

Managing the Implementation of C Projects

1

Goals for Today's Class

- Compiling multiple files (modules)
- Scope and extent of variables in C (global, static)
- Using the preprocessor

2

Multiple Files

- Why multiple files?
 - Modularity!
- Execution of a program begins in the **main** function. The **main** function can call **other functions**
 - Functions defined in the same file
 - Function defined in other files or libraries
- A module is a collection of related functions
- Review:
 - Function Prototypes
 - Header Files

3

Function Prototypes (Declarations)

- Informs compiler about a function's return type, name, and parameters
 - Used to check whether function is called correctly
- Should be placed before `main()` or before first function is defined

4

Example of Function Prototype

```
int doNothing(int y);
```

Function
Prototype

```
int main(void) {  
    printf("%d", doNothing(10));  
}
```

```
int doNothing(int y) {  
    return y;  
}
```

Function
Definition

5

Header Files

- Typically contain:
 - Function prototypes for the module
 - Constants (#define)
 - Structures and external variables
- Not mandatory but useful for information hiding
 - Users of module should only need to examine header file
 - Avoids code duplication, collects declarations together, makes changes easy.

6

Example of a Header File

```
#ifndef THIS_H
#define THIS_H

typedef struct{
    const char *name;
    int num;
} data;

void start(void);
void report(const char*,int );

#endif /* THIS_H */
```

7

Using Conditionals in Header Files

- Example:

```
#ifndef SOME_H
#define SOME_H

/* Insert header file info */

#endif
```

- Prevents multiple inclusions of header files, which causes compilation problems

8

Things to Know

- A C program can have only one main module (or file) with its main function
- Reference function prototypes for sub-modules using:

```
#include "subModule.h"
```

- `#include` is a preprocessor command to import text from another file.

9

Example

```
#ifndef CALC_H calc.h  
#define CALC_H  
  
int square(int x);  
  
#endif /* CALC_H */
```

```
#include "calc.h" calc.c  
  
int square(int x) {  
    return x*x;  
}
```

```
#include <stdio.h> driver.c  
#include "calc.h"  
  
int main() {  
    int x = 5;  
    printf("\nSquare of %d is %d\n", x, square(x));  
    return 0;  
}
```

10

More on #include

- `#include <some.h>`

searches the system include path (i.e., the directory or directories where `stdio.h`, `string.h`, etc. are located) for `some.h`

- `#include "some.h"`

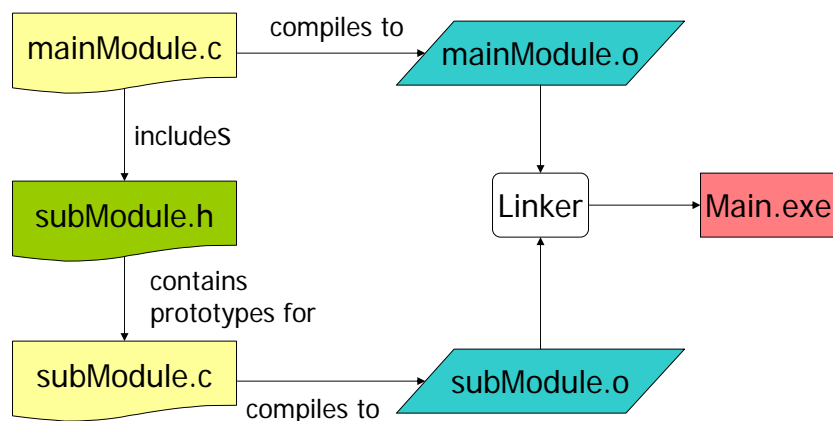
searches current directory

- `-ldir` (in the compile command)

appends `dir` to list of directories searched for header files and libraries for both `" "` and `< >`;

11

Files (.h,.c,.o,.exe)



12

Incremental Compilation

- Run

```
gcc -c
```

on sub-modules to generate object files

- Run

```
gcc mainModule.o subModule.o
```

to build final executable

13

Advantages of using Modules

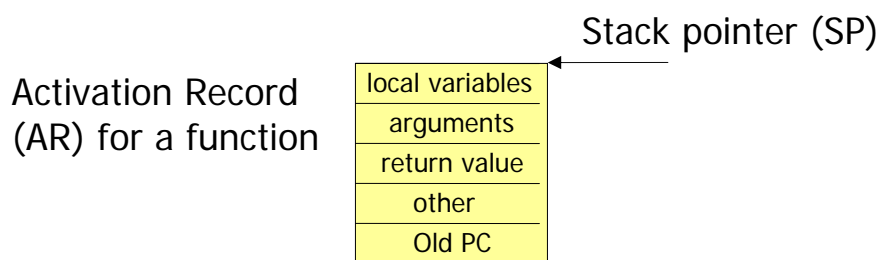
- Modules can be written and tested separately
- Large projects can be developed in parallel
- Reduces length of program, making it more readable
- Promotes the concept of abstraction

14

Scope and Extent

15

Execution Model for C



PC = Program Counter; points to execution code

16

Scope and Extent

- Scope
 - the range of visibility of a variable
- Extent
 - the length of time a variable stays in memory
- **Note:** a variable can be out of scope and still be in memory

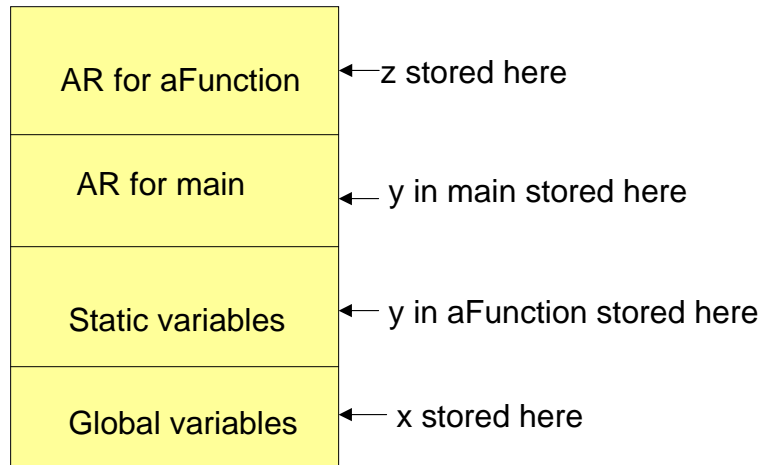
17

Example

```
int x = 0; ?  
  
void aFunction(void);  
  
int main(void) {  
    int y = 1;  
    ...  
} } Scope and extent of y in main  
  
void aFunction(void) {  
    static int y = 0; ?  
    int z = 5;  
    ...  
} } Scope and extent of z in  
aFunction
```

18

Memory Diagram



19

Global Variables in C

- Any variable defined outside a function
- Also called an external variable
- **Scope**: From point of definition to the end of the source file, apart from name clash
- **Extent**: From beginning to the end of the program
- To see a global in a different file:
`extern int x;`

... BUT ...

20

Declaration vs. Definition

- Declaration – stating that the name exists
- Definition – allocating memory to the variable
- Example:

```
int x;          /*Declaration and definition*/
extern int x; /*Declaration only*/
```

21

Example

```
#ifndef PRINT_H print.h
#define PRINT_H

extern int value;
void printValue();

#endif /* PRINT_H */
```

```
#include "print.h" print.c

int value = 10;

void printValue() {
    printf("%d", value);
}
```

```
#include <stdio.h> driver.c
#include "print.h"

int main() {
    printValue();
    return 0;
}
```

22

Global Variables vs. Modularity

- Avoid globals if possible!
- Globals violate modular independence
 - A global can be modified from any external file

23

Static Variables

- Declared in a function:
 - **Scope:** The function
 - **Extent:** From the first call to the function to the end of the program
 - **Example:** `static int x;`
- Static globals
 - Declared outside functions
 - **Scope:** **Only** in file where it is defined
 - **Extent:** Beginning to end of program

24

Example

```
int main(void) {
    int i;
    for(i = 0; i < 5; i++)
        printf("%d\n", Increment());
}

int Increment() {
    static int x = 0;
    x++;
    return x;
}
```

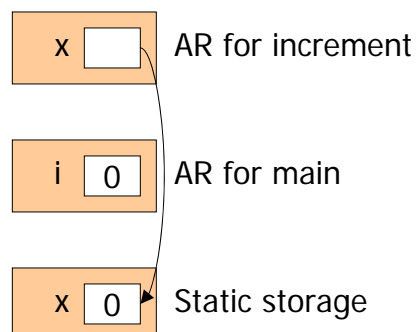
Output:

1
2
3
4
5

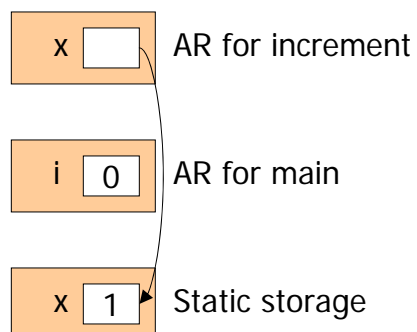
25

Memory Diagrams

First call to increment()



First call to x++



26

Memory Diagrams

After return from increment()

i 0 AR for main

x 1 Static storage

Static storage is still maintained while AR for increment is gone!

27

Using the PreProcessor

- Preprocessing

- #define
- #include
- Conditionals (#if, #else, #elif, #endif)

- Example

```
#define DEBUG 1
#if DEBUG
    initDebug();
#else
    initNormal();
#endif
```

28