

# Declarations

Auto: variable automatically created and destroyed (useless)  
Register: store the variable in the register for faster access  
Static: Exists throughout the program's execution  
Extern: defined elsewhere (another source file)

- Storage class specifiers

- Type specifiers

Int, char, float, double, void

- Type qualifiers

Const (variable value can't be modified), volatile  
(variable value can be changed by external factors)

- Function specifiers

Inline: (compiler should generate inline code for a function)  
\_Noreturn: specifies that a function does not return (due to exit or infinite loop=)

- Declarators

- Type names

Ex. Int \*ptr, where \*ptr is the declarator

- Type definitions

typedef: used to create an alias for an existing data type: ex. Typedef int myInt, an alias myInt for Int type


- Initialization

Assigning an initial value  
To a variable at the time of  
its declaration

Represent data types: int, char, or  
user defined structures

## 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)

A static variable must be initialised with a constant expression

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

Can't initialise a static variable from an address, since it may not be 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.

# 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 } Extern > static
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 (and does not mean the same as in C++)
- It is still in the C standard since changing the standard should not break existing C code
- **register** indicates to the compiler that the variable should be kept in a register if possible. usually ignored, except for semantic analysis: 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.
```

In C, there are strict type rules, and you cannot directly assign a pointer of one type to a pointer of a different type without using an explicit type cast.



## Type specifiers: basic types 2(2)

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

- The type **\_Bool** was introduced 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  
};
```

b:1 indicate that only 1 bit should be used to store the value of. A.k.a. bit-field width specifier

Specific signed or unsigned, avoid using only int as bitfield type

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

In struct, each member has its own separate memory space and the total size of the struct is the sum of the sizes of the members. They are stored sequentially in the memory and can be of different data types

# union

A union in C is a user-defined data type that allows different data types to be stored in the same memory location. Unlike structures, where each member has its own separate memory space, in a union, all members share the same memory space. This means that a union variable can hold values of different types, but only one at a time.

```
union u {  
    char    a[9];  
    double  b;  
};
```

- Alignment: for example, a 4-byte int wants to have an address that is a multiple of 4
- What is the size of this union?

# Flexible Array Member 1(3)

Flexibly assigning an array when the size is known

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

Pointer to an integer (int\*), which indicates it can be used to point to an array of integers

```
struct s* s;
```

Declares a pointer to a structure of type struct s

Custom allocation function

```
s = xmalloc(sizeof(struct s));
```

Allocates memory for an instance of the structure using xmalloc

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

Allocates memory for an array of n integers and assigns the address of the allocated memory to the a member of the structure. Each integer in the array will occupy sizeof(int) bytes

# Flexible Array Member 2(3)

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

← Array of ints, the size of this array is 1, but is intended to be used as a flexible array member

```
struct s*  
size_t      size;
```

```
size = sizeof(struct s) + (n-1) * sizeof(int);  
s = xmalloc(size);
```

← Calculates the total size needed for the structure, including flexible array member a

```
s->a[n-1] = 119; // Array index out of bounds ⇒ UB  
              // UB = undefined behavior
```

← Assigned value 119 to the last element of flexible array a.

↑ This works since the memory for a was dynamically allocated

- Known in the C standard as the "struct hack".
- Everybody "knows" it works but tools may complain.

# Flexible Array Member 3(3)

```
struct s {  
    size_t      n;  
    int         a[];    // Flexible array member  
};
```

```
struct s*      s;  
size_t         size;
```

```
size = sizeof(struct s) + n * sizeof(int);  
s = xmalloc(size);  
s->a[n-1] = 119; // OK ←
```

- Avoids storage for the pointer and is valid since C99
- Flexible array member cannot be only attribute
- Flexible array member must be last!
- Therefore at most only one flexible array member! ←
- Only for heap allocated structs!

# 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.



# Volatile qualifier

```
#include <signal.h>
volatile int x;
void catch_ctrl_c(int sig) // called when you hit CTRL-C.
{
    x = 0;
}

int main()    // below line will tell your computer to call the
{            // function catch_ctrl_c when you hit CTRL-C.
    signal(SIGINT, catch_ctrl_c);
    x = 1;
    while (x) // when compiling with optimisation and without
               ; // volatile, GCC thinks X cannot change!
}
```

# Const and restrict qualifiers

- The **const qualifier** informs the compiler it can put a variable in **read-only memory**. **Only an initialisation is permitted.**
- The **restrict** qualifier informs the compiler that **two parameters can not point to the same memory area**. This is new in C99 and its purpose is to tell the compiler some advanced tricks are legal.

*Restrict indicates that a and b don't overlap in memory*

```
void f(restrict int* a, restrict int* b, int n)
{
    int    i;
    for (i = 1; i < n-1; i++)
        a[i] = 2 * b[i-1] + 3 * b[i] + 4 * b[i+1];
}
```

# 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?

 a is an array of 12 pointers

# 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.

No copy => only pointers are used as input or return value

# Initialization

```
int      n = 10;

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.
    int      i[n];               // undefined values
    int      j[n] = { 1, 2, 3 }; // no
}
```