b = 2147483647;
    return a + b;
}
```

Limits of integer types	
Limits of core language integer types Defined in header <limits.h>	
BOOL_WIDTH (C23)	bit width of <code>_Bool</code> (macro constant)
CHAR_BIT	bit width of byte (macro constant)
MB_LEN_MAX	maximum number of bytes in a multibyte character (macro constant)
CHAR_WIDTH (C23)	bit width of <code>char</code> , same as CHAR_BIT (macro constant)
CHAR_MIN	minimum value of <code>char</code> (macro constant)
CHAR_MAX	maximum value of <code>char</code> (macro constant)
SCHAR_WIDTH (C23) SHRT_WIDTH (C23) INT_WIDTH (C23) LONG_WIDTH (C23) LLONG_WIDTH (C23)	bit width of <code>signed char</code> , <code>short</code> , <code>int</code> , <code>long</code> , and <code>long long</code> respectively (macro constant)
SCHAR_MIN SHRT_MIN INT_MIN LONG_MIN LLONG_MIN (C99)	minimum value of <code>signed char</code> , <code>short</code> , <code>int</code> , <code>long</code> and <code>long long</code> respectively (macro constant)
SCHAR_MAX SHRT_MAX INT_MAX LONG_MAX LLONG_MAX (C99)	maximum value of <code>signed char</code> , <code>short</code> , <code>int</code> , <code>long</code> and <code>long long</code> respectively (macro constant)
UCHAR_WIDTH (C23) USHRT_WIDTH (C23) UINT_WIDTH (C23) ULONG_WIDTH (C23) ULLONG_WIDTH (C23)	bit width of <code>unsigned char</code> , <code>unsigned short</code> , <code>unsigned int</code> , <code>unsigned long</code> , and <code>unsigned long long</code> respectively (macro constant)
UCHAR_MAX USHRT_MAX UINT_MAX ULONG_MAX ULLONG_MAX (C99)	maximum value of <code>unsigned char</code> , <code>unsigned short</code> , <code>unsigned int</code> , <code>unsigned long</code> and <code>unsigned long long</code> respectively (macro constant)

Fixed width integer types (since C99)	
Types Defined in header <stdint.h>	
int8_t int16_t int32_t int64_t	signed integer type with width of exactly 8, 16, 32 and 64 bits respectively with no padding bits and using 2's complement for negative values (provided only if the implementation directly supports the type)
int_fast8_t int_fast16_t int_fast32_t int_fast64_t	fastest signed integer type with width of at least 8, 16, 32 and 64 bits respectively
int_least8_t int_least16_t int_least32_t int_least64_t	smallest signed integer type with width of at least 8, 16, 32 and 64 bits respectively
intmax_t	maximum width integer type
intptr_t	integer type capable of holding a pointer
uint8_t uint16_t uint32_t uint64_t	unsigned integer type with width of exactly 8, 16, 32 and 64 bits respectively (provided only if the implementation directly supports the type)
uint_fast8_t uint_fast16_t uint_fast32_t uint_fast64_t	fastest unsigned integer type with width of at least 8, 16, 32 and 64 bits respectively
uint_least8_t uint_least16_t uint_least32_t uint_least64_t	smallest unsigned integer type with width of at least 8, 16, 32 and 64 bits respectively
uintmax_t	maximum width unsigned integer type
uintptr_t	unsigned integer type capable of holding a pointer
The implementation may define typedef names <code>intN_t</code> , <code>int_fastN_t</code> , <code>int_leastN_t</code> , <code>uintN_t</code> , <code>uint_fastN_t</code> , and <code>uint_leastN_t</code> when <code>N</code> is not 8, 16, 32 or 64. Typedef names of the form <code>intN_t</code> may only be defined if the implementation supports an integer type of that width with no padding. Thus, <code>uint24_t</code> denotes an unsigned integer type with a width of exactly 24 bits.	
Each of the macros listed in below is defined if and only if the implementation defines the corresponding typedef name. The macros <code>INTN_C</code> and <code>UINTN_C</code> correspond to the typedef names <code>int_leastN_t</code> and <code>uint_leastN_t</code> , respectively.	
Macro constants Defined in header <stdint.h>	
Signed integers : width	
INT8_WIDTH INT16_WIDTH INT32_WIDTH (C23)(optional) INT64_WIDTH	bit width of an object of type <code>int8_t</code> , <code>int16_t</code> , <code>int32_t</code> , <code>int64_t</code> (exactly 8, 16, 32, 64) (macro constant)
INT_FAST8_WIDTH INT_FAST16_WIDTH INT_FAST32_WIDTH (C23) INT_FAST64_WIDTH	bit width of an object of type <code>int_fast8_t</code> , <code>int_fast16_t</code> , <code>int_fast32_t</code> , <code>int_fast64_t</code> (macro constant)
INT_LEAST8_WIDTH INT_LEAST16_WIDTH INT_LEAST32_WIDTH (C23) INT_LEAST64_WIDTH	bit width of an object of type <code>int_least8_t</code> , <code>int_least16_t</code> , <code>int_least32_t</code> , <code>int_least64_t</code> (macro constant)
INTPTR_WIDTH (C23)(optional)	bit width of an object of type <code>intptr_t</code> (macro constant)
INTMAX_WIDTH (C23)	bit width of an object of type <code>intmax_t</code> (macro constant)

Aside: Avoid Implementation-Defined Behavior

- If you really can't use `<stdint.h>`'s fixed-width types, check sizes!



The screenshot shows the cppreference.com website page for the `sizeof` operator. The page title is "sizeof operator". Below the title, it states: "Queries size of the object or type. Used when actual size of the object must be known." Under the "Syntax" section, two forms are listed: `sizeof(type)` (1) and `sizeof expression` (2). Below the syntax, it says: "Both versions return a value of type `size_t`." Under the "Explanation" section, two points are listed: 1) Returns the size, in bytes, of the object representation of `type`; 2) Returns the size, in bytes, of the object representation of the type of `expression`. No implicit conversions are applied to `expression`.

```
sizeof(char) = 1;
```

```
sizeof(unsigned int) = 4;
```

```
sizeof(long int) = 8;
```

```
sizeof("test") = 5; // type is char[5]
```

```
sizeof(int[10]) = 40;
```

Signed Integer Representations

- void
- basic types
 - char
 - **signed integers**
 - unsigned integers
 - floating-point
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

... If the sign bit is one, the value shall be modified in one of the following ways:

- the corresponding value with sign bit 0 is negated (*sign and magnitude*);
- the sign bit has the value $-(2^M)$ (*two's complement*);
- the sign bit has the value $-(2^M - 1)$ (*ones' complement*).

Which of these applies is implementation-defined,

ISO Standard 9899:201x

```
int function(void)
{
    int a = -1;
    int b = 1;
    return a + b; // 1 - 1
}
```

Two's Complement

0xffff_ffff
+
0x0000_0001
=
0x0000_0000

Sign-Magnitude

0x1000_0001
+
0x0000_0001
=
0x0000_0000

Unsigned Integer Representations

- void
- basic types
 - char
 - signed integers
 - **unsigned integers**
 - floating-point
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

6.2.6.2 Integer types

- 1 For unsigned integer types other than **unsigned char**, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1} , so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ **using a pure binary representation** this shall be known as the value representation. The values of any padding bits are unspecified.⁵³⁾

Values stored in unsigned bit-fields and objects of type **unsigned char** shall be represented using a **pure binary notation**.⁴⁹⁾

ISO Standard 9899:201x

“Pure Binary Notation”

```
uint64_t function(void)
{
    uint64_t a = ULONG_MAX;
    uint64_t b = 1;
    return a + b; // overflow
}
```

0xffff_ffff_ffff_ffff

+

0x0000_0000_0000_0001

=

0x0000_0000_0000_0000

C's Type System

- void
- basic types
 - char
 - signed integers
 - unsigned integers
 - **floating-point**
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

10 There are three *real floating types*, designated as **float**, **double**, and **long double**.⁴²⁾ The set of values of the type **float** is a subset of the set of values of the type **double**; the set of values of the type **double** is a subset of the set of values of the type **long double**.

ISO Standard 9899:201x

Standard floating-point types

The following three types and their cv-qualified versions are collectively called standard floating-point types.

float — single precision floating-point type. Usually IEEE-754 binary32 format [↗](#).

double — double precision floating-point type. Usually IEEE-754 binary64 format [↗](#).

long double — extended precision floating-point type. Does not necessarily map to types mandated by IEEE-754.

- IEEE-754 binary128 format [↗](#) is used by some HP-UX, SPARC, MIPS, ARM64, and z/OS implementations.
- The most well known IEEE-754 binary64-extended format [↗](#) is x87 80-bit extended precision format [↗](#). It is used by many x86 and x86-64 implementations (a notable exception is MSVC, which implements **long double** in the same format as **double**, i.e. binary64).
- On PowerPC **double-double** [↗](#) can be used.

<https://en.cppreference.com/w/cpp/language/types>

Boolean (bool)

- `void`
- **basic types**
 - `char`
 - **signed integers**
 - **unsigned integers**
 - **floating-point**
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

- C has no built-in “Boolean” type

Recall:

Memory is **byte addressable**

<code>addr</code>	<code>0b0000_0001</code>	} Every address has 8 bits
<code>addr + 1</code>	<code>0b0000_0001</code>	
<code>addr + 2</code>	<code>0b0000_0001</code>	

- C only has values that are “**zero**” vs “**not zero**”

```
/* 0 == FALSE */
if(0) {}
if(!42) {}
if(6 - 6) {}
if(2 == 9) {}
if(8 < 1) {}
...
```

```
/* !0 == TRUE */
if(-1) {}
if(42) {}
if(!0) {}
if(3 == 3) {}
if(1 < 8) {}
if('a') {}
...
```

Boolean (bool)

- `void`
- **basic types**
 - `char`
 - **signed integers**
 - **unsigned integers**
 - **floating-point**
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

- `<stdbool.h>` defines a “Boolean” you can use

6.3.1.2 Boolean type

1 When any scalar value is converted to `_Bool`, the result is 0 if the value compares equal to 0; otherwise, the result is 1.⁵⁹⁾

2 An object declared as type `_Bool` is large enough to store the values 0 and 1.

ISO 9899:2011

<stdbool.h>

```
36 #define bool _Bool
37 #define true 1
38 #define false 0
```

<https://github.com/gcc-mirror/gcc/blob/master/gcc/ginclude/stdbool.h>

```
#include <stdbool.h>

void func(void)
{
    bool a = true;
}
```



`&a`
`&a+1` 0b0000_0001

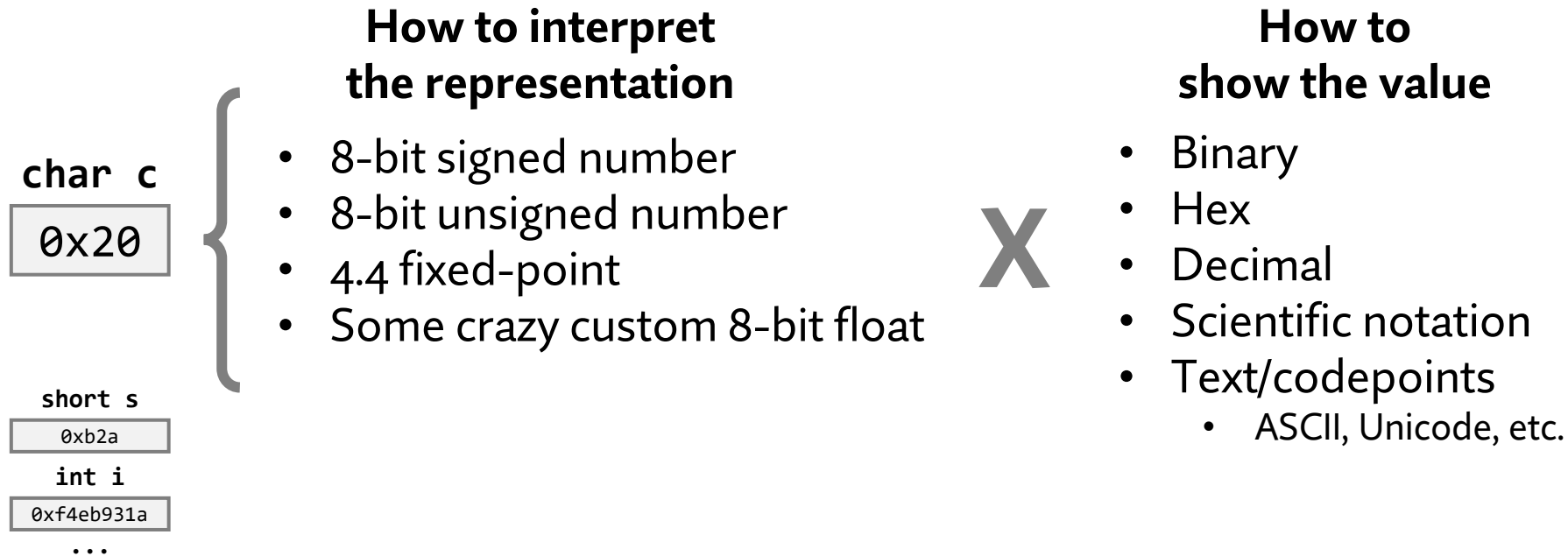
Basic Types: Agenda

- void
- **basic types**
 - char
 - signed integers
 - unsigned integers
 - floating-point
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

- Overview
- **Format strings**
- Initialization
- Arithmetic
- Type casting

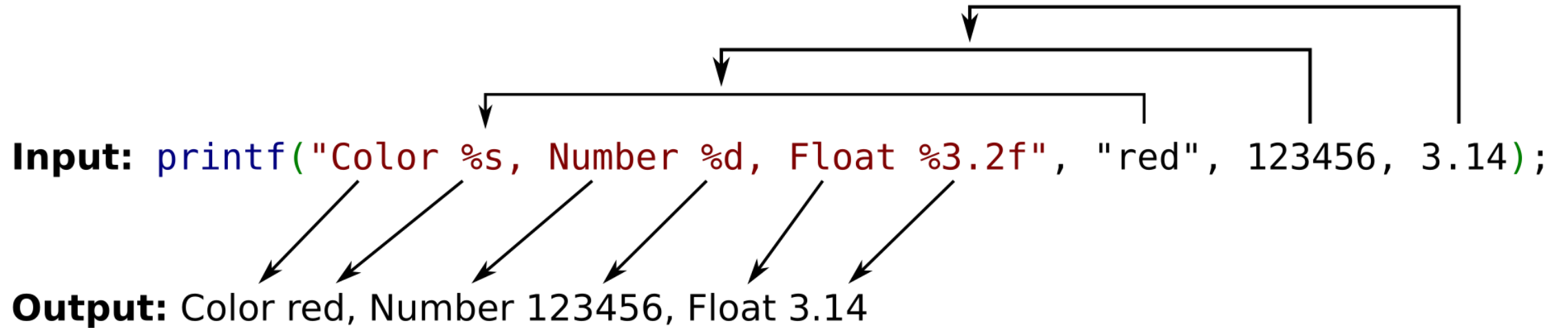
Printing Objects' Values

- Printing objects requires converting **representations** to **encoded text**
- **Imagine doing this manually...**



- Instead, the C standard library provides **one highly-configurable function**


```
int printf(const char *format, ...);
```



Format Strings

- It's worth learning the `printf()` family of functions



die.net

Site Search

Library

- linux docs
- linux man pages
- page load time

Toys

- world sunlight
- moon phase
- trace explorer

printf(3) - Linux man page

Name

`printf`, `fprintf`, `sprintf`, `snprintf`, `vprintf`, `vfprintf`, `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, ...);
```

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

Output to the terminal

Output to a file

} Output to a string

Let's take a look at format strings

Format Strings

- Specific “format specifiers” for every type you’d like to print

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

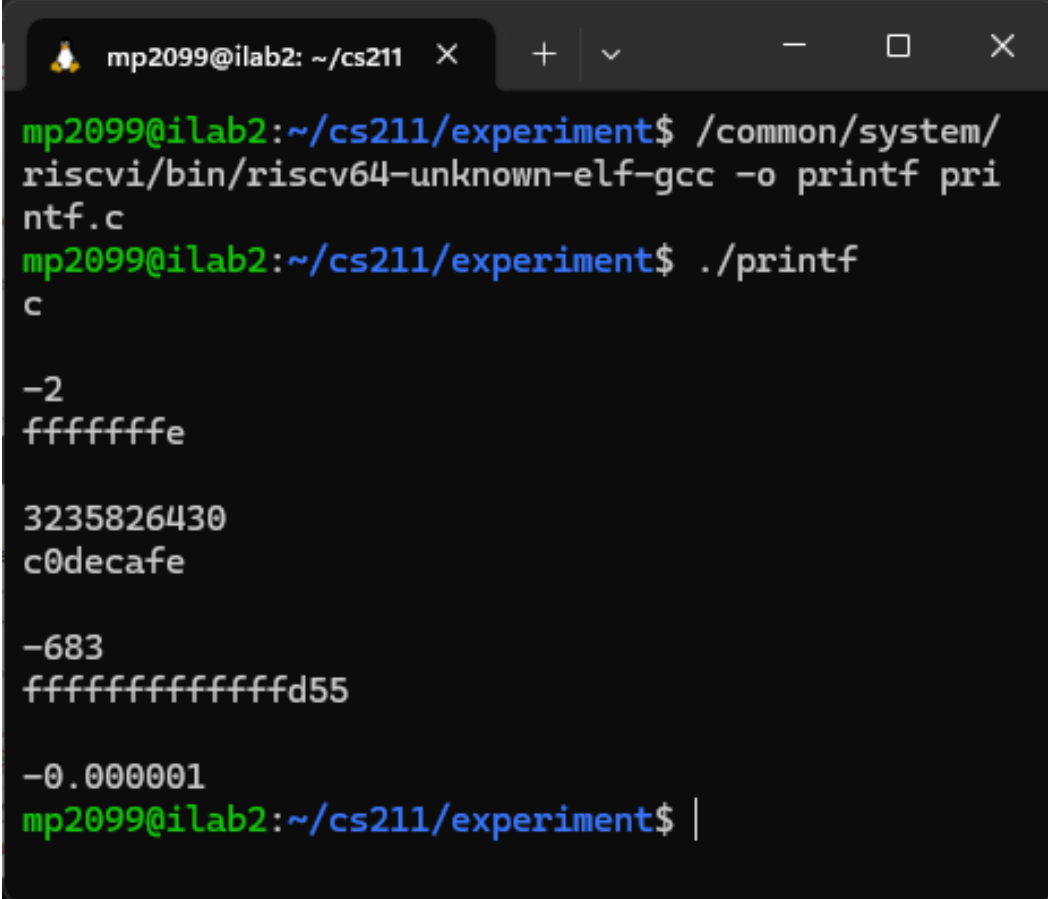
int main(int argc, char *argv[])
{
    char c = 'c';
    printf("%c\n\n", c);

    signed int s = -0x2;
    printf("%d\n", s);
    printf("%x\n\n", s);

    unsigned int u = 0xc0decafe;
    printf("%u\n", u);
    printf("%x\n\n", u);

    int64_t i = -683;
    printf("%ld\n", i);
    printf("%lx\n\n", i);

    float f = -0.000001;
    printf("%f\n", f);
}
```

A terminal window showing the compilation and execution of a C program. The user is in a directory ~/cs211/experiment. They compile printf.c using the riscv64-unknown-elf-gcc compiler. Then they run the resulting printf executable. The output shows the values of various variables: a character 'c', a signed integer -2 (hex 0xffffffe), an unsigned integer 3235826430 (hex c0decafe), a 64-bit integer -683 (hex ffffffff55), and a float -0.000001.

```
mp2099@ilab2: ~/cs211
mp2099@ilab2:~/cs211/experiment$ /common/system/riscvi/bin/riscv64-unknown-elf-gcc -o printf printf.c
mp2099@ilab2:~/cs211/experiment$ ./printf
c

-2
fffffffe

3235826430
c0decafe

-683
ffffffffffffffd55

-0.000001
mp2099@ilab2:~/cs211/experiment$
```

Format Specifiers

- Just look up the specifiers and eventually you'll learn the basic ones

specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

<https://cplusplus.com/reference/cstdio/printf/>

Basic Types: Agenda

- void
- **basic types**
 - char
 - signed integers
 - unsigned integers
 - floating-point
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

- Overview
- Format strings
- **Initialization**
- Arithmetic
- Type casting

Undefined Default Value

- C objects have **no default value**
 - **Undefined (garbage)** until explicitly initialized
 - Basically, whatever was leftover in memory
- But... C will let you use uninitialized objects 😞

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

#define ARRAY_SIZE 10

int main(int argc, char *argv[])
{
    long array[ARRAY_SIZE];
    for(int i = 0; i < ARRAY_SIZE; i++)
        printf("array[%d] = 0x%lx\n", i, array[i]);
    return EXIT_SUCCESS;
}
```

```
mp2099@iLab3:~/cs211/experiment$ ./uninitialized
array[0] = 0x555f5bfa5040
array[1] = 0x7f30aea9c83c
array[2] = 0x6f0
array[3] = 0x7ffe307a7629
array[4] = 0x7ffe307ef000
array[5] = 0x10101000000
array[6] = 0x2
array[7] = 0x178bfbff
array[8] = 0x7ffe307a7639
array[9] = 0x64
mp2099@iLab3:~/cs211/experiment$ ./uninitialized
array[0] = 0x5629b6db7040
array[1] = 0x7fd8d09d483c
array[2] = 0x6f0
array[3] = 0x7fff3fe54119
array[4] = 0x7fff3ffeb000
array[5] = 0x10101000000
array[6] = 0x2
array[7] = 0x178bfbff
array[8] = 0x7fff3fe54129
array[9] = 0x64
mp2099@iLab3:~/cs211/experiment$ ./uninitialized
array[0] = 0x561eb4533040
array[1] = 0x7f3f3a5cf83c
array[2] = 0x6f0
array[3] = 0x7fff61fc25a9
array[4] = 0x7fff61fe5000
array[5] = 0x10101000000
array[6] = 0x2
array[7] = 0x178bfbff
array[8] = 0x7fff61fc25b9
array[9] = 0x64
mp2099@iLab3:~/cs211/experiment$ |
```

Initialization: Different Bases

- Integer constants can be in:
 - Base 8: `0123` // octal
 - Base 10: `123` // decimal
 - Base 16: `0x123` // hexadecimal

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int array[4] = {123, 0123, 0x123, -0x123};
    for(int i = 0; i < 4; i++)
        printf("%d : %x\n", array[i], array[i]);
    return EXIT_SUCCESS;
}
```

```
mp2099@ilab3:~/cs211/experiment$ /common/system/riscvi/
bin/riscv64-unknown-elf-gcc -o constant constant.c
mp2099@ilab3:~/cs211/experiment$ ./constant
123 : 7b
83 : 53
291 : 123
-291 : fffffedd
mp2099@ilab3:~/cs211/experiment$ |
```

Initialization: Constant Types

- Constants are **positive integers** by default (even character literals, e.g., ‘a’)
 - If you want anything else, you need to provide a suffix
 - Expressions such as “-1” apply the minus operator to the integer value

Bunch of rules

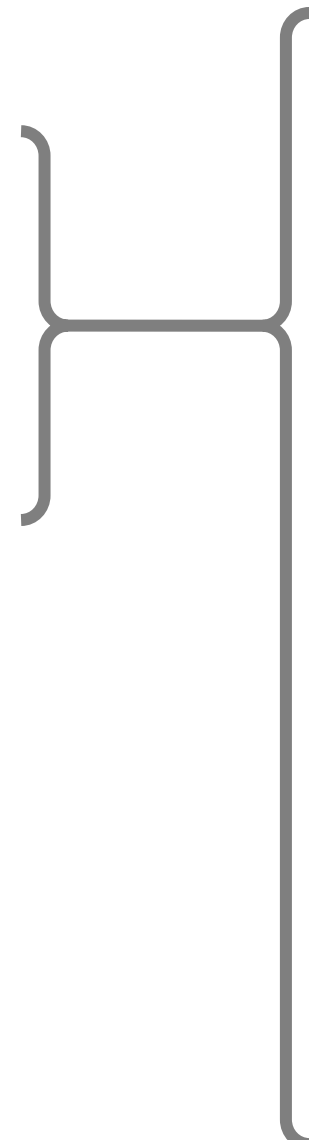
Types allowed for integer constants		
suffix	decimal bases	other bases
no suffix	<code>int</code> <code>long int</code> <code>unsigned long int</code> (until C99) <code>long long int</code> (since C99)	<code>int</code> <code>unsigned int</code> <code>long int</code> <code>unsigned long int</code> <code>long long int</code> (since C99) <code>unsigned long long int</code> (since C99)
u or U	<code>unsigned int</code> <code>unsigned long int</code> <code>unsigned long long int</code> (since C99)	<code>unsigned int</code> <code>unsigned long int</code> <code>unsigned long long int</code> (since C99)
l or L	<code>long int</code> <code>unsigned long int</code> (until C99) <code>long long int</code> (since C99)	<code>long int</code> <code>unsigned long int</code> <code>long long int</code> (since C99) <code>unsigned long long int</code> (since C99)
both l/L and u/U	<code>unsigned long int</code> <code>unsigned long long int</code> (since C99)	<code>unsigned long int</code> <code>unsigned long long int</code> (since C99)
ll or LL	<code>long long int</code> (since C99)	<code>long long int</code> (since C99) <code>unsigned long long int</code> (since C99)
both ll/LL and u/U	<code>unsigned long long int</code> (since C99)	<code>unsigned long long int</code> (since C99)

https://en.cppreference.com/w/c/language/integer_constant

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

int main(int argc, char *argv[])
{
    int i = 1;
    unsigned int ui = 1u;
    long int l = 1l;
    unsigned long int ul = 1ul;
    long long int ll = 1ll;
    unsigned long long int ull = 1ull;
    return EXIT_SUCCESS;
}
```


Basic Types: Agenda

- void
 - **basic types**
 - char
 - signed integers
 - unsigned integers
 - floating-point
 - enumerated types
 - derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers
- 
- Overview
 - Format strings
 - Initialization
 - **Arithmetic**
 - Type casting

Operations on Basic Types

- void
- basic types
 - char
 - signed integers
 - unsigned integers
 - floating-point
- enumerated types
- derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers

cppreference.com Create account

Page Discussion Standard

C C language Expressions

Arithmetic operators

Arithmetic operators apply standard mathematical operations to their operands.

This section is incomplete
Reason: consider a more general-purpose ToC for this and other tables that cover multiple topics

Operator	Operator name	Example	Result
<code>+</code>	unary plus	<code>+a</code>	the value of <code>a</code> after promotions
<code>-</code>	unary minus	<code>-a</code>	the negative of <code>a</code>
<code>+</code>	addition	<code>a + b</code>	the addition of <code>a</code> and <code>b</code>
<code>-</code>	subtraction	<code>a - b</code>	the subtraction of <code>b</code> from <code>a</code>
<code>*</code>	product	<code>a * b</code>	the product of <code>a</code> and <code>b</code>
<code>/</code>	division	<code>a / b</code>	the division of <code>a</code> by <code>b</code>
<code>%</code>	remainder	<code>a % b</code>	the remainder of <code>a</code> divided by <code>b</code>
<code>~</code>	bitwise NOT	<code>~a</code>	the bitwise NOT of <code>a</code>
<code>&</code>	bitwise AND	<code>a & b</code>	the bitwise AND of <code>a</code> and <code>b</code>
<code> </code>	bitwise OR	<code>a b</code>	the bitwise OR of <code>a</code> and <code>b</code>
<code>^</code>	bitwise XOR	<code>a ^ b</code>	the bitwise XOR of <code>a</code> and <code>b</code>
<code><<</code>	bitwise left shift	<code>a << b</code>	<code>a</code> left shifted by <code>b</code>
<code>>></code>	bitwise right shift	<code>a >> b</code>	<code>a</code> right shifted by <code>b</code>

Common arithmetic

Bitwise arithmetic

Arithmetic

- We are operating on **N-bit** numbers: all arithmetic is **modulo 2^N**

uint8_t: 129
int8_t: -127

1000_0001

+

uint8_t: 127
int8_t: 127

0111_1111

=

0000_0000

uint8_t: 129 + 127 = 256 % 256
int8_t: -127 + 127 = 0 % 256

1000_0001

-

0111_1111

=

0000_0010

uint8_t: 129 - 127 = 2 % 256
int8_t: -127 - 127 = -254 % 256

1000_0001

x

0111_1111

=

1111_1111

uint8_t: 129 * 127 = 16383 % 256
int8_t: -127 * 127 = -16129 % 256

signed overflow = undefined behavior

1000_0001

/

0111_1111

=

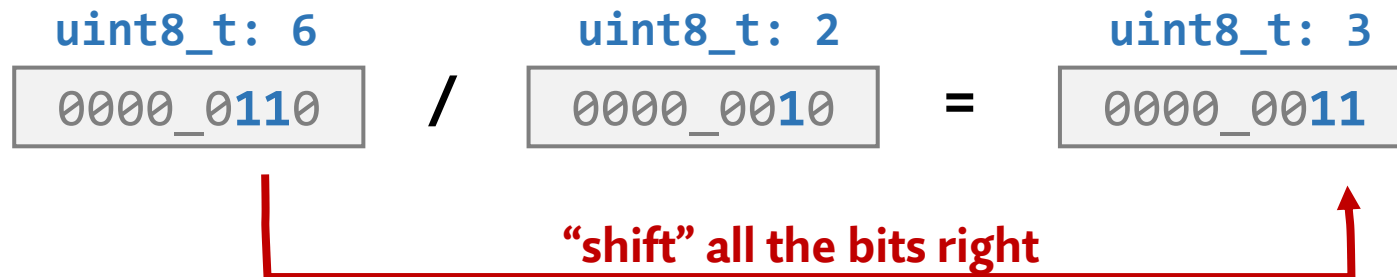
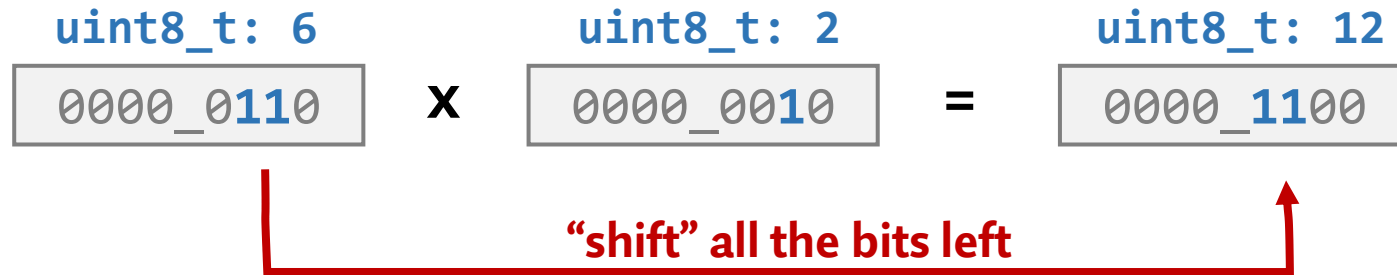
0000_0001

ffff_ffff

truncation (integer division)
uint8_t: 129 / 127 = 1.^{015748...}
int8_t: -127 / 127 = -1.0

Bitwise Operations: Bit Shift

- Sometimes, we want to manipulate bits directly



```
(6 * 2) == (6 << 1)
(6 * 4) == (6 << 2)
...
```

```
(6 / 2) == (6 >> 1)
(6 / 4) == (6 >> 2)
...
```

- **Unsigned values only!** Shifting a signed type with a negative value:
 - << is **undefined**
 - >> is **implementation defined**

Bitwise Operations: Bit-Parallel Logic

- We can use Boolean algebra operators bit-by-bit

AND

```
0 & 0 = 0
1 & 0 = 0
0 & 1 = 0
1 & 1 = 1
```

OR

```
0 | 0 = 0
1 | 0 = 1
0 | 1 = 1
1 | 1 = 1
```

XOR

```
0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 1
1 ^ 1 = 0
```

NOT

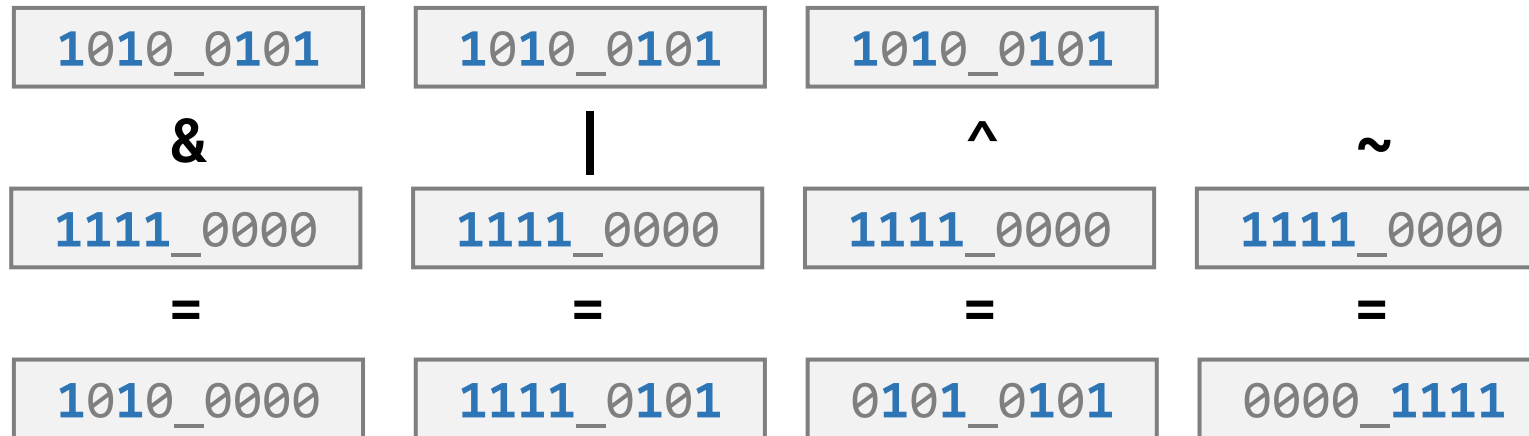
```
~0 = 1
~1 = 0
```

Logical

```
0xa5 && 0xf0 = 1
0xa5 || 0xf0 = 1
!0xf0 = 0
```

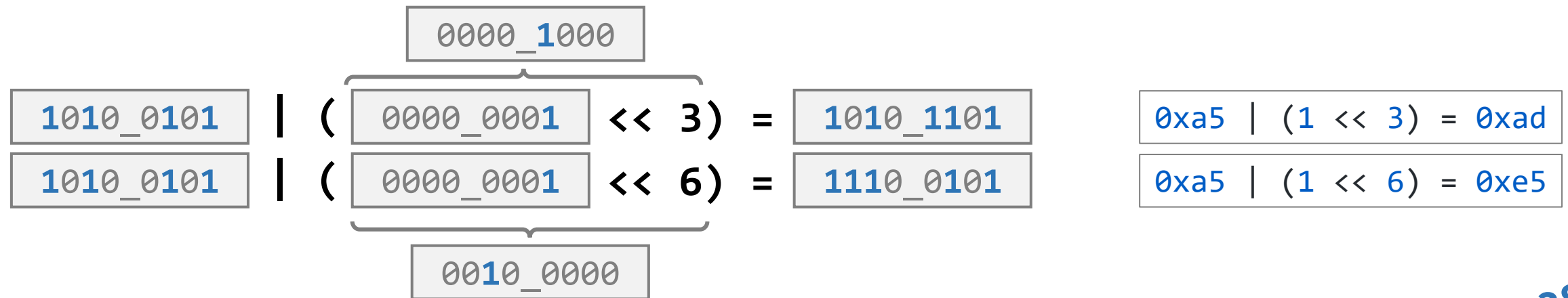
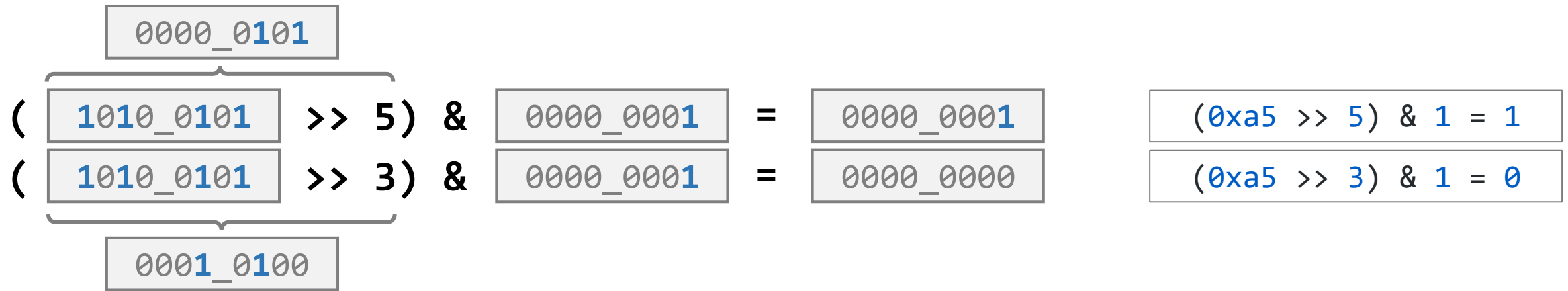
Bitwise

```
0xa5 & 0xf0 = 0xa0
0xa5 | 0xf0 = 0xf5
0xa5 ^ 0xf0 = 0x55
~ 0xf0 = 0x0f
```

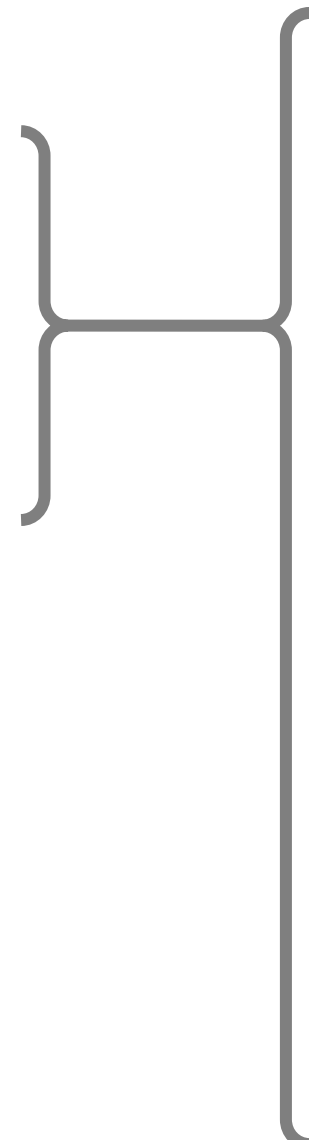


Bitwise Operations: Bit Masking

- We can **get / set** specific bits by combining shift/and/or



Basic Types: Agenda

- void
 - **basic types**
 - char
 - signed integers
 - unsigned integers
 - floating-point
 - enumerated types
 - derived types
 - arrays
 - structures
 - unions
 - functions
 - pointers
- 
- Overview
 - Format strings
 - Initialization
 - Arithmetic
 - **Type casting**

Integer Type Casting

- Convert one type to another

```
uint32_t x = 3;  
uint64_t y = (uint64_t)x;
```

```
&x 0x0000_0003  
&y 0x0000_0000_0000_0003
```

```
int32_t x = INT32_MIN;  
int64_t y = (int64_t)x;
```

```
&x 0x8000_0000  
&y 0xffff_ffff_8000_0000
```


Implicit Casting

- **Recall:** literals have default type “unsigned int”
- The compiler **implicitly casts** for you as needed

`uint64_t x = 3;` \longrightarrow `uint64_t y = (uint64_t)3;`

- This can cause **undefined behavior**

`uint64_t x = (1 << 40);` \longrightarrow `uint64_t x = (uint64_t)(1 << 40);`

↓ fix

`uint64_t x = (1u << 40);`

`uint64_t x = ((uint64_t)1 << 40);`

undefined behavior:
shifting beyond object width

Integer Casting Rules (1/2): Representable

- If the new type **can represent** the value, the **value is preserved**

<code>int32_t x = 3;</code>	<code>&x</code>	<code>0x0000_0003</code>
<code>uint64_t y = (uint64_t)x;</code>	<code>&y</code>	<code>0x0000_0000_0000_0003</code>
<code>int32_t x = -3;</code>	<code>&x</code>	<code>0xffff_fffd</code>
<code>int64_t y = (int64_t)x;</code>	<code>&y</code>	<code>0xffff_ffff_ffff_fff3</code>
<code>int64_t x = -3;</code>	<code>&x</code>	<code>0xffff_ffff_ffff_fff3</code>
<code>int32_t y = (int32_t)x;</code>	<code>&y</code>	<code>0xffff_fff3</code>
<code>uint8_t x = 253;</code>	<code>&x</code>	<code>0xfd</code>
<code>uint64_t y = (uint64_t)x;</code>	<code>&y</code>	<code>0x0000_0000_0000_00fd</code>

Integer Casting Rules (2/2): Not Representable

- If the new type **canNOT** represent the value:

Unsigned + Same/Narrower: Truncate

```
int32_t x = INT_MAX;  
uint8_t y = (uint8_t)x;
```

```
&x 0x7fff_ffff  
&y 0xff
```

```
int8_t x = -3_MAX;  
uint8_t y = (uint8_t)x;
```

```
&x 0xfd  
&y 0xfd
```

Signed: Implementation Defined

Don't do this 😊

Unsigned + Wider: Sign Extend

```
int8_t x = -3;  
uint32_t y = (uint32_t)x;
```

```
&x 0xfd  
&y 0xffff_fffd
```

Next Week

- Derived Types
- Expressions, operations, and constants

CS 211: Intro to Computer Architecture

5.1: C Data Representations

Minesh Patel

Spring 2025 – Tuesday 18 February