CS265
Advanced Programming Techniques

C Declarations

Properties of variables

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

Properties of Variables

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

For how long does the storage for the variable persist?

- The storage duration 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

Storage Duration

Example:

```
int i=15;

void f(void)
{
   int j=20;

   for (int i=0; i<10; i++)
   {
      printf("%d\n", i);
   }

   printf("%d\n", j);
   printf("%d\n", i);
}
```

`int i=15;` →

- ✓ static storage duration
- ✓ persists while the program is running

`int j=20;` →

- ✓ automatic storage duration
- ✓ storage allocated when the function starts
- ✓ storage deallocated when the function ends

`for (int i=0; i<10; i++)` →

- ✓ automatic storage duration
- ✓ storage allocated when the block starts
- ✓ storage deallocated when the block ends

Properties of Variables

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

Where in the file is the variable visible?

- 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

Scope

Example:

```
int i=15;

void f(void)
{
   int j=20;

   for (int i=0; i<10; i++)
   {
       printf(“%d\n”, i);
   }

   printf(“%d\n”, j);
   printf(“%d\n”, i);
}
```

- ✓ file scope
- ✓ block scope
- ✓ block scope

Properties of Variables

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

What files can share the variable?

- 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

Linkage

Example:

```
int i=15;
```

- ✓ external linkage
- ✓ may be shared with other files

```
void f(void)
{
   int j=20;
```

- ✓ no linkage
- ✓ can't be shared

```
   for (int i=0; i<10; i++)
   {
       printf("%d\n", i);
   }

   printf("%d\n", j);
   printf("%d\n", i);
}
```

- ✓ no linkage
- ✓ can't be shared

Properties of Variables

- Example:

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

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

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

Declaring vs Defining in C

- Subtle but distinct difference between declaring and defining a variable or function
- When you declare something you are telling the compiler that there is something with that type and that name but not all details are given
- When you define something, you are giving full details
- Particularly useful when you are working with multiple files
  - e.g. you may declare a function in every file, but you define it only in one file

Defining vs Declaring Functions

- Function definition

```
int func()
{
  return 2;
}
```

- Function declaration

```
int func();
```

Defining vs Declaring Variables

- Variable definition and declaration
- The variable `x` is stored in the storage associated with the file that contains this code (Data Segment of memory)

```
int x;

int main()
{
  x = 3;
}
```

- Variable declaration only
- The storage of the variable is somewhere else

```
extern int x;

int f()
{
  x = 4;
}
```

Defining vs Declaring Variables

- Variable declaration and definition in the same file

```
extern int x;

int func()
{
  x = 4;
}

int x;
```

Declaration Syntax

storage-class type-qualifier type-specifier function-specifier declarators ;

Declaration Syntax – Declarators

storage-class type-qualifier type-specifier function-specifier declarators ;

Required, separated by commas, may be followed by an initializer

- Identifiers (names of simple variables) `int i;`
- Identifiers preceded by `*` (pointer names) `int *p;`
- Identifiers followed by `[]` (array names) `int a[10];`
- Identifiers followed by `()` (function names) `float f(float);`

Declaration Syntax – Storage Class

storage-class type-qualifier type-specifier function-specifier declarators ;

optional but if present, it should come first

`static` and `extern` are the most important

`auto`

`static`

`extern`

`register`

Declaration Syntax – Type Qualifier

storage-class *type-qualifier* *type-specifier function-specifier declarators* ;

optional

`const`

`volatile`

`restrict`

Declaration Syntax – Type Specifier

storage-class type-qualifier type-specifier function-specifier declarators ;

required

`void`	`struct`
`char`	`union`
`short`	`enum`
`int`	`typedef`
`long`	
`float`	
`double`	
`signed`	
`unsigned`	

Declaration Syntax – Function Specifier

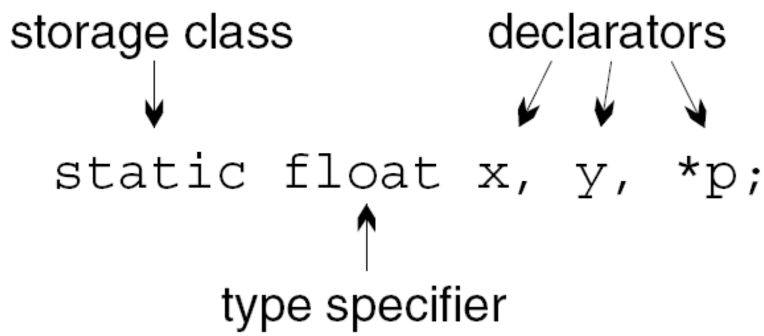
storage-class type-qualifier type-specifier function-specifier declarators ;

only applies to functions

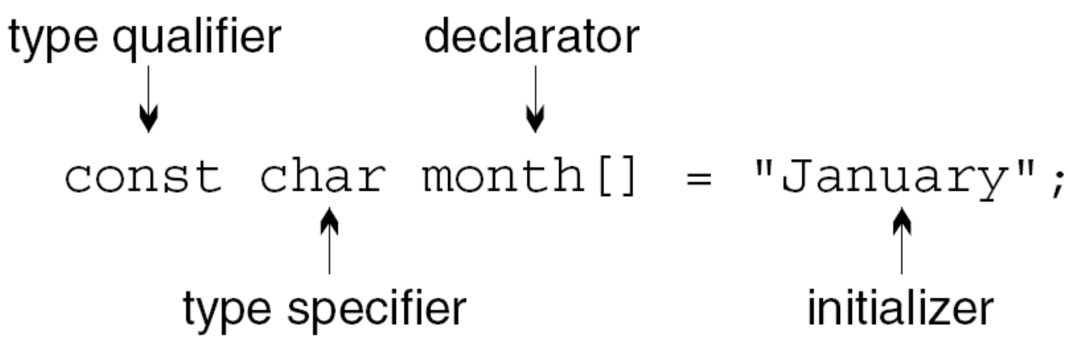
`inline`

Declaration Syntax - Examples

- A declaration with a storage class and three declarators:



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



Declaration Syntax - Examples

- 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

The `auto` Storage Class

Almost never used

- 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.

This

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

is the same as this

```
void f(void)

{

  auto int j;

}
```

Super important

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:

```c
int i;
```

static storage duration
file scope
external linkage

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

automatic storage duration
block scope
no linkage

```c
static int i;
```

static storage duration
file scope
internal linkage

```c
void f(void)
{
   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

Example

```
int foo()
{
  static int count = 0;
  return count++;
}
```

- Try calling this repeatedly, perhaps from several different functions or files even, and you'll see that count keeps increasing, because in this case `static` gives the variable a lifetime equal to that of the entire execution of the program.

`strtok()` function

```
char* getfield(char* line, int num) {
   char * token;
   int token_count = 1;


   /* get the first token from the line */
   token = strtok(line, ",");


   /* walk through other tokens */


   while (token != NULL) {
      if (num == token_count)
         return(token);
      token = strtok (NULL, ",");
      token_count ++;
   }
   return NULL;
}
```

How does `strtok` know where it is in the line?

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.

Super important

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:

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

void f(void)
{
  extern int j;    static storage duration
                   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

Not that important anymore

- 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.
- It 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`

Example – Storage Duration

For how long does the storage for the variable persist?

```
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;
}
```

Storage Duration = static

Scope Duration = automatic

Where in the file is the variable visible?

Example - Scope

```
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;
}
```

Scope = file

Scope = block

Example - Linkage

What files can share the variable?

```
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;
}
```

Linkage = external

Linkage = internal

Linkage = defined elsewhere, most often it will be external

Linkage = none

Example – in Summary

```
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;
}
```

Name	***Storage Dur.***	***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.

Type Qualifiers

- There are three type qualifiers:

```
const
volatile
restrict
```

The `const` Type Qualifier

- `const` is used to declare "read-only" objects of any type
- Examples:

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

- 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)

The `const` Type Qualifier

- 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.
- 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.

The `const` Type Qualifier

- 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.

The `volatile` Type Qualifier

- For low level programming
- On some computers, certain memory locations are “volatile”
- The value stored at such a location can change as a program is running, even though the program itself isn’t storing new values there
- For example, when holding data coming directly from input devices.
  - The most recent character typed at the keyboard

The `restrict` Type Qualifier

- Used only with pointers
- When we use `restrict` with a pointer `p`, we tell the compiler that `p` is the only way to access the object
- Cannot use a different pointer to access the object (cannot use pointer aliasing)
- The compiler can use this knowledge for optimizations

Function 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);
```

- 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.

```
void f(void);
```

Declaration Syntax – Declarators

storage-class type-qualifier type-specifier function-specifier declarators ;

Only used with functions – can have only one value:

`inline`

Inline Functions

- `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
- The word `inline` suggests that the compiler replaces each call of the function by the machine instructions for the function
- Declaring a function to be `inline` doesn't 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
- There are lots of restrictions about how to use inline functions across files (check the book!)

Lessons

- Lesson 1: Learn C to become a power programmer
- Lesson 2: C / C++ are the defacto systems programming languages

Resources

- These notes
- *C Programming: A modern Approach* by K.N. King, 2008
- Chapter 18