Declarations

Based on slides from K. N. King

Bryn Mawr College
CS246 Programming Paradigm

Declaration Syntax

- 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 Specifiers

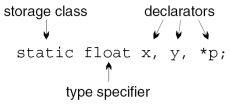
- Declaration specifiers fall into three categories:
  - Storage classes (at most one; if present, should come first)
    - `auto`, `static`, `extern`, and `register`.
  - Type qualifiers (zero or more)
  - Type specifiers
    - `E.g., void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`
    - specifications of structures, unions, and enumerations.
      `struct point { int x, y; },`
      `struct { int x, y; }, struct point.`
    - `typedef` names
- Type qualifiers and type specifiers should follow the storage class

Declarators

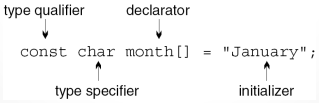
- Declarators include:
  - Identifiers (names of simple variables)
  - Identifiers followed by `[]` (array names)
  - Identifiers preceded by `*` (pointer names)
  - Identifiers followed by `()` (function names)
- Declarators are separated by commas.
- A declarator that represents a variable may be followed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

Declaration Examples

- A declaration with a storage class and three declarators:

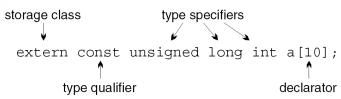


- A declaration with a type qualifier and initializer but no storage class:

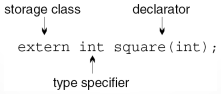


Declaration Examples

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



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



Properties of Variables

- Every variable in a C program has three properties:
  - **Storage duration** determines when memory is set aside for the variable and when that memory is released
  - **Scope** is the portion of the program text in which the variable can be referenced.
  - **Linkage** determines the extent to which a variable can be shared.

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.

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

```
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 across the entire run of the program.
- 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.

```
void func() {
    static int x = 0;
    printf("%d\n", x);
    x = x + 1;
}

int main() {
    func(); // prints 0
    func(); // prints 1
    func(); // prints 2
    return 0;
}
```

The `static` Storage Class

- 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; //` not a *definition* of `i`.
- A variable can have many *declarations* in a program but should have only one *definition*.

The `extern` Storage Class

Exception:

- 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

- Storage duration: always static
- Inside a block – block scope; otherwise, file scope.
- Linkage: if the variable was declared `static` earlier in the file (outside of any function definition) – internal linkage; otherwise, external linkage.

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

void f(void)
{
  extern int j;    // static storage duration, block scope, ? 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

- 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

- `const` is used to declare "read-only" objects.
- Examples:

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

Declarators

- Identifiers (names of simple variables)
  - Simplest case: a declarator is just an identifier

```
int i;
```

- Identifiers followed by `[]` (array names)
- Identifiers preceded by `*` (pointer names)

```
int *p;
```

- Identifiers followed by `()` (function names)

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 `*`.
- Examples:

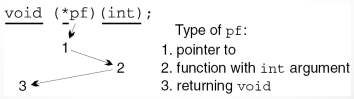
```
int *ap[10]; //ap is an array of pointers.
float *fp(float);
  //fp is a function that returns a pointer.
```

Deciphering Complex Declarations

- Example:

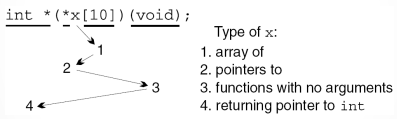
```
void (*pf)(int);
```

  - 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

- Example:
- `int *(*x[10])(void);`



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.

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};
```

- 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;
  …
}
```

- 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.

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.