Abstract Data Types

Object Attributes

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Object attributes

- A variable has a name, a type, a size, and a value

```
char c;
int i;
float f;
list_t e;
```

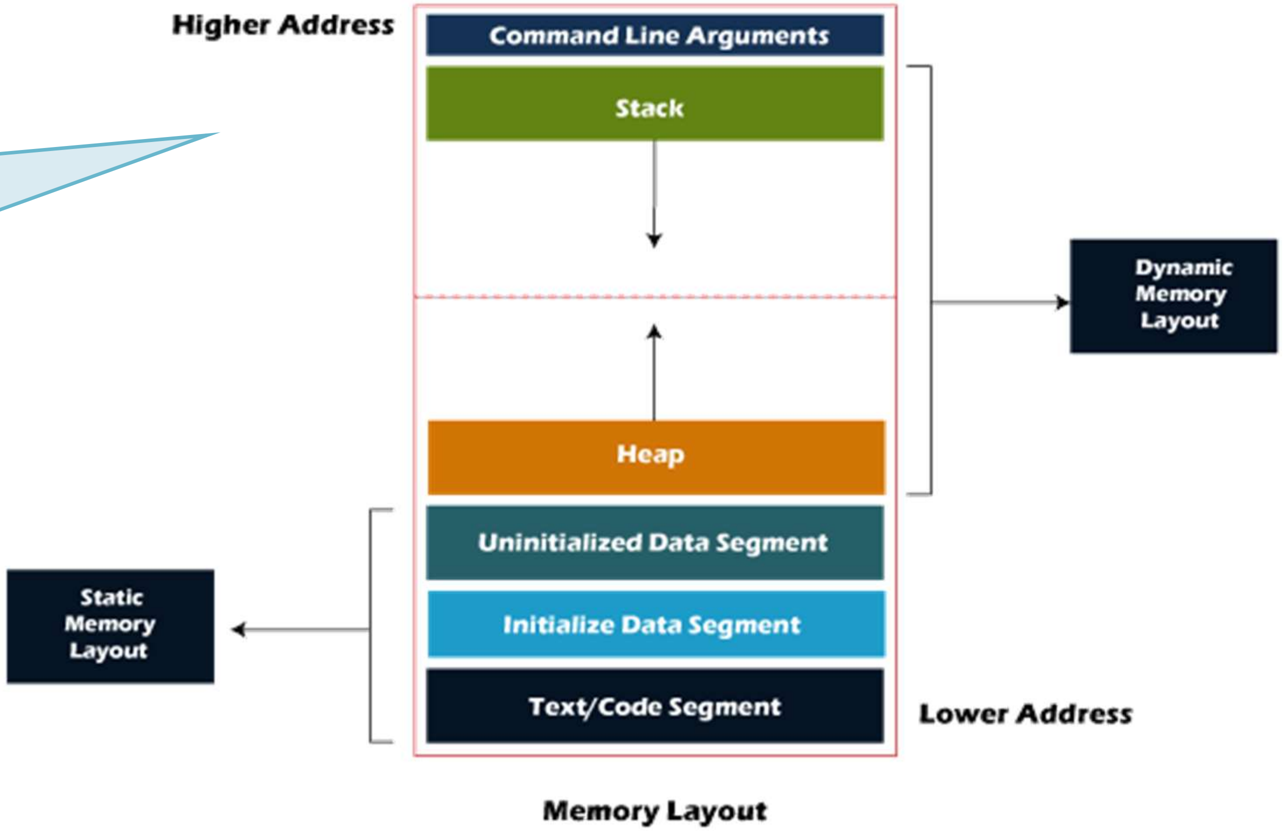
Name c, type char, size 1 byte, value undefined

- In general, each identifier, i.e., a variable or a function name, has 4 other attributes
  - Storage class, storage duration, scope, linkage

Object attributes

- Storage class
  - The part of the memory program where the object (variables or functions) is allocated

Object file
(*.c source file
after compilation)



Object attributes

- Storage duration
  - The time period during which the identifier exists in memory, i.e., how long the storage allocation continues to exist

global --> all duration of the program
local --> only when inside the function

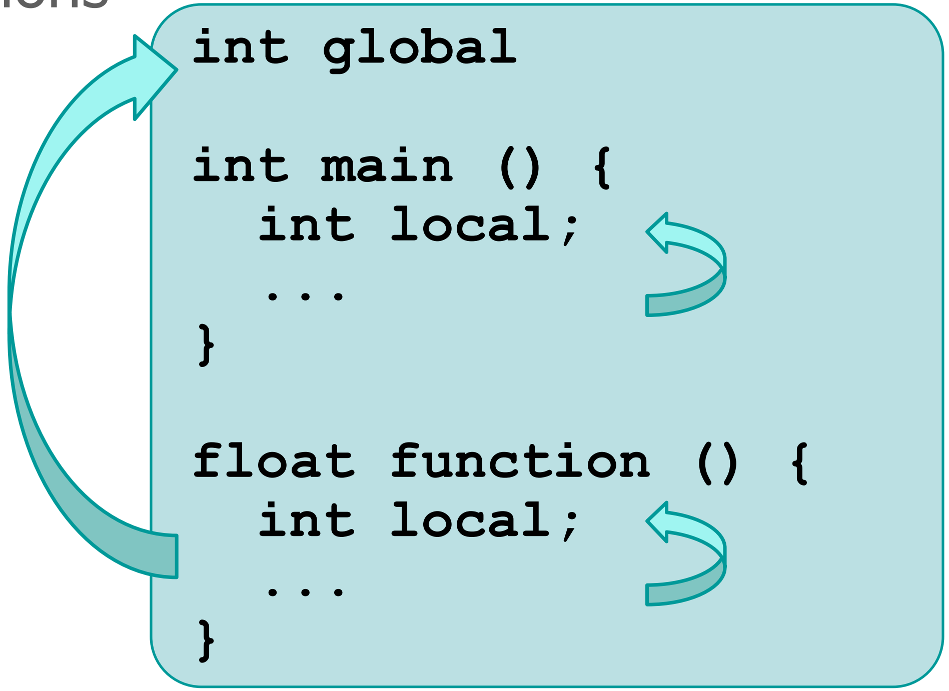
```
int global

int main () {
  int local;
  ...
}

float function () {
  int local;
  ...
}
```

Object attributes

- Scope
  - An identifiers scope is where the identifier can be referenced in a program
    - Some can be referenced throughout the entire program
    - Others from only portions of a program



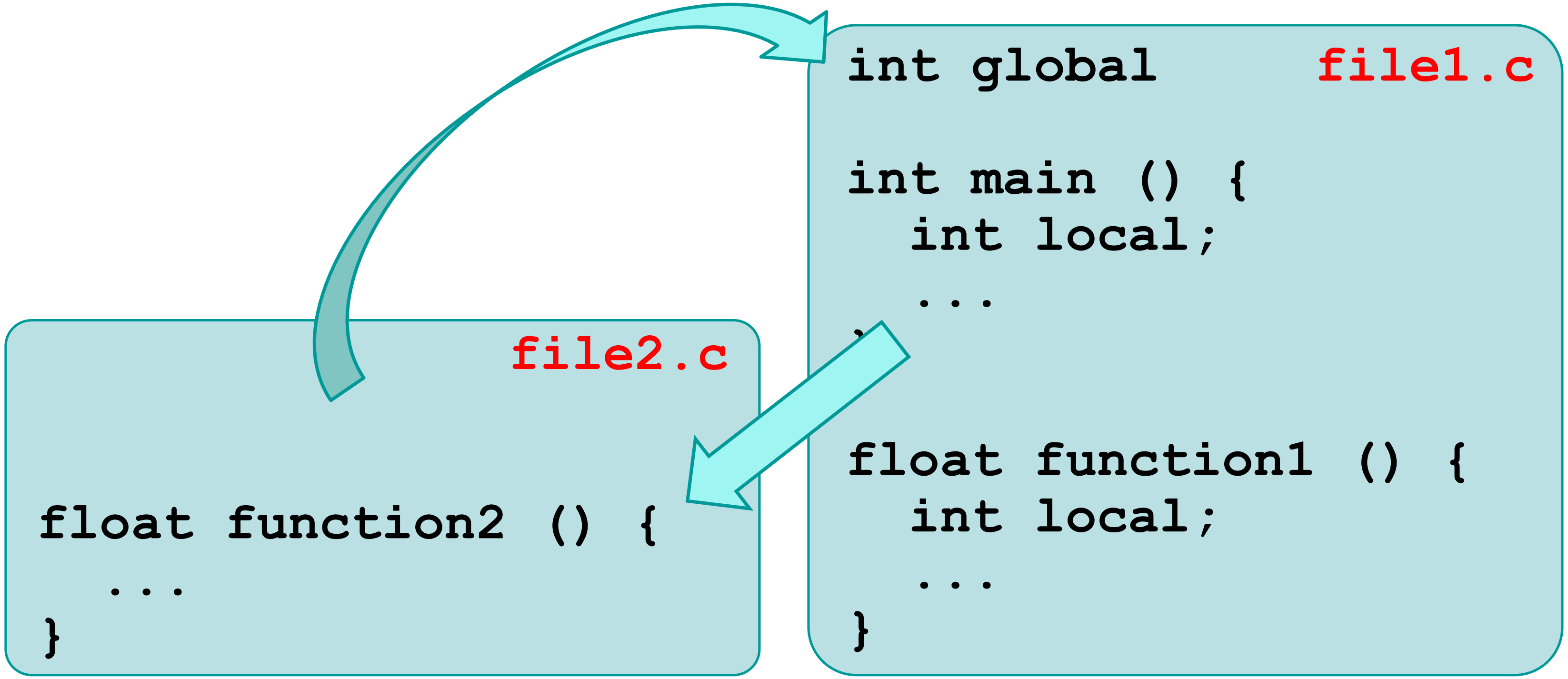
```
int global

int main () {
  int local;
  ...
}

float function () {
  int local;
  ...
}
```

Object attributes

- Linkage
  - An identifiers linkage determines for a multiple-source-file program whether the identifier is known only in the current source file or in any source file with proper declarations



file1.c

```
int global

int main () {
  int local;
  ...
}

float function1 () {
  int local;
  ...
}
```

file2.c

```
float function2 () {
  ...
}
```

Storage class and duration

- C provides four storage class specifiers
  - Automatic (or auto) what we have used so far
    - Only for variables, not for functions
    - Default in many cases
  - Register arcaic
    - Only for variables
    - Old-fashion, rarely used today
  - Extern new
    - Default for functions
    - May be indicated explicitly for variables
  - Static
    - Used for both variables and functions

Hug?
What do they mean?

Storage class and duration

- The **automatic** (or auto) storage class is the most common one, being the **default** for all local variables
  - Local variables have automatic storage duration by default
  - The keyword **auto** is used to declare variables of automatic storage explicitly but it is **rarely** used
  - Variables with automatic storage duration
    - Are created when the block in which they are defined is entered
    - They exist while the block is active
    - They are destroyed when the block is exited (and their value is lost)

Storage class and duration

  - Automatic variables are defined and considered local to the block in which they are defined
  - They have to be **explicitly initialized**, as they do not have a predefined default values after the definition
  - If the block is re-entered, the system once again allocates the memory for the variable, but the previous values remain unknown

- Notice that only variables (not functions) can have automatic storage duration

Examples

Automatic objects

The main program

```
int main (...) {
  auto int i;
  float f;
  ...
}
```

auto int i; ←→ int i;

A function

```
<type> function (...) {
  int i;
  float f;
  {
    int j, k;
    char c = ‘x’;
    ...
  }
  ...
}
```

Variables must be initialized to have a known value

A block of instructions

Storage class and duration

however it is just a suggestion, if it not possible it does not raise an error

- The storage class **register** means that variables should be stored in high-speed memory registers if this is physically and semantically possible
- Its use is an attempt to improve execution speed
  - When speed is a concern the programmer may choose a few variables that are frequently used and define them as belonging to the storage class register
  - As compiler have become exceedingly efficient to optimize the programmer's code, the keyword register is **archaic** and should rarely be used

variables that are critical in terms of speed, however there are limited registers, and therefore they may not be allocated

Examples

Register objects

```
int main (...) {
  register int i;
  ...
}
```

May be use as variable to iterate, ergo, please "compiler" maintains it into a register

```
<type> function (...) {
  register int i;
  ...
}
```

Ditto ... but maybe there are no more registers available ... ergo "register" may be ignored

Storage class and duration

- The meaning of **extern** differs from variables and functions
- For variables
  - Extern is the default storage class for **global** variables
  - Global variables
    - Are created by placing variable definitions outside any function definition
    - Are the most common method to transmit information across blocks and functions
    - Retain their values throughout the entire execution of the program

Storage class and duration

- For global variables the key extern
  - Can be omitted
    - When the program spans a unique *.c file, and variables are defined at the beginning of the file
  - Must be inserted
    - With one single source file, when the variable is used before the original definition within that file
    - With several source files, when the variable definition in included in another file
- For extern variables the initialization is automatically done at compilation time, i.e., when memory is allocated for the variables

there might be problems if, the global variables are defined twice, an external variable is not found

Storage class and duration

- In general, it is a good programming practice to avoid using external variables
  - Any function in the program can access and alter an extern variable
  - Extern variables weaken the concept of data abstraction, independent module, and black box
    - The black box concept is essential to the development of a modular program with modules
  - They make debugging more difficult
- This is not to say that external variables should never be used
  - There are occasions in which external variable significantly simplifies an implementation

Examples

Extern variables

```
int i, j;
int main (...) { ... }
void f1 (...) { ... }
void f2 (...) { ... }
```

Variables i and j can be used by the main, f1, and f2 as well

```
int main (...) { ... }
int i, j;
void f1 (...) { ... }
void f2 (...) { ... }
```

Variables i and j can be used only by f1 and f2

Variables defined in file1 can be used in files2 as well

file1.c

```
int i, j;

void f1 (...) { ... }
```

file2.c

```
extern int i, j;

void f2 (...) { ... }
```

Storage class and duration

- For functions
  - The use of extern for functions has a slightly different meaning
  - All function are of storage class extern by default
    - The use of the keyword extern can be implicitly or explicitly
    - Functions
      - Must be defined only once
      - Can be declared (without or with the extern keyword) in any file in which we want to use them

Examples

Extern functions

```
extern float f1 (int);
float f2 (int);

float f1 (...) { ... }
float f2 (...) { ... }
```

Both f1 and f2 are "global" by default and can be used everywhere after the prototype

Often, "extern" indicates that the functions is defined somewhere else

file1.c

```
float f1 (int);
extern float f2 (int);

float f1 (...) { ... }
```

file2.c

```
extern float f1 (int);
float f2 (int);

float f2 (...) { ... }
```

Each function is defined just once but can be used in both files

Storage class and duration

- **Static** objects have two important uses
  - Static local variables **retain** their previous value when a block is re-entered
    - This is in contrast to ordinary automatic variables, which lose their value upon exit and must be re-initialized
  - **Static extern** functions are private, i.e., restricted in scope
    - They are unavailable in other files or earlier in the same file
      - They cannot be defined as extern in other files
    - Static extern objects are **locally global**

Examples

Static objects

```
void f (...) {
  static int i = 0;
  ...
}
```

Variable i is local to f but it maintains its value.
It is initialized, during the first call, to 0 and its value is retained in all subsequent calls

```
void f (...) {
  static int i = 0;
  if ...
  print ("Number of calls %d, i++);
  ...
  f(...);
  return;
}
```

For example, I can use a static variable to understand how many times I have called a specific (recursive or non-recursive) function

Examples

Static objects

file.c

```
static float var;

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

Function f has a scope (visibility) restricted to file.c.
Somehow, f is a local to file.c but in file.c it is also global (global to the family of functions defined in file.c after that line).
The same considerations hold for the variable.

```
void global_f1 (...) { }
void global_f2 (...) { }
void global_f3 (...) { }

static void local_f1 (...) { }
static void local_f2 (...) { }
static void local_f3 (...) { }
```

For example, I can use a static function when I do not want to make them exportable, visible from outside.
These functions are "not released", "outside" the library or for "internal usage only"

global_f1 should be defined in one file only

local_f1 should be defined locally in each file