Contents Lecture 3

- Arrays, matrices, and lists in C
- Introduction to the ISO C standard, Chapter 7
- Lexical elements, Chapter 8

Multidimensional arrays in C

- The language has no concept of multidimensional arrays.
- Instead you simply use arrays of arrays.

Arrays of arrays

```
double m[3][4];
double x[2][3][4][5];
```

- So `m` is an array with three elements, where each element is an array of four doubles.
- `x` has two elements.

Multidimensional arrays with calloc

- Suppose we want an $m \times n$ matrix from calloc. How do we do?
- A one-dimensional array is declared as: `double* a`.
- Here a is a pointer which points to the start of the calloc-ed memory.
- A two-dimensional matrix, can be declared as `double** m`.
- But how can we allocate memory for it???
- First allocate an array which can hold m pointers to the rows,
- and then allocate memory for each row.

More from previous slide

```
double** make_matrix(int m, int n)
{
        double**        a;
        int             i;

        a = calloc(m, sizeof(double*));
        for (i = 0; i < m; i += 1)
                a[i] = calloc(n, sizeof(double));
        return a;
}
```

- Now we can write `double** m = make_matrix(3, 4);`
- We can access the elements as `m[i][j]`.

Alternatives

- Instead of doing $m + 1$ calls to calloc, we can make one big:

```
double*          a = calloc(m * n, sizeof(double));
```

- Unfortunately, we cannot use it as a two-dimensional matrix. Assume we want `a[i][j]`:

```
for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
                a[ i * n + j] = ...
```

- The row number is determined by `i` and each row has `n` elements.
- We cannot write `a[i][j]` since the type of `a[i]` is a double and not an array.

malloc/calloc/realloc/free

- The data allocated by `void* calloc(size_t count, size_t size)` is initialized to zeroes.
- There is an alternative function `void* malloc(size_t size)` which leaves the data uninitialised.
- Using malloc but forgetting to initialize the data leads to painful bugs.
- You will often notice that the data is already zeroed by malloc but that is only by accident (by chance).
- The function `void* realloc(void* ptr, size_t size)` tries to extend (or shrink) the memory area pointed to by ptr, and if that is not possible it allocated new memory and copies to old content. Why can that be dangerous ?

If it fails to allocate a new block of memory due to fragmentation or insufficient memory, it reutrns null, and thus the original pointer ptr will not be freed leading to memory leak i.e. the memory block will be lost and not way to free it

Lists in C

- There are of course various kinds of lists, eg:
  - Single linked,
  - Single linked, with header pointing to the end (instead of having data).
  - Null terminated double linked,
  - Circular double linked.

An example circular double linked list

```
typedef struct list_t list_t;

struct list_t {
        list_t* succ;
        list_t* pred;
        void*   data;
};
```

- Without the typedef we must write `struct list_t` everywhere.
- By circular is meant that the head's predecessor points to the last node and the successor of the last node points to the head.

Making a list node


```
list_t* new_list(void* data)
{
        list_t*          list;

        list = malloc(sizeof(list_t));

        list->succ = list; // (*list).succ = list;
        list->pred = list; // (*list).pred = list;
        list->data = data; // (*list).data = data;

        return list;
}
```

- The arrow is a shorthand for `(*list).` and was added to C very early.

Freeing of a list Circular

```
void free_list(list_t** head)
{
        list_t*         h = *head;
        list_t*         p;
        list_t*         q;
        if (h == NULL)
                return;
        p = h->succ;
        while (p != h) {
                q = p->succ;
                free(p);
                p = q;
        }
        free(p);
        *head = NULL;
}
```

Copy the head to local var h

Pointer to traverse the list

temporary pointer to store next node

Check if empy

Init p to be the successful of the head

Traverse the list and free each node

Save the next node's pointer before freeing the curr

Free the current

Move to next node

Free the last node (original head) outside the loop

Set head to null to indicate an empty list

Comments on free

```
int*    a;
int*    b;
a = malloc(sizeof(int));
b = a;
free(a);
*a = 12; // wrong.
a;       // wrong.
b;       // wrong.
```

- After you have freed an object, any mention of that object is wrong, and the behavior is undefined. Anything is permitted to happen according to the C standard.

Iterating through a circular list

```
#include <stddef.h>

size_t length(list_t* head)
{
        size_t          count;      The count
        list_t*         p;          Local variable

        if (head == NULL)           Check if list is null
                return 0;
        count = 0;
        p = head;                   Create a local variable of the head
        do {
                count += 1;
                p = p->succ;        Move to next node
        } while (p != head);        While we haven't reached our destination
        return count;
}
```

Strings in C

- Strings are adjacent characters terminated with a 0.
- `"C is fun"` is a string and consists of 9 bytes.
- Eg `char v[10]` can hold a string.
- Eg `char* s` can point to a string — but it *is* no string.
- If we also do `s = malloc(10);` it is still no string.
- However, `s` points to memory which can hold a string.
- If we now do `s = "C is fun"` we just leak the 10 bytes from malloc

Each char is 1 byte, represented with 8 bits

Character arrays and string literals

```
char* s = "c is fun";
char  a[10] = "c is fun";   // 10 elements
char  b[]   = "c is fun";   // 9 elements
char  c[8]  = "c is fun";   // 8 elements but dangerous
```

- `s` points to a string literal so

  ```
  *s = 'C';
  ```

  is invalid since the string literal is read-only
- `a` to `c` are normal arrays so we can modify them
- For `a` to `c` the strings are really just used to inform the compiler what the arrays should be initialized to and are not needed in the program
- The array `c` will contain no terminating zero byte

Copying a string

- To make a copy of a string, we can use the following function.
- The type `size_t` is an unsigned integer type, e.g. `unsigned int` or `unsigned long`.
- `size_t` is defined in `stdio.h`, `stdlib.h` and `stddef.h`

Only neutral numbers

```
char* copy_string(char* s)
{
        size_t  length;
        char*   t;

        length = strlen(s);
        t = malloc(length + 1); // why + 1 ???
        if (t != NULL)
                strcpy(t, s);   // library function
        return t;
}
```

Store the length of the input string s

Pointer to store the copy of the string

Total amount needed to store the string + null character

Check if the allocation was successful, if not, proceed to copy

strcopy copies the content of string s to the newly allocated memory t

`size_t strlen(const char* s);`

Calculates the length of a null-terminated string

- `const` means this function promises not to modify what `s` points to.

```
size_t strlen(const char* s)
{
        size_t  length = 0;
        while (*s != 0) {       // have we reached the zero?
                length += 1;    // one more char found.
                s += 1;         // step to the next character.
        }
        return length;
}
```

An alternative `size_t strlen(const char* s);`

Calculates the length of a null-terminated string

```
size_t strlen(const char* s)
{
        const char*   s0 = s;
        while (*s != 0)
                s += 1;
        return s - s0;
}
```

Save the initial value of pointer s

Iterates through the characters of the string pointed by s until it encounters null termiantor

Move to the next character in the string

Calc the length of the string, i.e. the number of characters between the initial and final pointers.

- Pointer difference is the number of elements between what the pointers point to
- Subtracting two pointers must be a signed integer type — not `size_t`
- The type is called `ptrdiff_t`
- With good compilers, these two versions result in the same machine code though.

A simpler size_t strlen(const char* s);

```
size_t strlen(const char* s)
{
        size_t  i;

        i = 0;
        while (s[i] != 0)
                i += 1;
        return i;
}
```

- This is simplest to read
- With good compilers, these three versions result in the same machine code though.
- Bottom line: keep it simple until you know it is "worth" trying to optimize it by hand

The C Programming Language

- Terminology for discussing the C Standard
- Lexical elements
- Declarations
- Expressions
- Statements
- Preprocessing directives
- The Standard C Library

The C Standard

- The C compiler and the Standard Library provided with the compiler is referred to as the *Implementation*.
- The Standard consists of requirements at different levels on a program:
- *Constraints* can be checked at compile-time. Eg forgotten declaration of a variable or a syntax error.
- If a Constraint is violated by a program, it *must* be diagnosed by the compiler.
- *Semantics*. The behavior of a language construct is normally described in a Semantics section of the Standard.

Implementation-defined behavior

- An implementation is free to make certain decisions about the behavior which it must follow consistently and document.
- This is called *Implementation-defined behavior*.
- Examples include
  - The size and precision of various types.
  - How bit-fields are layed out in memory.
  - Whether right shift of an signed integer is arithmetic or logical.
  - Whether the **register** keyword has any effect on performance.
- Portable programs should avoid using some of the language constructs with implementation-defined behavior.

Unspecified behavior

- *Unspecified behavior* lets the implementation decide on the behavior and it does not have to document the behavior since it can vary "randomly" eg due to optimization, and should be avoided if it can affect observable behavior.
- Examples include
  - The order of evaluation in + is unspecified.

```
int a = 12, b = 13;
int f(void) { printf("%d\n", a); return a; }
int g(void) { printf("%d\n", b); return b; }
int main() { f() + g(); return 0; }
```

  - The order of evaluation of arguments in function calls.
  - Whether two identical string literals share memory.
  - Whether **setjmp** is a macro or identifier with external linkage; **&setjmp** is bad.

Undefined behavior

- The worst situation is *undefined behavior*; (ugly form of bug).
- The implementation is permitted to do *anything* including
  - Terminating compilation with an error message.
  - Continuing without understanding what happened.
  - Continuing possibly with a warning message.
- Examples of undefined behavior include
  - A requirement which is not a Constraint is violated.
  - An invalid pointer is dereferenced.
  - A stack variable is used before it was given a value.
  - Divide by zero.
  - Array index out of range.

Lexical elements

smallest units in the source code that have a specific meaning to the compiler or interpreter

- Character sets
- Keywords
- Identifiers
- Universal character names
- Constants
- String literals
- Punctuators
- Header names
- Preprocessing numbers
- Comments

Character sets

- The *Basic character set* must be supported by all C compilers
  - Lower and upper case Latin alphabet
  - Decimal digits
    - ```
      ! " # % & ' ( ) * + , - . / :
      ; < = > ? [ \ ] ^ _ { | } ~
      ```
- *Extended character sets* may optionally be supported and can include Swedish, Japanese etc. Represented by multibyte characters.
- Trigraph sequences: be careful in strings: "trigraph? what??!"

```
        ??= #       ??) ]       ??! |       ??( [
??' ^
        ??> }       ??/ \       ??< {       ??- ~
```
```

Keywords

auto	extern	short	while
break	float	signed	_Alignas
case	for	sizeof	_Alignof
char	goto	static	_Atomic
const	if	struct	_Bool
continue	inline	switch	_Complex
default	int	typedef	_Generic
do	long	union	_Imaginary
double	register	unsigned	_Noreturn
else	restrict	void	_Static_assert
enum	return	volatile	_Thread_local

New in C99: `inline`, `restrict`, `_Bool`, `_Complex`, and `_Imaginary`
New in C11: `_Alignas`, `_Alignof`, `_Atomic`, `_Generic`, `_Noreturn`, `_Static_assert`, and `_Thread_local`

Identifiers

- An identifier starts with a nondigit and then may contain digits
- A nondigit is underscore, [A-Z], [a-z], a universal character name, or an implementation-defined multibyte character
- It is not portable to use Å, Ä, or Ö in identifiers (as in Java)
- Identifiers with a leading underscore are reserved for the system: don't use them

```
// in a header file: #define _num        1234567890
typedef struct _num {
        struct _num*    next;
        int             value;
} num;
```

Universal character names (UCNs)

- Used to specify any Unicode character
- Written as `\Unnnnnnnn` or `\unnnn` where n is a hex digit.
- Can be used in identifiers, strings, and character constants

Constants 1(4)

- Integer constants:
  - integer-suffix: combination of **u**, **U**, **l**, **L**, **ll**, **LL**
  - decimal-constant integer-suffix, eg **1ULL**
  - octal-constant integer-prefix, eg **0123**
  - hexadecimal-constant integer-prefix **0xabc123**
- Floating constants:
  - **float** constant, eg **123.456e12F**
  - **double** constant, eg **123.456e12**
  - **long double** constant, eg **123.456e12L**
  - C99: hexadecimal floating constant, eg **0xap-3** $= 10 \times 2^{-3} = 1.25$

Constants 2(4)

```
float   x;
int main()
{
        x += 0.1;
}
main:   lis 4,x@ha
        lis 5,.LC0@ha
        lfs 5,x@l(4)
        lfd 4,.LC0@l(5)
        fmr 3,5
        fadd 2,3,4
        frsp 1,2
        stfs 1,x@l(4)
        blr
```

```
float    x;
int main()
{
         x += 0.1F;
}
main:   lis 4,x@ha
        lis 5,.LC0@ha
        lfs 2,x@l(4)
        lfs 3,.LC0@l(5)
        fadds 1,2,3
        stfs 1,x@l(4)
        blr

// No conversion to double!
```

Constants 3(4)

- Character constants
  - Normal character constant:

    ```
    '1'   'A'
    ```

  - Simple escape character constant:

    ```
    '\''   '\"'   '\?'   '\\'   '\a'   '\b'
    '\f'   '\n'   '\r'   '\t'   '\v'
    ```

  - Octal character constant, one, two, or three digits:

    ```
    '\1'   '\12'   '\123'
    ```

  - Hexadecimal character constant, any number of digits:

    ```
    '\x1'   '\x12'   '\x123'   '\x1234'   etc
    ```

    But more than two will most likely cause an overflow (implementation-defined)
  - Universal character name:

    ```
    '\U12345678'   '\u00ab'
    ```

Constants 4(4)

Indicates long literal

- Wide character constants
  - Like normal character constant but with an **L** prefix:

```
#include <wchar.h>     /* or <stddef.h> or <stdlib.h> */

wchar_t w = L'A';
```

  - The size of the type **wchar_t** is usually two or four bytes

String literals 1(2)

- Adjacent string literals are automatically concatenated: **"hello, "** **"world"** becomes **"hello, world"**
- Strings are ended with a zero character: 0 or $'\backslash 0'$
- The string consisting of bytes 255, '8', and 0 *cannot* be written as:

  ```
  "\xff8"
  ```

  but the following works

  ```
  "\3778"      "\xff" "8"
  ```

String literals 2(2)

- A wide string is written as **L"hello, world"**
- In ANSI C from 1989 (and still in most C compilers today), mixing normal strings and wide string resulted in undefined behavior
- In C99 the resulting string literal becomes wide.