# Lecture 13: C23

- `#define __STDC_VERSION__   202311L`
- Official name ISO/IEC 9899:2024
- C23 is an informal name and there is only one ISO C at a time
- The next revision is expected around 2030
- Enable with `gcc -std=c23` or next with `gcc -std=c2y`

# Signed integers and removed features

- Now all signed integers are represented with two's complement
- Trigraphs are removed
- K&R function definitions are removed:

```
main(argc,argv) /* default type is int */
char** argv;
{
}
```

- `void f()` now means no parameters

# Constants

- `0b` and `0B` prefixes
- `0b1001 = 9`
- A digit separator `'` (can be put anywhere in a number)
- `123'456'78 = 12345678`
- `0b1111'1101'11'111'001`

# Compound literal

- Storage class specifier in compound literal

```
struct s { int a; };

struct s* s = &(static struct s) { 0 };
```

# memset_explicit

- In <string.h>
- `memset_explicit(void* p, int c, size_t n)`
- Set first `n` bytes of what `p` points to `(unsigned char)c`.
- Similar to normal `memset` except that it must be performed and never be optimized away.
- Avoiding `memset` below "cannot" affect program output :-)

```
memset(p, 0xa3, n); /* useful to crash buggy program. */
free(p);
```

# memccpy

- In <string.h>
- `memccpy(void* restrict s, void* restrict t, int c, size_t n)`
- Copy `n` bytes from `t` to `s` but stop after copying a `c` (converted to `unsigned char`)
- Similar to normal `memcpy` and also here `s` and `t` must not overlap

# strdup and strndup

- In <string.h>
- `char* strdup(const char* s)`
  `char* strndup(const char* s, size_t n)`
- Allocate memory with `malloc` and copy `s` to it and return the new copy
- At most `n` bytes are copied by `strndup`
- The returned string is null-terminated
- It originates from Berkeley's UNIX

## memalignment

- In <stdlib.h>
- `size_t memalignment(const void* p)`
- Return the alignment of a pointer
- For example: 24 `==` `0b11000` is aligned on 8
- It has three trailing zeros and $2^3 = 8$
- It can be implemented as:

```
size_t memalignment(const void* p)
{
        size_t              a = (size_t)p;

        return a & -a;

        //      a = 11000 = 24
        //     -a = 00111 + 1 = 01000
        // a & -a = 11000 & 01000 = 8
}
```

# Type-generic macros in C99

- They check the size of the argument to conclude which type it is
- The purpose is to avoid specifying e.g. `f` or `lf` suffixes for `cos` in the source code

  ```
  #include <tgmath.h>
  float x, v;

  x = cos(v);                    // instead of x = cosf(v)
  ```
- No real language support in C but macros can be used
  - Check size of the argument
  - Select which function to call based on the size
  - Typically needs compiler support for a `typeof(expr)` operator
  - The typeof operator is now ISO C as we will see
- See `tgmath.h` in Gnu libc or Musl libc

# <stdbit.h>

- `stdc_count_ones(value)` // type-generic macro
- `stdc_count_ones_uc(unsigned char value)`
- `stdc_count_ones_us(unsigned short value)`
- `stdc_count_ones_ui(unsigned int value)`
- `stdc_count_ones_ul(unsigned long value)`
- `stdc_count_ones_ull(unsigned long long value)`
- Power instruction: `popcnt`

# <stdbit.h>

- `stdc_count_zeros(value)`
- `stdc_count_zeros_uc(unsigned char value)`
- `stdc_count_zeros_us(unsigned short value)`
- `stdc_count_zeros_ui(unsigned int value)`
- `stdc_count_zeros_ul(unsigned long value)`
- `stdc_count_zeros_ull(unsigned long long value)`

# <stdbit.h>

- `stdc_leading_zeros(value)`
- `stdc_leading_zeros_uc(unsigned char value)`
- `stdc_leading_zeros_us(unsigned short value)`
- `stdc_leading_zeros_ui(unsigned int value)`
- `stdc_leading_zeros_ul(unsigned long value)`
- `stdc_leading_zeros_ull(unsigned long long value)`
- Power instruction: `cntlz`

# <stdbit.h>

- `stdc_leading_ones(value)`
- `stdc_leading_ones_uc(unsigned char value)`
- `stdc_leading_ones_us(unsigned short value)`
- `stdc_leading_ones_ui(unsigned int value)`
- `stdc_leading_ones_ul(unsigned long value)`
- `stdc_leading_ones_ull(unsigned long long value)`

# <stdbit.h>

- `stdc_trailing_ones(value)`
- `stdc_trailing_ones_uc(unsigned char value)`
- `stdc_trailing_ones_us(unsigned short value)`
- `stdc_trailing_ones_ui(unsigned int value)`
- `stdc_trailing_ones_ul(unsigned long value)`
- `stdc_trailing_ones_ull(unsigned long long value)`

# <stdbit.h>

- `stdc_trailing_zeros(value)`
- `stdc_trailing_zeros_uc(unsigned char value)`
- `stdc_trailing_zeros_us(unsigned short value)`
- `stdc_trailing_zeros_ui(unsigned int value)`
- `stdc_trailing_zeros_ul(unsigned long value)`
- `stdc_trailing_zeros_ull(unsigned long long value)`
- Power instruction: `cnttz`

# <stdbit.h>

- Returns first one counting from left plus one or zero if none found
- `stdc_first_leading_one(value)`
- `stdc_first_leading_one_uc(unsigned char value)`
- `stdc_first_leading_one_us(unsigned short value)`
- `stdc_first_leading_one_ui(unsigned int value)`
- `stdc_first_leading_one_ul(unsigned long value)`
- `stdc_first_leading_one_ull(unsigned long long value)`

# <stdbit.h>

- Returns first zero counting from left plus one or zero if none found
- `stdc_first_leading_zero(value)`
- `stdc_first_leading_zero_uc(unsigned char value)`
- `stdc_first_leading_zero_us(unsigned short value)`
- `stdc_first_leading_zero_ui(unsigned int value)`
- `stdc_first_leading_zero_ul(unsigned long value)`
- `stdc_first_leading_zero_ull(unsigned long long value)`

## <stdbit.h>

- Returns first one counting from right plus one or zero if none found
- `stdc_first_trailing_one(value)`
- `stdc_first_trailing_one_uc(unsigned char value)`
- `stdc_first_trailing_one_us(unsigned short value)`
- `stdc_first_trailing_one_ui(unsigned int value)`
- `stdc_first_trailing_one_ul(unsigned long value)`
- `stdc_first_trailing_one_ull(unsigned long long value)`

# <stdbit.h>

- Returns first zero counting from right plus one or zero if none found
- `stdc_first_trailing_zero(value)`
- `stdc_first_trailing_zero_uc(unsigned char value)`
- `stdc_first_trailing_zero_us(unsigned short value)`
- `stdc_first_trailing_zero_ui(unsigned int value)`
- `stdc_first_trailing_zero_ul(unsigned long value)`
- `stdc_first_trailing_zero_ull(unsigned long long value)`

# <stdbit.h>

- Returns true if there is exactly one nonzero bit
- `stdc_has_single_bit(value)`
- `bool stdc_has_single_bit_uc(unsigned char value)`
- `bool stdc_has_single_bit_us(unsigned short value)`
- `bool stdc_has_single_bit_ui(unsigned int value)`
- `bool stdc_has_single_bit_ul(unsigned long value value)`
- `bool stdc_has_single_bit_ull(unsigned long long value)`

# <stdbit.h>

- Returns the largest power of two that is not greater than the value
- `stdc_bit_floor(value)`
- `stdc_bit_floor_uc(unsigned char value)`
- `stdc_bit_floor_us(unsigned short value)`
- `stdc_bit_floor_ui(unsigned int value)`
- `stdc_bit_floor_ul(unsigned long value)`
- `stdc_bit_floor_ull(unsigned long long value)`

# <stdbit.h>

- Returns the smallest power of two that is not less than the value
- `stdc_bit_ceil(value)`
- `stdc_bit_ceil_uc(unsigned char value)`
- `stdc_bit_ceil_us(unsigned short value)`
- `stdc_bit_ceil_ui(unsigned int value)`
- `stdc_bit_ceil_ul(unsigned long value)`
- `stdc_bit_ceil_ull(unsigned long long value)`

# <stdbit.h>

- Returns the smallest number of bits needed to store the value (or zero)
- Computed as: $\text{value} = 0\ ?\ 0 : 1 + \lfloor \log_2(\text{value}) \rfloor$
- `stdc_bit_width(value)`
- `stdc_bit_width_uc(unsigned char value)`
- `stdc_bit_width_us(unsigned short value)`
- `stdc_bit_width_ui(unsigned int value)`
- `stdc_bit_width_ul(unsigned long value)`
- `stdc_bit_width_ull(unsigned long long value)`

# <time.h>

- `time_t timegm(struct tm* t)`
- Previously available in Musl libc and Gnu libc
- It converts the time pointed to by the argument into a `time_t` representation

# `<math.h>`

- `double exp10(double x)`: $10^x$
- `double exp2(double x)`: $2^x$
- `double cospi(double x)`: computes $\cos(\pi x)$
- `double acospi(double x)`: computes $\text{acos}(x)/\pi$
- Similar functions for other trigonometric functions
- Math functions for decimal floating point types such as: `_Decimal32 cosd32(_Decimal32 x);`
- IBM has supported decimal floating point types for several years and open sourced an implementation called `decNumber`
- Used to improve accuracy of financial calculations and Power has instructions for this

# `<stdio.h>`

- `%b` conversion specifier for `printf` and `scanf` family of functions

# <stdlib.h>

- `0b` and `0B` in strings for `strtol` family of functions

# C Preprocessor

- `#elifdef`
- `#elifndef`

# C Preprocessor `__has_include`

- `__has_include("file.h")`
- `__has_include(<file.h>)`

```
#if __has_include("file.h")
#include "file.h"
#define we_found_file_h 1
#else
#define we_found_file_h 0
#endif
```

# C Preprocessor: #warning

- `#warning` has been supported by many compilers since before ANSI C
- Similar to `#error`
- The preprocessor prints what is in the line
- Compiler switches can of course make warnings be treated as errors

```
#error this must result in compilation failure
#warning check this out
```

## C Preprocessor: #embed — binary resource inclusion

- The *main purpose* is to take "binary" initializer expressions from a file
- We can do that with normal text files already with an `#include`

```
int a[] = {
#include "a.txt"
};
int b[] = {
#embed "b.bin"
};
```

- The effect is the same as using:

  ```
  int bb[1000];
  FILE* fp = fopen("b.bin", "r");
  fread(bb, 1000, sizeof(int), fp);
  ```

# C Preprocessor: `__has_embed`

- `__has_embed` works like `__has_include`

# C Preprocessor: #embed parameters: `limit(n)`

- Implementation resource width: number of bits in the resource
- Resource width: same as implementation resource width or can be found in a `limit` parameter
- `CHAR_BIT` is the number of bits in a `char` and is at least 8
- The embed element width is `CHAR_BIT`
- If `sizeof(int) == 4`, then this will read 5 ints:

```
#define B 20
int b[] = {
#embed "b.bin" limit(B)
};
```

- The limit can e.g. be used to read a part of an "infinite resource"

# C Preprocessor: #embed parameter suffix

- We can add normal initializer expressions after what was read

```
int b[] = {
#embed "b.bin" suffix(,1,2,3)
};
```

# C Preprocessor: `#embed` parameter `prefix`

- We can add normal initializer expressions before what was read

```
int b[] = {
#embed "b.bin" prefix(1,2,3,)
};
```

# C Preprocessor: `#embed` parameter `if_empty`

- `if_empty` has a token list which is used if the size of the embedded resource is zero
- It can be made empty by a zero limit.

# C Preprocessor: #embed not only for initializers

- #embed can be used outside initilizers

```
int main()
{
        // how to return one from main in a weird way
        return
#embed "unused.bin" limit(0) if_empty(1)
        ;
}
```

# typeof and typeof_unqual

```
int             a;
typeof(a)       b;
typedef float num;
typeof(num)     c;
num             d;    // usually better
```

- Instead of writing the type we can use `typeof` another variable or a type as with `sizeof`
- This has been supported by many C compilers for years and typically been called `__typeof__`
- Alternative: `typeof_unqual` excludes any type qualifier such as `const`

# nullptr and nullptr_t

- A null pointer is always unequal to any pointer to an object or function
- A null pointer constant is created by one of:
  - 0
  - (void*)0
  - nullptr
- nullptr is a predefined constant and a keyword (as opposed to NULL)

```
typedef typeof_unqual(nullptr) nullptr_t;
```

# Variably modified types is back in ISO C

- Variable length arrays are still optional (since C11)
- Note the difference between a VLA and a variably modified type

```
void f(int m, int n, int i, int j)
{
        int     a[n];      // VLA
        int     (*p)[n];   // not VLA

        p = calloc(m * n, sizeof(int));

        p[i][j] = 1;
}
```

# Bit-precise integers: `_BitInt(n)`

- `_BitInt(n)` — signed integer with $n \geq 2$
- `unsigned _BitInt(n)` — unsigned integer with $n \geq 1$
- `BITINT_MAXWIDTH` is at least what is needed for the type `unsigned long long`, i.e. 64, declared in `<limits.h>`
- Bit-precise integers are not integer promoted to `int` (if smaller than an `int`)

```
_BitInt(2)      a2;
_BitInt(3)      a3;
_BitInt(33)     a33;
int             c;

a2 * a3;  // a2 is converted to _BitInt(3) result is _BitInt(3)
a33 * c;  // c is converted to _BitInt(33) result is _BitInt(33)
```

# Bit-precise integers constants

```
1wb     // _BitInt(2)
1uwb    // unsigned _BitInt(1)
7wb     // _BitInt(4)
-1wb    // _BitInt(2)
-1uwb   // unsigned _BitInt(1) with value 1
```

# Checked integer arithmetic

- Optional feature:
  `#define __STDC_VERSION_STDCKDINT_H__ 202311L`
- In `<stdckdint.h>`
- Type generic macros
- `bool ckd_add(type1* result, type2 a, type3 b)`
- `bool ckd_sub(type1* result, type2 a, type3 b)`
- `bool ckd_mul(type1* result, type2 a, type3 b)`
- Operations are performed as if with infinite range
- Return value is zero if the exact result could be represented
- Otherwise the result is wrapped around (and still stored)
- Source operands can have any integer type except:
  `bool`, `char`, `_BitInt`, or enumerated type

# auto

- Old meaning of `auto` remains
- New meaning is to infer the type in a variable declaration
- Not for return or parameter types (as in C++)

```
auto a = 1;                       // int
auto b = 2.0;                     // double
auto c = &b;                      // double*
```

- Before the rule was that the enumeration constants must be representable as an `int`.
- Now we can specify an *underlying type*:

```
enum a : unsigned long long { X, Y, Z };
```

# `constexpr`

- Enums are limited to integer constants
- Non-integer constants typically are created with macros
- `constexpr` allows constant expressions of any data type
- In C only for variables and in C++ also for functions
- `constexpr` may appear with auto, register, or static

```
constexpr int a = 52;
constexpr auto b = 36;
static int c[a][b];
```

- Advantage over macros: normal C scope rules
- Advantage over macros: easier life for debuggers (but dwarf has support for macros since many years)
- But no *essential* difference as I see it.

# Summary of new keywords

- New keywords from macros (bool was a macro defined as `_Bool`)

```
true false bool nullptr
```

- New spelling of keywords:

```
alignas alignof bool static_assert thread_local
```

- New keywords

```
typeof typeof_unqual constexpr
_Decimal32 _Decimal64 _Decimal128
```

# Syntax changes

- Labels before declarations — before <id> ':' <stmt>

```
A:      int a; // ok in C23
        goto A;
```

- Label at end of `{ }` without semicolon

```
{                          {
B:   // ok in C23          B: ;       // before
}                          }
```

- Zero initialization with `int a = { }`
- Even for VLA `int a[m] = { }`
- Equivalent to: `int a[m]; memset(a, 0, sizeof a);`
- But normal initializer is still forbidden for VLA.

# Syntax changes

- Named parameter is no longer required before ellipses

```
void f(...)
{
        va_list ap;

        va_start(ap);
}
```

```
void f(int n, ...)
{
        va_list ap;

        va_start(ap, n);
}
```

- `va_start` does not need a parameter
- Single argument `static_assert(expression)` added

# Attribute syntax [[ ]]

- There are standard attributes and possibly implementation defined attribute
- The purpose of the standard attributes is to let the programmer give extra information to the compiler but the program must work properly even if the compiler ignores them completely
- Implementation defined attributes are not specified in the standard but can be vendor specific
- An example:

```
switch (x) {
case 1: f();
        [[fallthrough]];
case 2: g();
        break;
default:
        h();
}
```

# Standard attributes

- `deprecated` — e.g. warn about use of deprecated parts of an API
- `fallthrough` — previous slide
- `maybe_unused` — tell compiler to not warn about unused symbol
- `nodiscard` — tell compiler return value should be used
- `noreturn` — tell compiler the function will not return
- `_Noreturn` — tell compiler the function will not return
- `reproducible` — tell compiler some optimizations can be done
- `unsequenced` — tell compiler some optimizations can be done

# [[reproducible]]

- The following function `hash` must return the same value when called with identical input (in this case what the parameter points to).
- It may modify global state (variables, registers, OS kernel) if it restores the previous values.

```
size_t hash(const char* s) [[ reproducible ]];
```

- Possible optimizations for `[[ reproducible ]]`:
  - redundancy elimination
  - memoization, or
  - lazy evaluation

# [[unsequenced]]

- This attribute is for pure functions which only compute their value based on the parameters.
- This is stronger than reproducible and can allow optimizations in more situations.
- The following function `f` does not depend on any modifiable state and results in the same value whenever the argument is the same.
- `sqrt` can have side effects and be called with different rounding modes so in general it should not have `[[ unsequenced ]]`

```
int f(int x) [[ unsequenced ]];

double sqrt(double x);
```

- Possible optimizations for `[[ unsequenced ]]`:
  - redundancy elimination
  - memoization, or
  - lazy evaluation

# Remark on previous two slides

- A good optimizing C compiler normally can conclude `[[ reproducible ]]` and `[[ unsequenced ]]` by itself using link-time optimization if needed
- A reasonable use of them is for code that is not available as source
- Another is for very complicated functions (weird source)
- There exist weird C programs, as we will see next (from IOCCC)