Chapter 18

Declarations

Declaration Syntax

- Declarations furnish information to the compiler about the meaning of identifiers.
- Examples:

```
int i;
float f(float);
```

- General form of a declaration:

  declaration-specifiers declarators `;`

- ***Declaration specifiers*** describe the properties of the variables or functions being declared.
- ***Declarators*** give their names and may provide additional information about their properties.

Declaration Syntax

- Declaration specifiers fall into three categories:
  - Storage classes
  - Type qualifiers
  - Type specifiers
- C99 has a fourth category, ***function specifiers,*** which are used only in function declarations.
  - This category has one member, the keyword `inline`.
- Type qualifiers and type specifiers should follow the storage class, but there are no other restrictions on their order.

Declaration Syntax

- There are four ***storage classes:*** `auto`, `static`, `extern`, and `register`.
- At most one storage class may appear in a declaration; if present, it should come first.
- In C89, there are only two ***type qualifiers:*** `const` and `volatile`.
- C99 has a third type qualifier, `restrict`.
- A declaration may contain zero or more type qualifiers.

Declaration Syntax

- The keywords `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, and `unsigned` are all ***type specifiers.***
- The order in which they are combined doesn't matter.
  - `int unsigned long` is the same as `long unsigned int`.
- Type specifiers also include specifications of structures, unions, and enumerations.
  - Examples: `struct point { int x, y; }`, `struct { int x, y; }`, `struct point`.
- `typedef` names are also type specifiers.

Declaration Syntax

- Declarators include:
  - Identifiers (names of simple variables)
  - Identifiers followed by `[]` (array names)
  - Identifiers preceded by `*` (pointer names)
  - Identifiers followed by `()` (function names)
- Declarators are separated by commas.
- A declarator that represents a variable may be followed by an initializer.

Declaration Syntax

- A declaration with a storage class and three declarators:

storage class — declarators

```
static float x, y, *p;
```

type specifier

- A declaration with a type qualifier and initializer but no storage class:

type qualifier — declarator

```
const char month[] = "January";
```

type specifier — initializer

Declaration Syntax

- A declaration with a storage class, a type qualifier, and three type specifiers:

storage class — type specifiers

```
extern const unsigned long int a[10];
```

type qualifier — declarator

- Function declarations may have a storage class, type qualifiers, and type specifiers:

storage class — declarator

```
extern int square(int);
```

type specifier

Storage Classes

- Storage classes can be specified for variables and—to a lesser extent—functions and parameters.
- Recall that the term *block* refers to the body of a function (the part in braces) or a compound statement, possibly containing declarations.

Properties of Variables

- Every variable in a C program has three properties:
  - Storage duration
  - Scope
  - Linkage

Properties of Variables

- The ***storage duration*** of a variable determines when memory is set aside for the variable and when that memory is released.
  - ***Automatic storage duration:*** Memory for variable is allocated when the surrounding block is executed and deallocated when the block terminates.
  - ***Static storage duration:*** Variable stays at the same storage location as long as the program is running, allowing it to retain its value indefinitely.

Properties of Variables

- The ***scope*** of a variable is the portion of the program text in which the variable can be referenced.
  - ***Block scope:*** Variable is visible from its point of declaration to the end of the enclosing block.
  - ***File scope:*** Variable is visible from its point of declaration to the end of the enclosing file.

Properties of Variables

- The ***linkage*** of a variable determines the extent to which it can be shared.
  - ***External linkage:*** Variable may be shared by several (perhaps all) files in a program.
  - ***Internal linkage:*** Variable is restricted to a single file but may be shared by the functions in that file.
  - ***No linkage:*** Variable belongs to a single function and can't be shared at all.

Properties of Variables

- The default storage duration, scope, and linkage of a variable depend on where it's declared:
  - Variables declared *inside* a block (including a function body) have *automatic* storage duration, *block* scope, and *no* linkage.
  - Variables declared *outside* any block, at the outermost level of a program, have *static* storage duration, *file* scope, and *external* linkage.

Properties of Variables

- Example:

```
int i;        static storage duration
              file scope
              external linkage

void f(void)
{
  int j;      automatic storage duration
              block scope
              no linkage
}
```

- We can alter these properties by specifying an explicit storage class: `auto`, `static`, `extern`, or `register`.

The `auto` Storage Class

- The `auto` storage class is legal only for variables that belong to a block.
- An `auto` variable has automatic storage duration, block scope, and no linkage.
- The `auto` storage class is almost never specified explicitly.

The `static` Storage Class

- The `static` storage class can be used with all variables, regardless of where they're declared.
  - When used *outside* a block, `static` specifies that a variable has internal linkage.
  - When used *inside* a block, `static` changes the variable's storage duration from automatic to static.

The `static` Storage Class

- Example:

```
static int i;

void f(void)
{
  static int j;
}
```

`static int i;`: static storage duration, file scope, **internal linkage**

`static int j;`: **static storage duration**, block scope, no linkage

The `static` Storage Class

- When used outside a block, `static` hides a variable within a file:

```
static int i;  /* no access to i in other files */

void f1(void)
{
  /* has access to i */
}

void f2(void)
{
  /* has access to i */
}
```

- This use of `static` is helpful for implementing information hiding.

The `static` Storage Class

- A `static` variable declared within a block resides at the same storage location throughout program execution.
- A `static` variable retains its value indefinitely.
- Properties of `static` variables:
  - A `static` variable is initialized only once, prior to program execution.
  - A `static` variable declared inside a function is shared by all calls of the function, including recursive calls.
  - A function may return a pointer to a `static` variable.

The `static` Storage Class

- Declaring a local variable to be `static` allows a function to retain information between calls.
- More often, we'll use `static` for reasons of efficiency:

```
char digit_to_hex_char(int digit)
{
  static const char hex_chars[16] =
    "0123456789ABCDEF";

  return hex_chars[digit];
}
```

- Declaring `hex_chars` to be `static` saves time, because `static` variables are initialized only once.

The `extern` Storage Class

- The `extern` storage class enables several source files to share the same variable.
- A variable declaration that uses `extern` doesn't cause memory to be allocated for the variable:

```
extern int i;
```

  In C terminology, this is not a *definition* of `i`.
- An `extern` declaration tells the compiler that we need access to a variable that's defined elsewhere.
- A variable can have many *declarations* in a program but should have only one *definition*.

The `extern` Storage Class

- There's one exception to the rule that an `extern` declaration of a variable isn't a definition.
- An `extern` declaration that initializes a variable serves as a definition of the variable.
- For example, the declaration

```
extern int i = 0;
```

  is effectively the same as

```
int i = 0;
```

- This rule prevents multiple `extern` declarations from initializing a variable in different ways.

The `extern` Storage Class

- A variable in an `extern` declaration always has static storage duration.
- If the declaration is inside a block, the variable has block scope; otherwise, it has file scope:

```
                 static storage duration
extern int i;    file scope
                 ? linkage

void f(void)
{
                     static storage duration
  extern int j;      block scope
                     ? linkage
}
```

The `extern` Storage Class

- Determining the linkage of an `extern` variable is a bit harder.
  - If the variable was declared `static` earlier in the file (outside of any function definition), then it has internal linkage.
  - Otherwise (the normal case), the variable has external linkage.

The `register` Storage Class

- Using the `register` storage class in the declaration of a variable asks the compiler to store the variable in a register.
- A ***register*** is a high-speed storage area located in a computer's CPU.
- Specifying the storage class of a variable to be `register` is a request, not a command.
- The compiler is free to store a `register` variable in memory if it chooses.

The `register` Storage Class

- The `register` storage class is legal only for variables declared in a block.
- A `register` variable has the same storage duration, scope, and linkage as an `auto` variable.
- Since registers don't have addresses, it's illegal to use the `&` operator to take the address of a `register` variable.
- This restriction applies even if the compiler has elected to store the variable in memory.

The `register` Storage Class

- `register` is best used for variables that are accessed and/or updated frequently.
- The loop control variable in a `for` statement is a good candidate for `register` treatment:

```
int sum_array(int a[], int n)
{
  register int i;
  int sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];
  return sum;
}
```

The `register` Storage Class

- `register` isn't as popular as it once was.
- Many of today's compilers can determine automatically which variables would benefit from being kept in registers.
- Still, using `register` provides useful information that can help the compiler optimize the performance of a program.
- In particular, the compiler knows that a `register` variable can't have its address taken, and therefore can't be modified through a pointer.

The Storage Class of a Function

- Function declarations (and definitions) may include a storage class.
- The only options are `extern` and `static`:
  - `extern` specifies that the function has external linkage, allowing it to be called from other files.
  - `static` indicates internal linkage, limiting use of the function's name to the file in which it's defined.
- If no storage class is specified, the function is assumed to have external linkage.

The Storage Class of a Function

- Examples:

```
extern int f(int i);
static int g(int i);
int h(int i);
```

- Using `extern` is unnecessary, but `static` has benefits:
  - ***Easier maintenance.*** A `static` function isn't visible outside the file in which its definition appears, so future modifications to the function won't affect other files.
  - ***Reduced "name space pollution."*** Names of `static` functions don't conflict with names used in other files.

The Storage Class of a Function

- Function parameters have the same properties as `auto` variables: automatic storage duration, block scope, and no linkage.
- The only storage class that can be specified for parameters is `register`.

Summary

- A program fragment that shows all possible ways to include—or omit—storage classes in declarations of variables and parameters:

```
int a;
extern int b;
static int c;

void f(int d, register int e)
{
  auto int g;
  int h;
  static int i;
  extern int j;
  register int k;
}
```


Summary

Name	***Storage Duration***	***Scope***	***Linkage***
`a`	static	file	external
`b`	static	file	†
`c`	static	file	internal
`d`	automatic	block	none
`e`	automatic	block	none
`g`	automatic	block	none
`h`	automatic	block	none
`i`	static	block	none
`j`	static	block	†
`k`	automatic	block	none

†In most cases, `b` and `j` will be defined in another file and will have external linkage.

Summary

- Of the four storage classes, the most important are `static` and `extern`.
- `auto` has no effect, and modern compilers have made `register` less important.

Type Qualifiers

- There are two type qualifiers: `const` and `volatile`.
  - C99 has a third type qualifier, `restrict`, which is used only with pointers.
- `volatile` is discussed in Chapter 20.
- `const` is used to declare "read-only" objects.
- Examples:

```
const int n = 10;
const int tax_brackets[] =
  {750, 2250, 3750, 5250, 7000};
```

Type Qualifiers

- Advantages of declaring an object to be `const`:
  - Serves as a form of documentation.
  - Allows the compiler to check that the value of the object isn't changed.
  - Alerts the compiler that the object can be stored in ROM (read-only memory).

Type Qualifiers

- It might appear that `const` serves the same role as the `#define` directive, but there are significant differences between the two features.
- `#define` can be used to create a name for a numerical, character, or string constant, but `const` can create read-only objects of *any* type.
- `const` objects are subject to the same scope rules as variables; constants created using `#define` aren't.

Type Qualifiers

- The value of a `const` object, unlike the value of a macro, can be viewed in a debugger.
- Unlike macros, `const` objects can't be used in constant expressions:

```
const int n = 10;
int a[n];              /*** WRONG ***/
```

- It's legal to apply the address operator (`&`) to a `const` object, since it has an address; a macro doesn't have an address.

Type Qualifiers

- There are no absolute rules that dictate when to use `#define` and when to use `const`.
- `#define` is good for constants that represent numbers or characters.

Declarators

- In the simplest case, a declarator is just an identifier:

```
int i;
```

- Declarators may also contain the symbols `*`, `[]`, and `()`.
- A declarator that begins with `*` represents a pointer:

```
int *p;
```


Declarators

- A declarator that ends with `[]` represents an array:

```
int a[10];
```

- The brackets may be left empty if the array is a parameter, if it has an initializer, or if its storage class is `extern`:

```
extern int a[];
```

- In the case of a multidimensional array, only the first set of brackets can be empty.

Declarators

- C99 provides two additional options for what goes between the brackets in the declaration of an array parameter:
  - The keyword `static`, followed by an expression that specifies the array's minimum length.
  - The `*` symbol, which can be used in a function prototype to indicate a variable-length array argument.
- Chapter 9 discusses both features.

Declarators

- A declarator that ends with `()` represents a function:

```
int abs(int i);
void swap(int *a, int *b);
int find_largest(int a[], int n);
```

- C allows parameter names to be omitted in a function declaration:

```
int abs(int);
void swap(int *, int *);
int find_largest(int [], int);
```

Declarators

- The parentheses can even be left empty:

```
int abs();
void swap();
int find_largest();
```

  This provides no information about the arguments.
- Putting the word `void` between the parentheses is different: it indicates that there are no arguments.
- The empty-parentheses style doesn't let the compiler check whether function calls have the right arguments.

Declarators

- Declarators in actual programs often combine the `*`, `[]`, and `()` notations.
- An array of 10 pointers to integers:

```
int *ap[10];
```

- A function that has a `float` argument and returns a pointer to a `float`:

```
float *fp(float);
```

- A pointer to a function with an `int` argument and a `void` return type:

```
void (*pf)(int);
```


Deciphering Complex Declarations

- But what about declarators like the one in the following declaration?

```
int *(*x[10])(void);
```

- It's not obvious whether `x` is a pointer, an array, or a function.

Deciphering Complex Declarations

- Rules for understanding declarations:
  - ***Always read declarators from the inside out.*** Locate the identifier that's being declared, and start deciphering the declaration from there.
  - ***When there's a choice, always favor `[]` and `()` over `*`.*** Parentheses can be used to override the normal priority of `[]` and `()` over `*`.

Deciphering Complex Declarations

- Example 1:

  `int *ap[10];`

  `ap` is an *array* of *pointers*.
- Example 2:

  `float *fp(float);`

  `fp` is a *function* that returns a *pointer*.

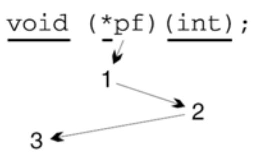
Deciphering Complex Declarations

- Example 3:

  `void (*pf)(int);`
- Since `*pf` is enclosed in parentheses, `pf` must be a pointer.
- But `(*pf)` is followed by `(int)`, so `pf` must point to a function with an `int` argument.
- The word `void` represents the return type of this function.

Deciphering Complex Declarations

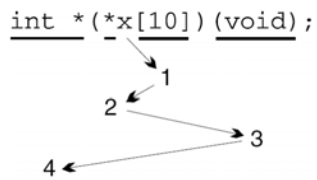
- Understanding a complex declarator often involves zigzagging from one side of the identifier to the other:



Type of `pf`:
1. pointer to
2. function with `int` argument
3. returning `void`

Deciphering Complex Declarations

- A second example of "zigzagging":



Type of `x`:
1. array of
2. pointers to
3. functions with no arguments
4. returning pointer to `int`

Deciphering Complex Declarations

- Certain things can't be declared in C.
- Functions can't return arrays:

```
int f(int)[];        /*** WRONG ***/
```

- Functions can't return functions:

```
int g(int)(int);     /*** WRONG ***/
```

- Arrays of functions aren't possible, either:

```
int a[10](int);      /*** WRONG ***/
```

- In each case, pointers can be used to get the desired effect.
- For example, a function can't return an array, but it can return a *pointer* to an array.

Using Type Definitions to Simplify Declarations

- Some programmers use type definitions to help simplify complex declarations.
- Suppose that `x` is declared as follows:

```
int *(*x[10])(void);
```

- The following type definitions make `x`'s type easier to understand:

```
typedef int *Fcn(void);
typedef Fcn *Fcn_ptr;
typedef Fcn_ptr Fcn_ptr_array[10];
Fcn_ptr_array x;
```


Initializers

- For convenience, C allows us to specify initial values for variables as we're declaring them.
- To initialize a variable, we write the = symbol after its declarator, then follow that with an initializer.

Initializers

- The initializer for a simple variable is an expression of the same type as the variable:

```
int i = 5 / 2;    /* i is initially 2 */
```

- If the types don't match, C converts the initializer using the same rules as for assignment:

```
int j = 5.5;      /* converted to 5 */
```

- The initializer for a pointer variable must be an expression of the same type or of type `void *`:

```
int *p = &i;
```

Initializers

- The initializer for an array, structure, or union is usually a series of values enclosed in braces:

```
int a[5] = {1, 2, 3, 4, 5};
```

- In C99, brace-enclosed initializers can have other forms, thanks to designated initializers.

Initializers

- An initializer for a variable with static storage duration must be constant:

```
#define FIRST 1
#define LAST 100

static int i = LAST - FIRST + 1;
```

- If `LAST` and `FIRST` had been variables, the initializer would be illegal.

Initializers

- If a variable has automatic storage duration, its initializer need not be constant:

```
int f(int n)
{
  int last = n - 1;
  …
}
```


Initializers

- A brace-enclosed initializer for an array, structure, or union must contain only constant expressions:

```
#define N 2

int powers[5] =
  {1, N, N * N, N * N * N, N * N * N * N};
```

- If `N` were a variable, the initializer would be illegal.
- In C99, this restriction applies only if the variable has static storage duration.

Initializers

- The initializer for an automatic structure or union can be another structure or union:

```
void g(struct part part1)
{
  struct part part2 = part1;
  …
}
```

- The initializer doesn't have to be a variable or parameter name, although it does need to be an expression of the proper type.

Uninitialized Variables

- The initial value of a variable depends on its storage duration:
  - Variables with *automatic* storage duration have no default initial value.
  - Variables with *static* storage duration have the value zero by default.
- A static variable is correctly initialized based on its type, not simply set to zero bits.
- It's better to provide initializers for static variables rather than rely on the fact that they're guaranteed to be zero.

Inline Functions (C99)

- C99 function declarations may contain the keyword `inline`.
- `inline` is related to the concept of the "overhead" of a function call—the work required to call a function and later return from it.
- Although the overhead of a function call slows the program by only a tiny amount, it may add up in certain situations.

Inline Functions (C99)

- In C89, the only way to avoid the overhead of a function call is to use a parameterized macro.
- C99 offers a better solution to this problem: create an ***inline function.***
- The word "inline" suggests that the compiler replaces each call of the function by the machine instructions for the function.
- This technique may cause a minor increase in the size of the compiled program.

Inline Functions (C99)

- Declaring a function to be `inline` doesn't actually force the compiler to "inline" the function.
- It suggests that the compiler should try to make calls of the function as fast as possible, but the compiler is free to ignore the suggestion.

Inline Definitions (C99)

- An inline function has the keyword `inline` as one of its declaration specifiers:

```
inline double average(double a, double b)
{
  return (a + b) / 2;
}
```

- `average` has external linkage, so other source files may contain calls of `average`.
- However, the definition of `average` isn't an external definition (it's an ***inline definition*** instead).
- Attempting to call `average` from another file will be considered an error.

Inline Definitions (C99)

- There are two ways to avoid this error.
- One option is to add the word `static` to the function definition:

```
static inline double average(double a, double b)
{
  return (a + b) / 2;
}
```

- `average` now has internal linkage, so it can't be called from other files.
- Other files may contain their own definitions of `average`, which might be the same or different.

Inline Definitions (C99)

- The other option is to provide an external definition for `average` so that calls are permitted from other files.
- One way to do this is to write the `average` function a second time (without using `inline`) and put this definition in a different source file.
- However, it's not a good idea to have two versions of a function: we can't guarantee that they'll remain consistent when the program is modified.

Inline Definitions (C99)

- A better approach is to put the inline definition of `average` in a header file:

```
#ifndef AVERAGE_H
#define AVERAGE_H

inline double average(double a, double b)
{
  return (a + b) / 2;
}

#endif
```

- Let's name this file `average.h`.

Inline Definitions (C99)

- Next, we'll create a matching source file, `average.c`:

```
#include "average.h"

extern double average(double a, double b);
```

- Any file that needs to call the `average` function can include `average.h`.
- The definition of `average` included from `average.h` will be treated as an external definition in `average.c`.

Inline Definitions (C99)

- A general rule: If all top-level declarations of a function in a file include `inline` but not `extern`, then the definition of the function in that file is inline.
- If the function is used anywhere in the program, an external definition of the function will need to be provided by some other file.

Inline Definitions (C99)

- When an inline function is called, the compiler has a choice:
  - Perform an ordinary call (using the function's external definition).
  - Perform inline expansion (using the function's inline definition).
- Because the choice is left to the compiler, it's crucial that the two definitions be consistent.
- The technique just discussed (using the `average.h` and `average.c` files) guarantees that the definitions are the same.

Restrictions on Inline Functions (C99)

- Restrictions on inline functions with external linkage:
  - May not define a modifiable `static` variable.
  - May not contain references to variables with internal linkage.
- Such a function is allowed to define a variable that is both `static` and `const`.
- However, each inline definition of the function may create its own copy of the variable.

Using Inline Functions with GCC (C99)

- Some compilers, including GCC, supported inline functions prior to the C99 standard.
- Their rules for using inline functions may vary from the standard.
- The scheme described earlier (using the `average.h` and `average.c` files) may not work with these compilers.
- Version 4.3 of GCC is expected to support inline functions in the way described in the C99 standard.

Using Inline Functions with GCC (C99)

- Functions that are specified to be both `static` and `inline` should work fine, regardless of the version of GCC.
- This strategy is legal in C99 as well, so it's the safest bet.
- A `static inline` function can be used within a single file or placed in a header file and included into any source file that needs to call the function.

Using Inline Functions with GCC (C99)

- A technique for sharing an inline function among multiple files that works with older versions of GCC but conflicts with C99:
  - Put a definition of the function in a header file.
  - Specify that the function is both `extern` and `inline`.
  - Include the header file into any source file that contains a call of the function.
  - Put a second copy of the definition—without the words `extern` and `inline`—in one of the source files.
- A final note about GCC: Functions are "inlined" only when the `-O` command-line option is used.