Contents of Lecture 11

- The C Library

The C Standard Library

- There are 24 header files in C99 and 29 in C11.
- We will go through some of the more important header files.

<assert.h>

- To check that your assumptions hold during execution, you can do as follows:

```
#include <assert.h>

void insert_first(list_t** list, void* data)
{
        assert(*list != NULL);

        /* ... */
}
```

- It is useful during development and can be used for consistency checks.
- Compiling with `cc -DNDEBUG` will disable the test. Therefore don't do:

```
assert((fp = fopen(name, "r")) != NULL);
```

- `assert` can be implemented as in Example 13.1.1

<assert.h>

- If NDEBUG is not defined, the expression is evaluated, and if it is nonzero, nothing happens.
- If the expression is false, an error message is printed and the abort function is called.
- Suppose you want to check that a pointer is 8 bytes:

```
assert(sizeof(void*) == 8);
```

- How can you check that during compilation?

Why is the following wrong?

```
#if sizeof(void*) != 8
#error the program assumes a pointer is 8 bytes.
#endif
```

Static assertion

- Since the preprocessor knows nothing about the `sizeof` operator we must do something else.
- C11 has a new construct for it called `_Static_assert`, but we can easily define a macro.
- The idea is:

```
int array[ sizeof(void*) == 8 ? 1 : -1 ];
```

- If the expression is false, we would declare an array with -1 elements which the compiler must complain about.
- To avoid:
  - actually declaring an array and waste memory,
  - having to invent a different array name every time

... we can do as in Example 9.11.1, which instead of an array variable declares an array typedef (wastes no memory) and uses token concatenation (##) to make the line number part of the name.

<ctype.h>

- `<ctype.h>` contains classification functions such as `isdigit`.
- They take an `int` parameter and return a nonzero value to indicate truth.
- It is wrong to write:

```
if (isdigit(c) == 1)
        /* ... */
```

Since the return value equally well could be 2 if c is a digit.

<fenv.h>

- Defines macros for exceptions:

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
```

- Defines macros for rounding modes:

```
FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD
```

Exceptions

- The exceptions can be set both by hardware and software.
- When a math function detects an invalid input argument it should set the `FE_INVALID` bit in the processor's floating point status register.
- There are functions for fetching a copy of the floating point status register and for testing and clearing bits, and other operations — see below.

<errno.h>

- C has traditionally stored error codes in a variable called `errno`.
- There are three standard errors:

```
EDOM
ERANGE
EILSEQ
```

- The first two refer to math errors: an argument was not in the domain of the function and the return value could not be represented in the range of the return type.
- The `EILSEQ` is used with an invalid multibyte character sequence.
- Operating systems define others such as
  - `ENOENT` for "No such file or directory", and
  - `EPERM` for "Permission denied".

Using errno

- We should set errno to zero before any call which might fail such as opening or removing a file and some math functions.
- For example:

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

int main(void)
{
        errno = 0;
        if (remove("/") == -1)
                perror("cannot remove \"/\"");
}
```

errno

- `errno` behaves as if it was declared as a global variable `int errno;`
- For multi-threaded programs this doesn't work very well — due to data-races.
- Each thread gets its own copy of `errno` and this typically is implemented as:

```
int* __get_errno_for_current_thread(void)
{
        return &current_thread->errno;
}

#define errno (*__get_errno_for_current_thread())
```

- Then we can use it as:

```
errno = 0;
/* ... */
```

errno and C11

- With C11 we can instead declare `errno` using:

```
_Thread_local int errno;
```

- This way each thread gets its own copy of `errno`.

Reporting errors from libraries

- `errno` is intended for use by system libraries such as the API's for performing system calls and Pthread libraries.
- System calls are special function calls provided by the operating system which means Windows has one set of system calls and UNIX, including MacOS X, Linux and AIX, have other sets.
- To report errors from your own libraries, it is often a good idea to define an enum with the different error codes.

<math.h>

- Compile with `-lm` at the end of the command: `gcc a.c -lm`
- Traditionally `errno` is used but C99 allows math exceptions to be tested in a different way.
- We need to check which way the library reports math errors using `math_errhandling`:

```
errno = 0;
sqrt(-1);

if (math_errhandling & MATH_ERRNO)
        /* ... */

if (math_errhandling & MATH_ERREXCEPT)
        /* ... */
```

Math errors reported with errno

```
if (math_errhandling & MATH_ERRNO) {
        if (errno == EDOM)
                puts("EDOM");
}
```

Math errors reported as exceptions

```
if (math_errhandling & MATH_ERREXCEPT) {
        except = fetestexcept(FE_ALL_EXCEPT);
        if (except & FE_INVALID)
                puts("FE_INVALID");
}
```

<inttypes.h>

- Using `<stdint.h>` we can declare exact width integers such as `int32_t`.
- How should we print them?

```
int32_t         a;

printf("a = %d\n", a);  // not portable
printf("a = %ld\n", a); // not portable
```

- What should we do?

`<inttypes.h>`

- This header file declares macros which are strings that can be used.
- For example:

```
#include <inttypes.h>

int32_t         a;

printf("a = %" PRId32 " \n", a);
```

- `<inttypes.h>` includes `<stdint.h>`.
- Stricly speaking this is also not portable since it is implementation defined whether there is an `int32_t` but if there is, this is how to print it.
- For instance a DSP-processor with 24-bit `int` may not have `int32_t`.

<setjmp.h>

- To jump to a label `L` we use `goto L;`
- In C we can also jump from one function to another.
- Consider:

```
void g(void) { /* ... */ }
void f(void) { g(); }
int main(void)
{
        /* ... */
        f();
}
```

- Usually `g` returns to `f` which returns to `main`.
- If we wish we can return from `g` directly to `main`.
- Instead of `return` we use `longjmp`.
- `longjmp` has an ever worse reputation than `goto` but can be useful.

What is the context of an executing thread?

- Program counter or PC
- Registers
- To make a jump to a function f, that function must already have an allocated stack frame and its program counter and registers must have been saved.
- Thus e.g. `main` cannot jump into the middle of any function — a call to the jumped-to function must already be active such as the call to `f` above.
- There is a type `jmp_buf` in which the PC and registers are saved.
- A jump is performed by loading all registers and finally the PC from such a `jmp_buf` variable.

Non-local jumps with `setjmp` and `longjmp`

- To make a non-local jump, two operations are needed:
- Initialize the `jmp_buf` variable — using `setjmp`.
- Calling the function `longjmp` with the `jmp_buf` variable as one of the parameters.
- The call to `longjmp` will result in another return from `setjmp`!
- To distinguish the initialization call of `setjmp` and the returning jump, `setjmp` returns zero when called to initialize a `jmp_buf` variable and the second paremeter to `longjmp` otherwise.

Typical usage

```c
#include <setjmp.h>

jmp_buf buf;

int main(void)
{
        switch (setjmp(buf)) {
        case 0: /* initialization. */ break;
        case 1: /* from longjmp.    */ break;
        }
}

void g(void)
{
        if (must_stop())
                longjmp(buf, 1);
}
```

Remarks

- Almost always non-local jumps are not needed.
- In a chess program which has found a winning move it can be appropriate to terminate a deep recursive search using `longjmp`.
- Functions with non-local jumps are **very** annoying to optimizing compilers and often result in slower code.

<signal.h>

- A signal is a way of notifying a running program that something has happened.

Signal	**Example cause**	**Default effect**
SIGABRT	`abort();`	Terminate the process
SIGFPE	Implementation defined	Terminate the process
SIGILL	Illegal instruction	Terminate the process
SIGINT	Ctrl-C	Terminate the process
SIGSEGV	Invalid address	Terminate the process
SIGTERM	kill <pid>	Terminate the process

Some UNIX-specific signals

Signal	**Cause**	**Default effect**
SIGSTOP	Ctrl-Z	Stop the process
SIGSTOP	kill -SIGSTOP <pid>	Stop the process
SIGCONT	kill -SIGCONT <pid>	Resume the process
SIGBUS	eg non-alignad memory access	Terminate the process
SIGKILL	kill -SIGKILL <pid>	Terminate the process
SIGKILL	kill -9 <pid>	Terminate the process

Common use

- To get informed about a signal, sent from the operating system, we must register a so called signal handler.
- A signal handler is simply a function that the operating system runs for us.
- If we have not registered a signal handler before a signal is received our program usually is terminated, i.e. that is the default action.
- To register a signal handler `catch_ctrl_c` for `SIGINT` we can do:

```
#include <signal.h>

void catch_ctrl_c(int s) { /* ... */ }

int main(void)
{
        signal(SIGINT, catch_ctrl_c); for (;;);
}
```

The `signal` function

- The signal function tells the operating system which function to call instead of terminating our program.
- The function `signal` returns the previously registered function for a particular signal number.
- The declaration of the `signal` is perhaps confusing to read:

```
void (*signal(int signum, void (*func)(int)))(int);
```

- The two parameters to `signal` are `signum` and `func`.
- The * before `signal` is there due to the return value is a pointer (to a function).
- Since the same function can be signal handler for different signals, the `int` parameter of the signal handler specifies which signal occurred.

Delivering a signal

- When an event happens which triggers a signal, the operating system blocks additional instances of the same signal to avoid having the signal handler being invoked multiple times for the same signal.
- This blocking is removed when the signal handler returns to the operating system.
- After that, the operating system will let the program resume execution.
- What happens if the signal handler instead of returning makes a `longjmp`?
- The signal will remain blocked since the operating system still thinks the signal handler has not returned.

`<stdlib.h>`

- To convert a number in string to an integer, the function `strtol` is useful.
- It takes three paramets:
  - A pointer to a string: `char* s`
  - An optional pointer to a pointer to a string: `char** end`
  - The base, 2-36 — or zero and then the base is inferred from the string.
- The function sets `*end` to point to the first character after the number — unless `end` is a null pointer.
- For example:

```
int     a;
char*   end;

a = strtol("119", &end, 2);
```

a is set to 3 and `end` to point to the 9.

<string.h>

- To split a string into parts, called tokens, the function `strtok` can be used.
- It is used in two phases:
  - First two parameters are provided:

```
char*   s;
char    a[] = "a string. hi: there";
char*   sep = " :.";

s = strtok(a, sep);
```

  - The first parameter **must be modifiable**.
  - The second parameter contains a set of characters which are used to separate tokens.
- If the first parameter is null, search continues in the previously used string.

strtok example

- For example:

```
char*   s;
char    a[] = "a string. hi: there";
char*   sep = " :.";

s = strtok(a, sep);
while (s != NULL) {
        printf("%s ", s);
        s = strtok(NULL, sep);
}
```

- The output will be: `a string hi there`
- The returned string assigned to `s` is null-terminated!
- That means `strtok` modifies the first non-null parameter which therefore must be modifiable.
- Using `char* s= "hello there";` may result in a read-only string!

Sorting array of int using qsort

- an array (i.e. a pointer to the first element)
- number of elements
- size of each element
- a comparison function

```
int compare(const void* ap, const void* bp)
{
        const int*      a = ap;
        const int*      b = bp;

        //return *a - *b;
        if (*a > *b)
                return 1;
        else if (*a < *b)
                return -1;
        else
                return 0;
}
```

Buffer overflows

- A buffer overflow means array index out-of-bounds errors.
- Checking that an array index is within the array bounds is not done in C, as in Java.
- The checking is only useful for programs with bugs.
- To avoid such errors, the following simple rule is sufficient:
  Don't trust untrusted data.
- In other words, make a sanity check for all input, and use range checking library functions.
- When there is a risk for overflow: check it explicitly.
- For C: make the calculation (how depends on the type).
- Java does also not report errors on overflow (and cannot check it for floating point values).

An example: `sprintf` and `snprintf`

- Both functions behave as `printf` but put their output in a buffer pointed to by the first parameter.
- The output is null terminated.
- `sprintf` assumes the buffer is sufficiently large.
- The second parameter of `snprint` specifies the buffer size.

Never use `gets`

- The function `gets` reads the next line of input from `stdin` and copies it to a buffer supplied to `gets`.
- No length check is done. Don't use `gets`. It may disappear from C.
- Use `fgets` instead which takes a buffer, a size, and a `FILE` pointer as parameters.

Another example: `strcpy` and `strncpy`

- `strcpy` copies the string pointed to by the second parameter into memory pointed to by the first parameter upto and including the terminating null byte.
- `strncpy` does the same but copies at most `n` bytes.
- Warning: `strncpy` may skip the null byte!
- Similar situation for `strcat` which appends a string.
- Use `strncat` instead.

Compiler protection against stack buffer overflow

- Buffer overflows for stack allocated arrays can often be detected by letting the compiler put ”canaries” with special values after such arrays.
- The term comes from mines when canary birds where used to detect toxic gases.
- If the value of the canary has changed, it is likely due to a buffer overflow (or some other write through an invalid pointer).
- This was supported by GCC some years ago.
- LLVM/clang has support for a tool called AddressSanitizer which behaves like Valgrind but is better integrated with GDB.

C vs C++

- I was requested to answer the question of why we should program in C when there is a language called C++.
- C compilers are reliable. The complexity of C++ make me think that not even a C++ front-end will ever be bug-free.
- C is nicer than Fortran — the other high-performance language.
- It is possible to make C code inefficient using a bad algorithm or for instance by not calling functions directly but always through a pointer to a struct which contains pointers to functions. This confuses optimizing compilers.
- Virtual functions in C++ behave like that so the main reason for using C++ over C makes your program slower.
- If you recompile a 10-year old C program, normally it just works.
- If you do that with C++ usually it does not compile.