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.

Declarations

- Storage class specifiers
- Type specifiers
- Type qualifiers
- Function specifiers
- Declarators
- Type names
- Type definitions
- Initialization

Storage class specifiers: static at file scope

```
static int count;     /* implicitly 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 initialized; /* zero. */
        if (!initialized) {
                init();
                initialized = 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.
}
```

- 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.

Storage class specifiers: extern 1(2)

```
extern int a;                   // does not reserve storage for a.

int main()
{
        sizeof a;
        return 0;
}
```

- If the value of an extern identifier is used, storage must have been reserved for it somewhere in some file.
- In the program above, the value of a is not used so no definition is needed

Storage class specifiers: extern 2(2)

```
static int a;           // reserves storage for a.
extern int a;           // a still invisible outside the file.
extern int b;           // b is global visible outside the file
static int b;           // Undefined behavior. static follows extern.
int fun();              // Implicitly external linkage.
int main() { fun(); }   // Use fun assuming it has external linkage.
static int fun() { }    // 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()
{
        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 int int32_t;
typedef struct info_t info_t;
struct info_t {
        info_t*         next;
        int             data;
};
```

- **typedef** creates a synonym for a type.
- **typedef** is not really a storage class specifier. called so for syntactic convenience only.

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

```
enum colour { RED, BLUE, GREEN };
enum a { a, b = 100, c };
typedef enum { PORSCHE, MERCEDES, KOENIGSEGG } car_t;
car_t  car = PORSCHE;
```

- An enum declares named int constants
- Enums are "better" than #defines because debuggers understand them
- The *tags* **colour** and **a** are in a name space different from variables and enumeration constants.
- The variable **car** 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 signedness impl. def.
        signed int    c:1;              // OK, one signed bit.
        unsigned int  d:1;              // OK, one unsigned bit.
        _Bool         e:1;              // OK.
        car_t         f:2;              // OK if implementation permits.
        int           g(int, int);      // No, not in C.
        int           (*h)(int, int);   // OK. Pointer to function.
        int           i[0];             // No (but valid in GCC).
        int           j[];              // OK if last member in C99
};
```

- Avoid using plain int as bitfield type. Specify whether it is signed or not.

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 parameter with restrict qualifier points to data which no other visible pointer can refer to. helps optimizer 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 returning pointer to int
int     (*k())();   // func returning pointer to func returning int
```

- 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()
{
        int         a[10] = { 1, 2 };    // rest will be set to zero.
        int         b[] = { 1, 2, 3 };   // sizeof b == 3 * sizeof(int)
        int         c[] = { [4] = 12 }; // c[0..3] == 0
        type_t      d = { .a = 3, .c = 5,6 }; // d.d == 6.
        int         e;                   // undefined value.
        static int f;                    // zero.
        typedef int array[];             // incomplete type array
        array       g = { 1, 2, };       // does not affect the type array.
        array       h = { 1, 2, 3 };     // OK: array still incomplete.
}
```