HIGH-LEVEL PROGRAMMING I

Storage duration, Scope, Linkage by Prasanna Ghali

Questions of Interest For Writing Large(r) Programs

- How can variables defined in one source file be referenced and shared by authors of other source files?
- What happens if multiple source files define variables and functions with similar names?
- What happens if a source file defines multiple variables in different regions of program text with same name?
- How can functions implemented in one source file be shared by authors of other source files?
- How can all different pieces be connected together to ensure program compiles, links, and executes correctly?

Review: Declaration/Definition

- *Declaration* makes type and name of variable known to compiler
- *Definition* is special kind of declaration that causes memory to be reserved for variable and initializes memory with optional set of values
- Declaration can occur *multiple* times
- Definition must occur *only once* in program

Declaration statement

declaration-specifier *declarators* ;

properties of variables being declared

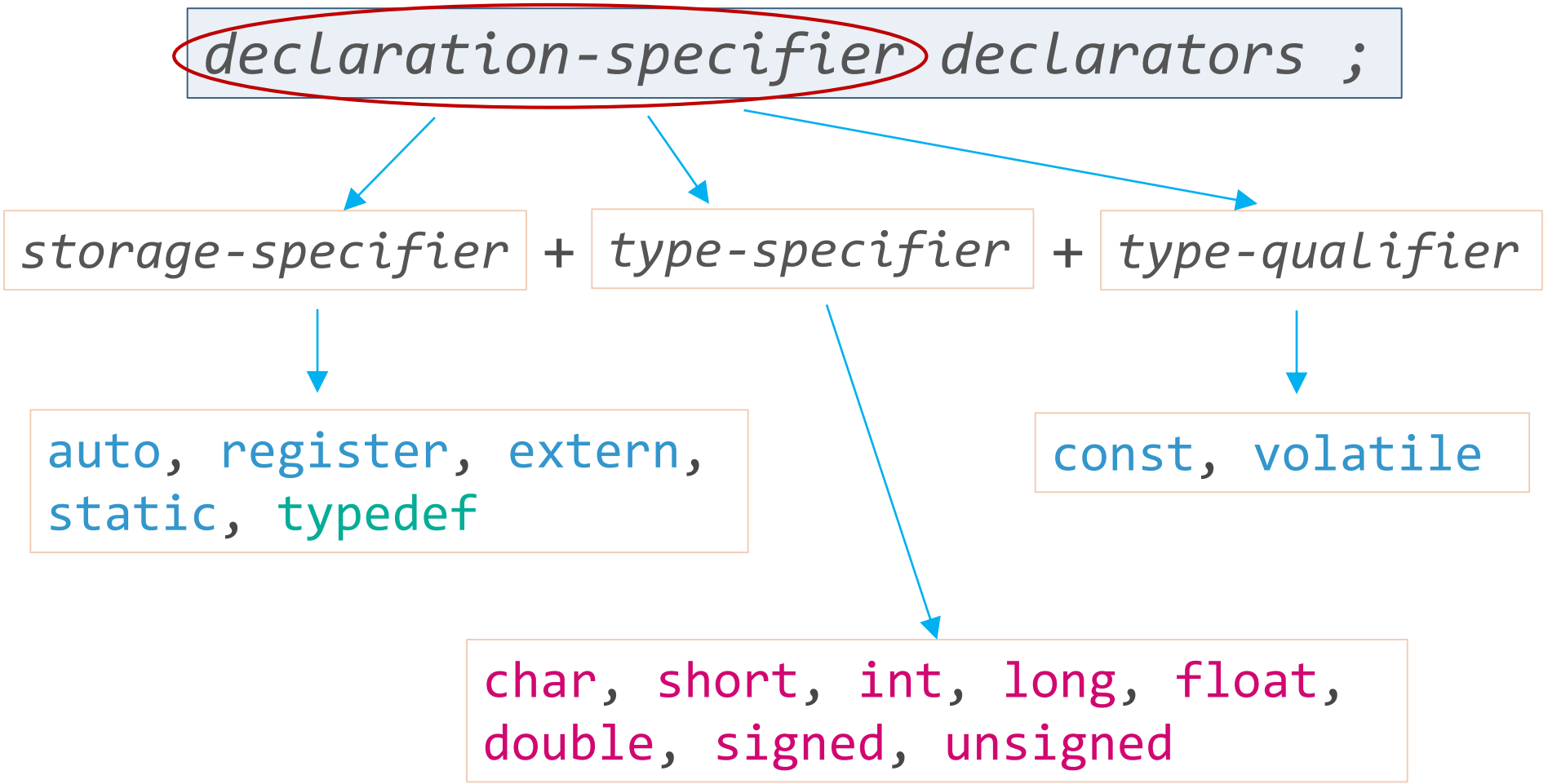
names being declared plus additional type information

```
int     x = 10;    // definition
float   f;         // definition
char    *pi;       // definition
double grades[10] = {1.1, 2.2}; // definition
```

Properties of variables

- In addition to typing information, every variable has 3 additional properties:
  - Storage duration
  - Scope
  - Linkage

Declaration statement



Storage duration of variable

- Determines *lifetime* of storage associated with variable
- Two kinds of variables: *internal* (or *local*) to code block and *external* (or *global*) to all blocks
- By default, internal variables have *automatic* storage duration
- By default, external variables have *static* storage duration

Automatic storage duration of variable

- Automatic variables are *local* or *internal* to function block
- Lifetime on *stack* begins when function is invoked and ends when function returns
- Don't retain their value from one call to next – must be initialized before use
- By default, all variables plus parameters in function or block have automatic storage duration

Storage specifier: `auto` keyword

- Automatic variables can be declared by adding `auto` storage specifier
- Irrelevant specifier because such variables get it by default
- Not to be used because keyword has different meaning in modern C++
- See *auto.c* for working example

Storage specifier: `register` keyword

- Automatic variables can be stored in CPU register using hints to compiler
- Storage specifier `register` is used to provide these hints
- Rarely used [except by embedded programmers] because CPUs too complex – best left to compilers
- See *register*.c for working example

Storage specifier: `static` keyword

- Internal variables that by default have automatic storage can be defined to have static storage duration using keyword `static`
- See *static-kwd.c* for working example

Static storage duration of variable

- Static variables can be internal to function or external to function
- Remain in existence throughout program duration - memory used to store static variables lasts for entire ***lifetime*** of program
  - Reside not in *stack* like automatic variables but in separate *data* section of memory
- When defined without initializer, zeroed out at program start up
- See *static-storage*.c for working example

What is Scope?

- Scope or *visibility* of variable is region of program text over which name is visible and can therefore be referenced by other entities
- Many types of scope: *file scope*, *function scope*, *block scope*
- Variables with same name but different scopes correspond to different memory locations
- Same variable name cannot be defined more than once in a scope

File Scope

- External variables are visible from declaration point to end of file
- Also known as *global variables* because they're visible in any functions defined after this variable's declaration

Function Scope

- Parameters and variables declared in function body have *function scope* and are known as *local variables*
  - Visible from point of declaration to end of function
  - Not visible outside function in which they're declared

Block Scope

- Delimited by curly braces inside function defines *block scope*
  - Applies also to compound statements within conditionals and loops
- Variable with block scope is visible from its point of declaration to end of block
- Variable in block scope hides identical name in outer scope

Scope: Examples (1/3)


```
void f1(int param) {  // scope of param starts here
  int a = 2;          // scope of a starts here
  int b = 10;         // scope of b starts here

  while (a < 10) {
    int x;            // scope of x starts here

    x = param * a++; // OK, param and a both in scope

    if (x == 5)
      b = 11;     // OK, b in scope
  }               // scope of x ends here

  x = 3;        // ERROR! x not in scope
}  // scope of a, b and param end here
```

Scope: Examples (2/3)


```
void f2(int param) {  // scope of param starts here
  int a = 2;     // scope of a starts here
  int param;     // ERROR! param already defined

  while (a < 10) {
    int x = 2;   // scope of x starts here
    double a;    // OK. Different scope, different a
    float param; // OK. Different scope, different param

    if (x == 5) {
      int x;     // OK. Different scope, different x
      int param; // OK. Different scope, different param
    } // scope of third x and param ends here
  }   // scope of second x, a and param ends here
}     // scope of first a and param ends here
```

Scope: Examples (3/3)


```
int a;  // Memory for a is allocated before program starts
        // Value of a is 0 (global variable)

void f3(int param) {
  int b;  // Memory for b is allocated when function starts
          // b is uninitialized (local variable)
  b = a * param; // OK, a and param in scope
}

int main(void) {
  int x;  // Memory for x is allocated when main starts
          // x is uninitialized because it is local variable)

  f3(x);  // Memory for copy of x is allocated when f3 starts
          // Memory is deallocated when f3 returns
  return 0;
}
// Memory for a is deallocated when program ends
```

Scoping rule

- Compiler will associate *closest declaration of an identifier* when references are made to variables and functions with same identifier
- See *scope.c* for working example

Linkage

- Linkage describes accessibility of variable between different source files or even within same source file
- Three types of linkage: *no linkage*, *external linkage*, *internal linkage*

No linkage

- Internal variables can only be accessed from inside function
- These internal variables of function have *no linkage* with other functions located in either same or in different source files

External linkage (1/5)

- External variables have default property that all references to them by same name, from any source file are references to same thing
- This means that compiler will export names of external variables to linker

External linkage (2/5)

- Individual source files compile but linker fails!!!
- External linkage implies program consisting of many source files must have *one and only one definition of external variable*

```c
// mine.c
int x = 10;

int main(void) {
  printf("mine:  %d\n", x);
  extern void foo(void);
  foo();
}
```

```c
// yours.c
int x = 20;

void foo(void) {
  printf("yours: %d\n", x);
}
```

External linkage (3/5)

- *yours.c* doesn't compile!!!

```c
// mine.c
int x = 10;

int main(void) {
  printf("mine:  %d\n", x);
  extern void foo(void);
  foo();
}
```

```c
// yours.c
int x = 20;

void foo(void) {
  printf("yours: %d\n", x);
}
```

External linkage (4/5)

- *yours.c* now compiles!!! Linker creates executable!!!

```
// mine.c
int x = 10;

int main(void) {
  printf("mine:  %d\n", x);
  extern void foo(void);
  foo();
}
```

```
// yours.c
int x = 20;
extern int x;

void foo(void) {
  printf("yours: %d\n", x);
}
```

External linkage (5/5)

- See *min-val.c* and *min-val-main.c* for working example of external linkage of variables

Internal linkage

- External variables have default property that references to them by same name, from any source file are references to same thing
- *Internal linkage* makes variable name visible to functions in same source file but not to other source files
- Keyword `static` shows up in different scenario to ensure *privateness* of variables
- See *mine*.c, *yours*.c, and *theirs*.c for working example

Functions: Storage duration

- Functions cannot be defined inside other functions
- Therefore, functions will have static storage duration
- Stored in region of program memory called *text* area

Functions: Scope

- Functions have file scope
- Scope of functions lasts from point at which it is declared to end of file being compiled
- If functions `main`, `x`, `y`, `z`, are defined in that order, and `main` calls these functions, then declarations of these names must be made before their first use in `main`

Functions: Linkage

- By default, functions have external linkage
- This means that function names are global, visible to any part of the entire program
- See *main-rev.c* and *rev.c* for working example

Review: Do We Know Answers?

- How can variables defined in one source file be referenced and shared by authors of other source files?
- What happens if multiple source files define variables and functions with similar names?
- What happens if a source file defines multiple variables in different regions of program text with same name?
- How can functions implemented in one source file be shared by authors of other source files?
- How can all different pieces be connected together to ensure program compiles, links, and executes correctly?