NWEN 241
Systems Programming

Sue Chard

suechard@ecs.vuw.ac.nz

Content

- New eBook in the library.

  S. Malik C++ Programming Program Design including data structures 8th Edition 2018

- Review C Program structure

- The part of storage to allocate for a variable – Storage class specifiers

Program structure

```
# include <stdio.h>

void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA;
  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
  printf("maxA %.1f    A[4]%.1f\n",*maxA,A[4]);
}
float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);

  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);
  return theMax;
};
```

This Code will not compile, although the find max function is defined in the code it is not defined before it is called

A

```
# include <stdio.h>

float *findMax(float A[], int N);

void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA;
  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
 printf("maxA %.1f    A[4]%.1f\n",*maxA,A[4]);
  }

float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);

  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);
  return theMax;
};
```

B

```
# include <stdio.h>

float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);

  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);
  return theMax;
};

void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA;
  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
 printf("maxA %.1f    A[4]%.1f\n",*maxA,A[4]);
  }
```

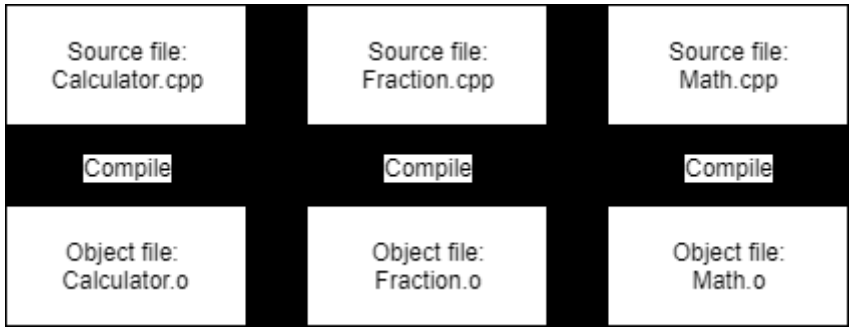
Both of these will compile and run

A . Uses a declaration of a function prototyte declared before the main function

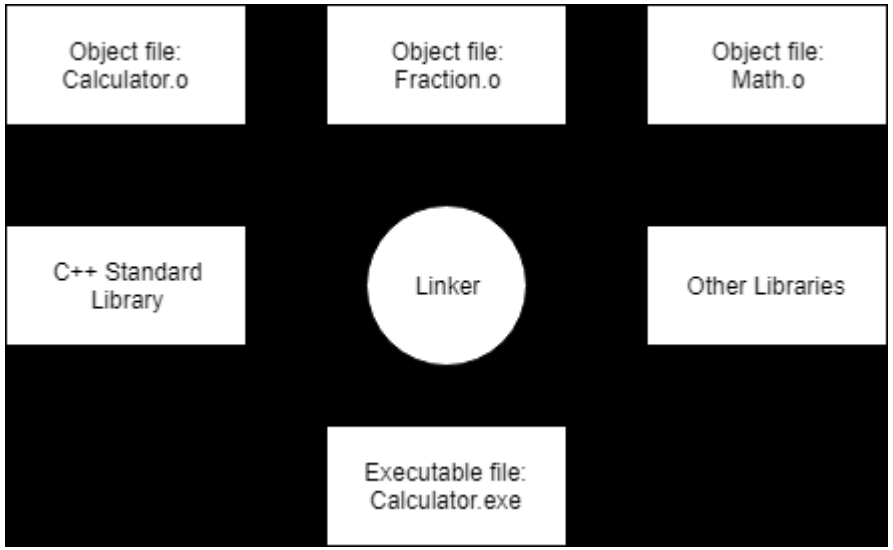
B. Declares and defines the function before the main function

- C Header files used by the Include statement
  - Include is a preprocessor directive
  - Instructs the compiler to read the source code from another file, this source code is then included in the code being compiled
  - The include statement is not limited to library header files such as <stdio.h> it can also include any file you specify

The compiler creates object files



The linker creates the executable file from the object files



This code could be in 3 separate files

```
# include <stdio.h>

float *findMax(float A[], int N);


void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA;
  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
  printf("maxA %.1f    A[4]%.1f\n",*maxA,A[4]);
}

float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);

  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);
  return theMax;
};
```

This would be in a header file e.g. “mylib” and included in the source code file similar to stdio

The main function would then be in a file by itself with two include statements

This function would be in a separate source code file defining mylib.
This would then be compiled into a separate object file
This is sometimes called a compilation unit
The linker would join the object files to create an executable

Variable Storage Class specifiers

C storage classes are:

- **`Auto (is the default)`**
- **`static`**
- **`register`**
- **`extern`**

Storage class of a variable determines its:

- ***Scope*** attribute – where is a variable visible
- ***Lifetime*** attribute – how long does a variable exists

Scope and Lifetime

- Lifetime/storage attributes can be:
  - **`static`** variables are allocated memory when program starts;
  - **`auto`** – automatic variables are allocated memory when execution enters the block that contains it;
  - **`register`** – reside in CPU's high speed memory
- Scope attributes can be:
  - **`local`** – **`v`** is only visible inside the current, innermost scope, independent of storage/lifetime attribute; e.g. there are **`local static`** variables in C
  - **`global`** – **`v`** is visible in the whole compilation unit, from the line of declaration to the end of file
  - **`extern`**al – **`v`** is visible in all compilation units; **`static`**

`auto` Storage Class

- **`auto`** is the default storage class for a variable defined inside a function body or a statement block
- **`auto`** prefix is optional; i.e. any locally declared variable is automatically **`auto`**, unless specifically defined to be static

Example:

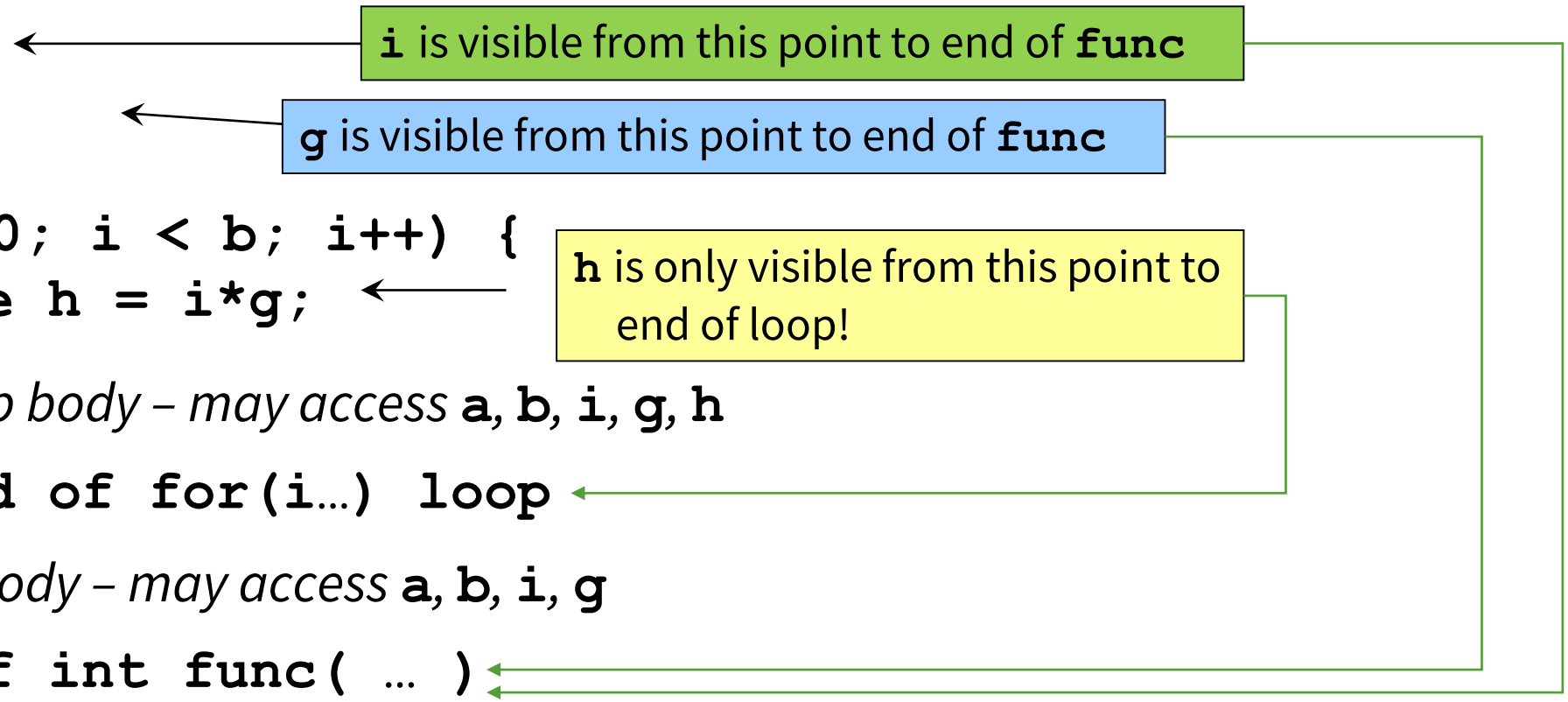
```
{
   auto double x; /* Same as: double x */
   int num;  /* Same as: auto int num; */
    . . .
}
```

`auto` Storage Class

- *Automatic variables* may *only* be declared *within* functions and compound statements {*blocks*}
  - Storage *allocated* when function or block is entered
  - Storage is *released* when function returns or block exits
- Parameters and result are similar to automatic variables
  - Storage is *allocated* and *initialized* by *caller* of function
  - Storage is *released* after function *returns* to caller.
- Variables declared within a function or compound statement are visible *only* from the point of declaration to the end of that function or compound statement.

`auto` Storage Class example

```
int func (float a, int b) {
  int i;
  double g;

  for (i = 0; i < b; i++) {
      double h = i*g;

      // loop body – may access a, b, i, g, h
  }   // end of for(i…) loop

   // func body – may access a, b, i, g
}//  end of int func( … )
```



`auto` Storage Class example

```
int func (float a, int b) {
  int i;
  double g;

  for (i = 0; i < b; i++) {
       double h = i*g;

       // loop body – may access a, b, i, g, h
  }    // end of for(i…)loop

   // func body – may access a, b, i, g
}//  end of int func( … )
```

Storage for **i** created.

Storage for **g** created

Storage for **h** created.

Storage for **h** released.

Storage for **g** released.

Storage for **i** released.

`auto` Storage Class initialization

- If an **`auto`** variable is defined but not initialized:
  - Variable has an unknown value when control enters its containing block
- If an **`auto`** variable is defined and initialized at the same time:
  - Variable is re-initialized **each** time control enters its containing block
- An **`auto`** variable's scope is limited to its containing block (i.e., it is **`local`** to the block)

`static` Storage Class

- Storage for a **`static`** variable:
  - Is allocated when execution begins
  - Exists for as long as the program is running
- A **`static`** variable may be defined either inside or outside a function's body.
- The **`static`** prefix must be included

  Example:

```
static double seed;
```

`static` Storage Class initialization

- If a **`static`** variable is defined but not initialized:
  - Is set to zero (0) once, when storage is allocated
- If a **`static`** variable is simultaneously defined and initialized:
  - Is initialized once, when storage is allocated
- A **`static`** variable defined inside a function body is visible only in its containing block
- A **`static`** variable defined outside a function body is visible to all blocks which follow it in the current compilation units
- If you wish it to be visible in other compilation units, it must be declared **`extern`**

`static` Storage Class example

```
#include <stdio.h>

void strange( int x )
{ // strange function
    static int y;  /* Persistent */
    if ( x == 0 )
        printf( "%d\n", y );
    else if ( x == 1 )
        y = 100;
    else if ( x == 2 )
        y++;
} //end of strange function

int main (void)
{ // main
    strange(1);  /* Set y in strange to 100 */
    strange(0);  /* Will display 100        */
    strange(2);  /* Increment y in strange  */
    strange(0);  /* Will display 101        */
    return 0;
} // end main
```

Program output

```
100
101
```

`register` Storage Class

- The fastest storage resides within the CPU itself in high-speed memory cells called *registers*
- The programmer can request the compiler to use a CPU register for storage

  Example:

  ```
  register int k;
  ```

- The compiler can ignore the request, in which case the storage class defaults to **`auto`**
- Some machines, e.g. stack architectures, have no user visible register

`extern` Storage Class (single source file)

- **`extern`** is the default storage class for a variable defined outside a function's body
- Storage for an **`extern`** variable:
  - Is allocated when execution begins
  - Exists for as long as the program is running
- If an **`extern`** variable is defined but not initialized:
  - Set to zero (0) once, when storage is allocated
- If an **`extern`** variable is defined and initialized:
  - Initialized once, when storage is allocated
- An **`extern`** variable is visible in all functions that follow its definition (i.e., it is **`global`**)

`extern` Storage Class example

```
#include <stdio.h>

float x = 1.5; /* Definition - extern class - global */

void show (void)
{
   printf("%f\n", x); /* Access global x */
}

int main (void)
{
   printf("%f\n", x); /* Access global x */

   show();

   return 0;
}
```

Storage Classes in Multiple Files

- Functions stored in a single source file can be divided into separate source files.

- Variables defined in one source file can be accessed from other source files via the **`extern`** storage class.

- An **`extern`** variable can be defined in **one file** only. However, it may be declared from other files.

Storage Classes in Multiple Files

- An **`extern`** variable is defined exactly once in a file by placing it outside all blocks.
- If an **`extern`** variable is not initialized at definition time
  → **`extern`** prefix must be omitted
- If an **`extern`** variable is initialized at definition time
  → **`extern`** prefix is optional
- An **`extern`** variable is declared in another file by using the **`extern`** prefix.

  Example:

```
extern int k;
```

Declare global variables:

file3.h

```
extern int global_variable;  /* Declaration of the variable */
```

file1.c

```
#include "file3.h"  /* Declaration made available here */
#include "prog1.h"  /* Function declarations */

/* Variable defined here */
int global_variable = 37;    /* Definition checked against declaration */

int increment(void) { return global_variable++; }
```

File2.c

```
#include "file3.h"
#include "prog1.h"
#include <stdio.h>

void use_it(void)
{
   printf("Global variable: %d\n", global_variable++);
}
```

prog1.h

```
extern void use_it(void);
extern int increment(void);
```

prog1.c

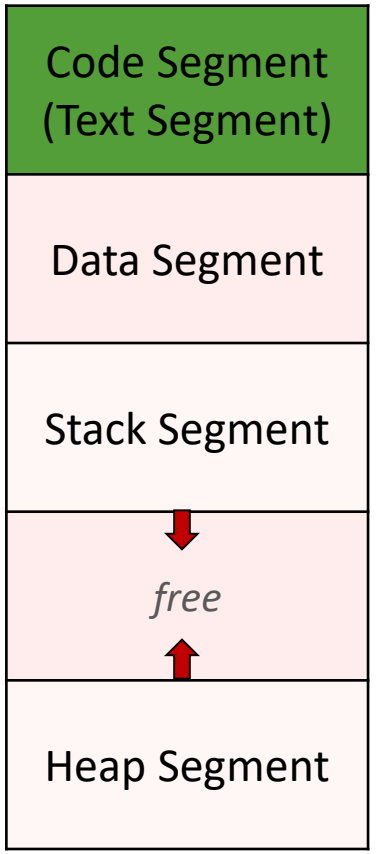
```
#include "file3.h"
#include "prog1.h"
#include <stdio.h>

int main(void)
{
   use_it();
   global_variable += 19;
   use_it();
   printf("Increment: %d\n", increment());
   return 0;
}
```

prog1 uses prog1.c, file1.c, file2.c, file3.h and prog1.h

Memory Layout of a Program

Memory space for program code includes space for machine language code and data

- **Text / Code Segment**
  - Contains program's machine code
- Data spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Stack Segment** – For temporary data, e.g. local variables in a function; expands / shrinks as program runs
  - **Heap Segment** – For dynamically allocated memory; expands / shrinks as program runs



Memory Storage Layout

Contains the program's machine code	Code Segment (Text Segment)
Contains static data (e.g., **`static`** class, **`extern`** globals)	Data Segment
Contains temporary data (e.g., **`auto`** class)	Stack Segment
Unallocated memory that the stack and heap can use	*free*
Contains dynamically allocated data – later…	Heap Segment