Contents of Lecture 4: Declarations

- Implicint int
- Storage class specifiers
- Type specifiers
- Enumeration specifiers
- Type qualifiers

Now obsolete: implicit `int`

- Sometimes you can see code such as:

```
main()              // invalid
{

}
```

- or even:

```
#include <stdio.h>

count;              // invalid

float   x;
```

- In earlier versions of C one could skip the type, which then became `int`, and is called **implicit int**.
- Calling a function before its declaration also set its return type to `int`.
- It's invalid C so don't use it — but compilers often allow it...

Storage class specifiers

- Last lecture we discussed the different kinds of storage durations.
- Now we will see how to specify some of them explicitly.
- Dynamic (important) and temporary (less important) storage duration are not specified by the programmer using any particular syntax but defined by the standard.
- The storage class specifiers are:

```
typedef          extern          static
_Thread_local    auto            register
```

- Of these `typedef` does not refer to any kind of storage duration — instead it introduces another name of a type and not a new type:

```
typedef int      num_t;
int*             p;
num_t*           q;
p = q; // valid since p and q have the same type.
```

Storage class specifiers: static at file scope

```
static int count;     /* initialized to zero. */

static void init(void)
{
        /* Do some initializations... */
}
```

- Used to make an identifier invisible outside the source file
- With static at file scope, there is no risk of name conflicts with other files.
- Always use static for file-local identifiers.

Storage class specifiers: static at block scope 1(2)

```
int fun(int a, int b)
{
        static int beenhere;    /* initialized to zero. */
        if (!beenhere) {
                init();
                beenhere = 1;
        }
        /* do the normal work... */
}
```

- Used to make an identifier invisible outside the block (function in this case)
- Static storage duration: variable is not located on the stack but among global variables; preserves its value across function calls

Storage class specifiers: static at block scope 2(2)

```
int fun(int a, int b)
{
        static int c = 12;  // OK.
        static int* d = &c; // OK.
        static int* e = &a; // Invalid: non-constant.
}
```

- A static variable can be initialized with a constant expression
- An address may or may not be constant: `&c` is a constant expression but `&a` is not — why?
- Answer: the address of a stack allocated variable is the sum of the stack pointer and a constant — and the stack pointer is certainly not a constant.

Storage class specifiers: extern

```
static int a;                // internal linkage.
extern int a;                // OK — still internal linkage.
extern int b;                // external linkage.
static int b;                // No: static follows extern.
int f(void);                 // Implicitly external linkage.
int main(void) { f(); }      // OK.
static int f(void) { }       // No: undefined behavior!
```

- The `extern` does not change a previously declared visible storage class.
- `static` followed by `extern` is OK but `extern` followed by static is not.
- These rules have to do with how one-pass compilers can be implemented, assembler code may already have been generated which cannot be changed.

Storage class specifiers: auto and register

```
int fun(void)
{
        register int c;
        int*         d = &c; // invalid.
        register int e[4]    // OK.
        register struct { int a; int b; } f; // OK.
}
```

- auto is completely useless
- register indicates to the compiler that the variable should be kept in a register if possible. usually ignored, except for semantic analysis of the following:
- The address of a variable with register storage class cannot be taken.

Storage class specifiers: typedef

```
typedef struct list_t list_t;
struct list_t {
        list_t*         next;
        void*           data;
};
```

- `typedef` creates a synonym for a type.

Type specifiers: basic types 1(2)

- The type specifiers are: `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, `unsigned`, `_Bool`, `_Complex`, `_Imaginary`, *struct-or-union*, *enum-specifier*, and *typedef-name*.
- The type specifiers are combined into lists including `signed char`, `unsigned char`, `char`, `signed long long` and `long double`.
- Note that `signed char`, `unsigned char`, and `char` all are different types: `char` behaves like one of the other two (which is implementation-defined) but it is a distinct type.

```
char*               s;
unsigned char*      t = s; // invalid.
```

Type specifiers: basic types 2(2)

```
_Bool a;
#include <stdbool.h>
bool  b;
```

- The type `_Bool` is new in C99.
- `<stdbool.h>` defines `bool` as a macro which expands to `_Bool`.
- A bool can only take the values zero and one.
- An assignment to a bool variable stores a one if the expression is not zero.

Type specifiers: enum

```
#include <limits.h>

enum color_t { RED, BLUE, GREEN };
enum large_t { a, b, c, d = INT_MAX - 1, e };
enum too_large_t { f = INT_MAX, g }; // invalid
enum color_t    color;
```

- Unless the value is set explicitly, it becomes one more than the previous, and zero for the first.
- An enum declares named `int` constants.
- The constants must be representable as `int` but the compiler may decide to use a smaller type.
- Many compilers allow larger constants.
- Enums are sometimes better than #defines because debuggers understand them.
- The variable `color` can be used where an int can be used, eg as an array index.

Type specifiers: structs and unions

```
struct s {
        int          a;                    // OK.
        int          b:1;                  // OK, but avoid.
        signed int   c:1;                  // OK.
        unsigned int d:1;                  // OK.
        _Bool        e:1;                  // OK.
        color_t      f:2;                  // Impl. def.
        int          g(int, int);          // No.
        int          (*h)(int, int);       // OK.
        int          i[0];                 // No.
        int          j[];                  // See below.
};
```

- Avoid using plain `int` as bit-field type. Specify whether it is signed or not.
- The `j[]` is called a flexible array member and will be explain soon.

Type qualifiers

```
const int a = 12;   // OK.
const int* b = &a;  // OK.
*b = 13;            // invalid.
a = 14;             // invalid.
```

- `const` the variable cannot be changed after initialization.
- `volatile` the variable can be changed in "mysterious" ways: do not put it in a register.
- `restrict` a pointer with restrict qualifier points to data which no other visible pointer can refer to if any of them modifies the data, and helps compilers to optimize code but can cause extremely obscure bugs if the programmer is not careful.

Declarators: Type constructors

- There are three type constructors:
  - Array
  - Function
  - Pointer
- Array and Function have higher precedence than Pointer
- Place array dimension or the function's parenthesis to the right of the declarator and a star before the declarator
- Confusion arises because the type cannot be read from left to right but must be read from "inside" to the "outside": `int (*a[12])(int);`. What is the type of a?

Declarators: Examples

```
int     a;             // int
int     *b;            // pointer to int
int     **c;           // pointer to pointer to int
int     d[4];          // array of int
int     e[4][5];       // array of array of int
int     *f[4];         // array of pointers
int     (*g[4])[5];    // array of pointers to array of int
int     *h();          // function returning pointer to int
int     (*i)();        // pointer to function returning int
int     *j()();        // NO: func returning func
int     (*k())();      // func returning pointer to func
```

- A function cannot return a function or an array, only pointers to them.

Initialization

```
typedef struct { int a, b, c, d; } type_t;

int main(void)
{
        int        a[10] = { 1, 2 };
        int        b[] = { 1, 2, 3 };
        int        c[] = { [4] = 12 };
        type_t     d = { .a = 3, .c = 5,6 };
        int        e;            // undefined value.
        static int f;            // zero.

        return 0;
}
```

A struct with an array

- Consider the struct below:

```
#define N       (10)
struct s {
        int     a[N];
};
```

- Now suppose you want to decide on N during execution!
- How can you do that?

A struct with a pointer

```
#include <stdlib.h>

struct s {
        size_t          n;
        int*            a;
};

/* ... */

struct s*       s;

s = malloc(sizeof(struct s));
s->n = n;
s->a = malloc(n * sizeof(int));
```

- This wastes memory for the pointer and time for two `malloc` calls.
- How can we fool the compiler?

The struct hack (invalid code)

- This is an official term in the ISO standard.

```
#include <stdlib.h>

struct s {
        size_t          n;
        int             a[1];
};

/* ... */

struct s*       s;

s = malloc(sizeof(struct s) + (n-1) * sizeof(int));
s->n = n;
```

- Unfortunately, `s->a[1]` is invalid C — and the C compiler or a runtime testing tool can strike back!

Flexible array member

- New in C99.

```
#include <stdlib.h>

struct s {
        size_t          n;
        int             a[];
};

/* ... */

struct s*       s;

s = malloc(sizeof(struct s) + n * sizeof(int));
s->n = n;
```

- Now `s->a[0]` up to and including `s->a[s->n-1]` is valid C.

Restrictions

- A flexible array member may only be used in a struct.
- It must be the last struct member.
- It must not be the only struct member.
- We cannot declare an array of a struct with a flexible array member.
- We can only declare a pointer to such a struct and not a variable directly since the compiler is not required to be able to figure out how much memory should be allocated.

Variable length array: VLA

- A local array (with automatic storage duration, i.e. allocated on the stack) can have a non-constant size:

```
void f(size_t n)
{
        int     a[n];

        /* ... */
}
```

- We the execution comes to the declaration it evaluates and remembers the size of `n` and the compiler must allocate memory for the array.
- This memory is allocated on the stack simply by changing the stack pointer.
- Thus the programmer does not have to (and cannot) deallocate that memory.
- The memory is automatically deallocated when the function returns.

Local array only

- Since the memory for a VLA is allocated from the stack, only local arrays can have a non-constant size.
- VLA's are different from flexible array members.
- A VLA must be an ordinary identifier and not for instance a struct member.
- VLA's are new in C99 and very useful.

Restrictions of VLA's

- Unfortunately:
  - VLA's has become optional in C11, but since `gcc` supports them most compilers will also.
  - It's impossible to know whether the allocation succeeded or not.
- Use with care and **never** for a size supplied as program input — otherwise a certain security risk.
- Commercial compilers use it (or a similar approach called `alloca`) to improve speed.
- For instance allocating an array of object pointers can make iteration through a data structure simpler and faster — forget it in a nuclear power plant though.

VLA's and normal arrays

- A VLA is organized in memory the same way as other arrays are.
- The only difference really is that the compiler must produce code to remember the array sizes.

```
void f(int a[3][4]);

void g(size_t m, size_t n)
{
        int     b[m][n];

        f(b); // OK if m == 3 and n == 4.
}
```

VLA parameters

- A VLA can be a parameter:

```
void f(size_t m, size_t n, int a[m][n])
{
        /* ... */
}

int b[3][4];

void g(void)
{
        f(3, 4, b); // OK.
}
```

- This is not dangerous in any way since the matrix was not allocated on the stack.

A function prototype with VLA

- Consider f again:

```
void f(size_t m, size_t n, int a[m][n])
{
        /* ... */
}
```

- How can we declare f in a header file?
- We can use:

```
void f(size_t m, size_t n, int a[m][n]);
```

- But if we don't want to write m and n?
- Recall: we can write prototypes without parameter names:

```
void* malloc(size_t);
```

- The we do as follows:

```
void f(size_t, size_t, int [*][*]);
```

- Can we remove one or both of the stars?

Recall array parameters become pointer parameters

- An array parameter becomes a pointer parameter and we can therefore skip the size:

```
void f(size_t, size_t, int [][*]);
```

- Or:

```
void f(size_t, size_t, int (*)[*]);
```

- But that does not win a prize for beautiful C code.

Variably modified types

- A type with a VLA is called a variably modified type.
- We can declare a pointer to a VLA:

```
void f(size_t n)
{
        int     (*p)[n];

        /* ... */
}
```

Example 1.4.13

- With variably modified types, we can do:

```
#include <stdlib.h>

void f(size_t m, size_t n)
{
        double  (*a)[n];

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

        /* ... */

        a[i][j] = ....
}
```

- Only one memory allocation and we can use matrix notation.
- This works because the compiler must understand that the number of columns is given by `n`.

GDB — The GNU Debugger 1(4)

- Compile with -g switch to cc.
- GDB can take control over a running process: `$ gdb program pid`

Syntax	Example	Description
`r` *arguments...*	`r 20`	Starts the program with an argument lis
`b` *func*	`b main`	Sets a breakpoint in a function.
`b` *line*	`b 12`	Sets a breakpoint at a line in the current
`d` *number*	`d 1`	Deletes a breakpoint.
`c`		Continues execution.
`p` *var*	`p i`	Prints the value of a variable.
`display` *var*	`display i`	Prints value every time the program stop
`undisplay` *number*	`undisplay 2`	Cancels a display.

GDB — The GNU Debugger 2(4)

Syntax	Example	Description
`n`		Steps to the next line (skips called functi
`s`		Steps to the next line (steps into called fu
`p /x` *var*	`/x i`	Prints on hexadecimal format.
`p /u` *var*	`/u i`	Prints on unsigned format.
`p /d` *var*	`/d i`	Prints on signed format.
`p /c` *var*	`/c i`	Prints data as characters.
`p /f` *var*	`/f i`	Prints data as floating-point numbers.
`x` *format address*	`x /c &result`	Prints memory area. Hit return for next a
		Same formats as when printing (see abov
`display /i $pc`		Disassembles the next instruction to be e
`display $r4`		Displays register r4.

GDB — The GNU Debugger 3(4)

Syntax	Example	Description
`where`		Prints trace of all function calls.
`kill`		Terminates the debugger process.
`watch` *var*	`watch a[i]`	Sets a watchpoint (see below).
`whatis` *var*	`whatis a`	Print type of a variable or function.
`up`		Goes up to caller.
`down`		Goes down to callee.
`source` *file*	`source cmd`	Reads and executes commands.
`make`		Runs the UNIX make command.

GDB — The GNU Debugger 4(4)

- The print command can call a function in the application and print the return value: `p strlen(’’gdb is cool’’)`
- Watchpoint: watches for modifications of a variable. This can be *very* slow since usually entire virtual pages must be set read-only. Can be the best solution to kill ugly bugs when ”random” pointers are writing in memory where they should not. The syntax is: `watch variable`.
- GDB has both command line and graphical interfaces (the `ddd` and `xgdb` programs).

Make 1(4)

```
CC          = cc
CFLAGS      = -O5
CFLAGS      = -pedantic -Wall -g -Werror
OBJS        = arena.o list.o main.o

all: $(OBJS)
        $(CC) $(LDFLAGS) -o prog $(OBJS)

test: all
        prog > output
        diff output correct

dep:
        $(CC) -M *.c > depend

include depend
```

Make 2(4)

- Variables are set eg as `CFLAGS = -O5`. Only the last assignment counts.
- `OBJS = arena.o list.o main.o` tells make which object files should be created. To create eg arena.o, make looks for a file arena.c in the current directory and compiles it.
- `all: $(OBJS)`. Rules are written as `target: x y z...` and means that to produce target (the name before the colon) the names after the colon must first be produced. In our case, to produce `all`, each object file must be produced which makes does by compiling them, if necessary (see below).

Make 3(4)

- When all items on the right of the colon are ready (up-to-date) the commands on the lines below the rule are executed. Whenever any of these commands returns a nonzero value, make will stop and report an error (unless the command was preceded by a minus).
- A C file is recompiled if it has changed, or if any header file included from the C file has changed. The rule `dep` uses the C compiler switch `-M` to create a file with such dependencies.
- The rule `test: all` first recompiles the program if necessary, then runs it and puts the output in a file called `output` which is then compared (using `diff`) with the expected output in the file `correct`.

Make 4(4)

- By default, make will perform the first rule. To do the test, say `make test`, or move that rule first and run make with no arguments.
- `Note` that the commands below a rule must be preceded by a tab character.