The Three Attributes of an Identifier

What is an identifier?

- name of a variable
- name of a type
- name of a function

Identifiers have three essential attributes:

- storage duration (variables only)
- scope
- linkage

Declarations and Definitions

A *declaration* specifies the interpretation and attributes of a set of identifiers.

A *definition* of an identifier is a declaration for that identifier that:

- for an object, causes storage to be reserved for that object;

- for a function, includes the function body;

- for an enumeration constant, is the (only) declaration of the identifier;

- for a `typedef` name, is the first (or only) declaration of the identifier.

Storage Duration

storage duration

determines when, during execution of a program, memory is set aside for the variable and when that memory is released

automatic duration

- storage is allocated when the surrounding block of code is executed
- storage is automatically deallocated when the block terminates

static duration

- storage is allocated when execution begins
- variable stays in the same storage location as long as the program is running
- variable can retain its value indefinitely (until program terminates)

Automatic Storage Duration

automatic duration

- storage is allocated when the surrounding block of code is executed
- storage is automatically deallocated when the block terminates

```
...
void Sort(int list[], int Sz) {

   int startIdx = 0;

   ...
}
```

default for variables that are declared inside a block

created (memory allocated) on each call

initialized on each call

deallocated (memory reclaimed) when call ends

Static Storage Duration

static duration

- storage is allocated when execution begins
- variable stays in the same storage location as long as the program is running
- variable can retain its value indefinitely (until program terminates)

```
int numCallsToSort = 0;

...


void Sort(int list[], int Sz) {

   static int numSwaps = 0;

   ...
}
```

default for variables declared outside all blocks

initialized once, keeps its value until program ends

variable is declared inside a block, with keyword `static`

initialized once, keeps its value from one call to the next

Scope

scope

(of an identifier) the range of program statements within which the name is recognized as a valid name

block scope

- name is visible from its point of declaration to the end of the enclosing block
- place declaration of name within a block

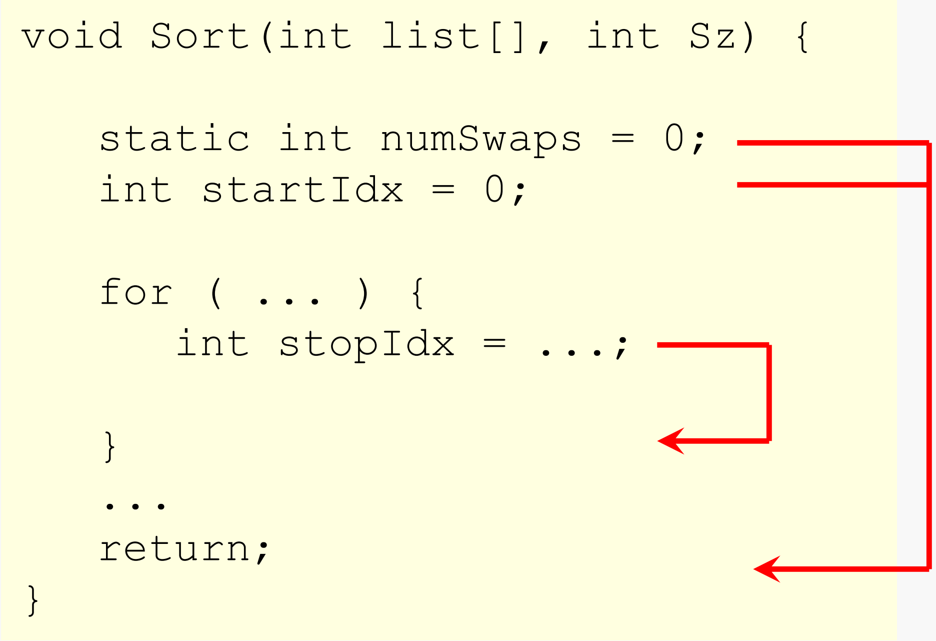
file scope

- name is visible from its point of declaration to the end of the enclosing file
- place declaration of name outside of all blocks (typically before any blocks)

Block Scope

block scope

- name is visible from its point of declaration to the end of the enclosing block
- place declaration of name within a block



```
void Sort(int list[], int Sz) {

   static int numSwaps = 0;
   int startIdx = 0;

   for ( ... ) {
      int stopIdx = ...;

   }
   ...
   return;
}
```

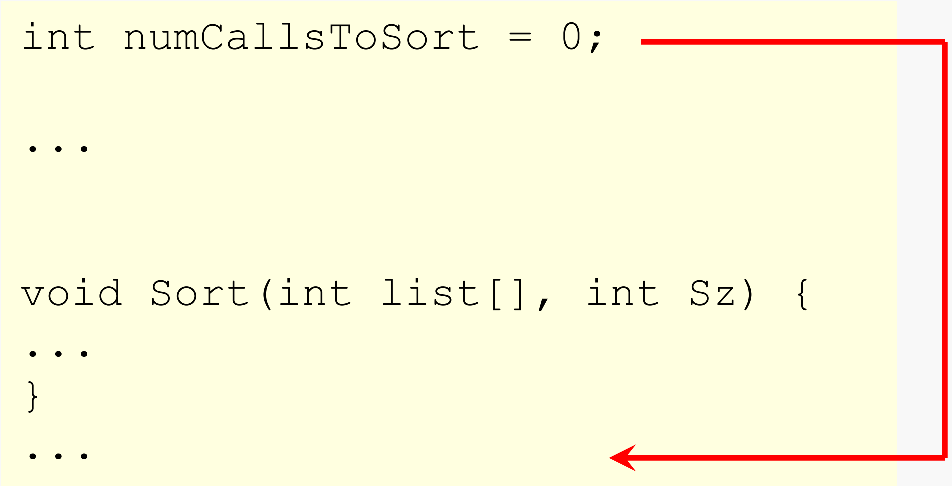
name is declared inside a block

name can only be referred to from declaration to end of block

File Scope

file scope

- name is visible from its point of declaration to the end of the enclosing file
- place declaration of name outside of all blocks (typically before any blocks)



```
int numCallsToSort = 0;

...


void Sort(int list[], int Sz) {
...
}
...
```

name declared outside all blocks

name can be referred to from any function within the file

potentially dangerous

avoid unless necessary

pass parameters instead

Linkage

linkage

determines the extent to which the name can be shared by different parts of the program

external linkage

- name may be shared by several (or all) files in the program

internal linkage

- name is restricted to a single file, but shared by all functions within that file

no linkage

- name is restricted to a single function

External Linkage

external linkage

- name may be shared by several (or all) files in the program

```
int numCallsToSort = 0;
...

void Sort(int list[], int Sz) {
...
}
```

name is declared outside all blocks

name can be referred to from other files

potentially very dangerous

use only if necessary

Internal Linkage

internal linkage

- name is restricted to a single file, but shared by all functions within that file

```
static int numCallsToSort = 0;
...

void Sort(int list[], int Sz) {
...
}
```

name is declared outside all blocks, using reserved word static

name cannot be referred to from other files

potentially dangerous

use only if necessary

No Linkage

no linkage

- name is restricted to a single function

```
...
void Sort(int list[], int Sz) {

   static int numSwaps = 0;
   int startIdx = 0;

   ...
}
```

name is declared inside a block

name can only be referred to within the block where the declaration is placed

Determining the Attributes

The default storage duration, scope and linkage of a variable depend on the location of its declaration:

inside a block

automatic storage duration, block scope, no linkage

outside any block

static storage duration, file scope, external linkage

When the defaults are not satisfactory, see:

```
auto
static
extern
register
```

Global Symbols?

global symbol

- name has external linkage
- references to the name are handled by the linker
- compiler and linker have relevant rules… for function names… for variable names

function names:

- functions not defined locally are assumed to be external; so compiler leaves resolving them to the linker
- triggers implicit declaration warning, and frequently link-time errors when actual function interface doesn't match implicit one
- global functions should be declared in header files

variable names:

- variables not declared locally are treated as errors
- must use "extern" declaration for global variables defined elsewhere

Function Names

```
static void func1( ) { . . . }
```

- defines file-local symbol `func1`; not global

```
void func2( ) { . . . }
```

- defines (strong) global symbol `func2`

```
static void func3( );
```

- defines no symbol; declares `func3`

```
void func4( );
extern void func4( );
```

- makes `e` an external reference

all examples are at file scope

Variable Names

```
static int x = 4;
static int y;
```

- defines file-local symbols `x` and `y`

```
int w;
```

- defines weak global symbol aka *common* symbol

```
int z = 4;
```

- defines strong global symbol

```
extern int ext;
```

- `ext` is defined somewhere else

all examples are at file scope

Use Case: Global Type Name

Rational.h

```
struct _Rational {
   int64_t top;
   int64_t bottom;
};
typedef struct _Rational Rational;
```

Data types are typically needed throughout an implementation, and so must be global.

The type declaration is usually placed in a suitable header file so it can be included as needed.

Use Case: Global Function Name

Rational.h

```
Rational Rational_Add(Rational left, Rational right);
```

Used for any function that needs to be called from other modules. Very common.

Place function declaration in suitable header file.

Use Case: Global Variable Name

```
// no example given . . . almost always a bad idea
```

Unlike types and functions, variables are mutable.

Making a variable global allows modifications to it to be made from anywhere.

Use Case: File-local Type Name

Index.c

```
struct _indexEntry {
   uint32_t location;
   int32_t  key;
};
typedef struct _indexEntry indexEntry;

static indexEntry Index[maxEntries];

uint32_t findEntry(int32_t keyValue) {
   . . .
}
```

Here, we create an array to index a collection of records.

The index uses objects that only make sense locally, so the type is file-local.

Use Case: File-local Variable Name

Index.c

```c
struct _indexEntry {
   uint32_t location;
   int32_t  key;
};
typedef struct _indexEntry indexEntry;

static indexEntry Index[maxEntries];

uint32_t findEntry(int32_t keyValue) {
   . . .
}
```

The array that holds the index entries is file-local, so various search and mutator functions can access it directly.

The search function shown here would be declared in a header file, so it is global.

Use Case: File-local Variable Name

Index.c

```
. . .

static indexEntry Index[maxEntries];
. . .
```

But… why make the array file-local?

Why not make it local to a function?

Answer: we need to populate the array with entries as we build the index, and the index (the array) needs to be persistent.

If we made the array local to a function

- it could only be accessed from within that function
- it would cease to exist when that function returned

Use Case: File-local Variable Name

Index.c

```
. . .

static indexEntry Index[maxEntries];
. . .
```

Why not make the array global?

Answer: we want to restrict modifications to the array.

If we made the array global, it could be changed from anywhere in the program.

Use Case: File-local Function Name

Rational.c

```
static Rational Rational_Reduce(Rational original);
. . .
static Rational Rational_Reduce(Rational original) {
   . . .
}
```

Used for any function that needs to be called only from the current module.

Place function declaration suitable `.c` file, and make the function `static`.