Pointers, Scopes, Flow of Control, and Linkers in C

010.133

Digital Computer Concept and Practice

Spring 2013

Lecture 10

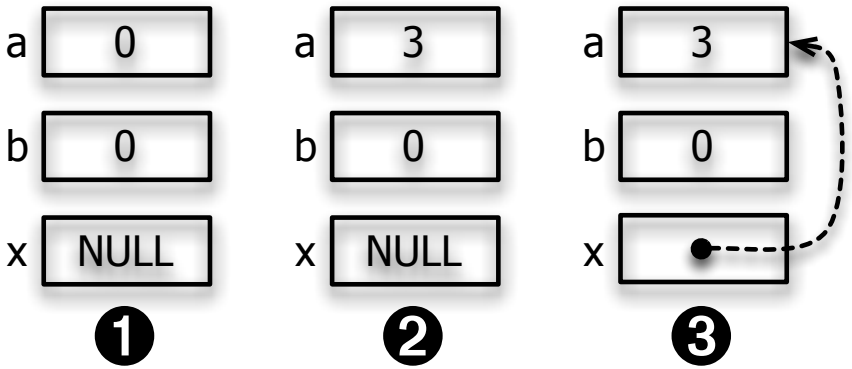
C Pointers

- A reference to a variable
  - The location (address) of the variable in the memory
- A pointer is a variable that stores a reference to another variable
  - It is said to point to the variable
  - Used in programs to access memory and manipulate addresses
- The basic syntax to declare a pointer is a type name followed by * and an identifier
  - For example,
    - `int *x;`
      - x is a pointer to an integer
      - x contains the address of an integer variable or value
- A null pointer has a special value (NULL) reserved for indicating that it does not refer to any valid object
  - For example,
    - `int *x = NULL;`

Reference Operator

- "address of" operator
- The reference (address) of a variable is obtained by preceding the variable with &

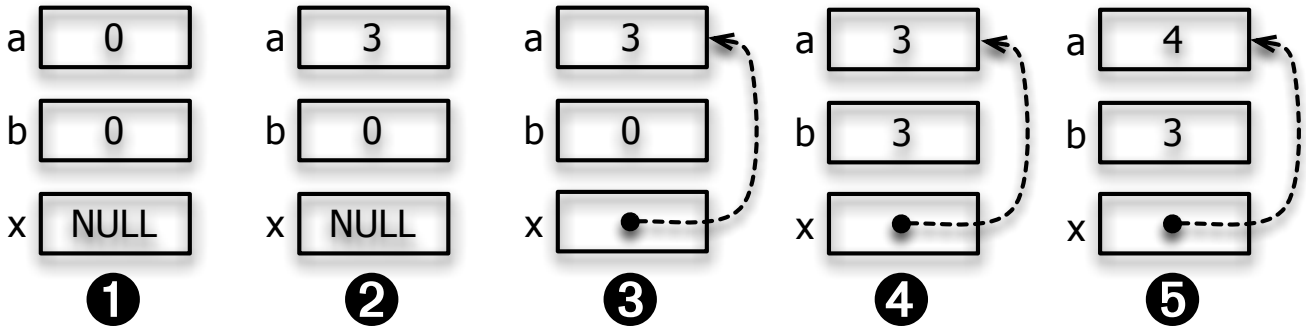
```
int *x;
int a, b;
                    // ❶
a = 3;              // ❷
x = &a;             // ❸
b = *x;             // ❹
*x = b + 1;         // ❺
```



Dereference Operator

- "value pointed by" operator
- Pointer dereferencing
  - Obtaining the value stored in the location which the pointer points to
- Preceding the pointer with *
- If a NULL pointer is dereferenced then a run-time error will occur and execution will stop likely with a "segmentation fault"
- The l-value of *x is the value of x and its r-value is the value stored in the location pointed to by x

```
int *x;
int a, b;
                    // ❶
a = 3;              // ❷
x = &a;             // ❸
b = *x;             // ❹
*x = b + 1;         // ❺
```

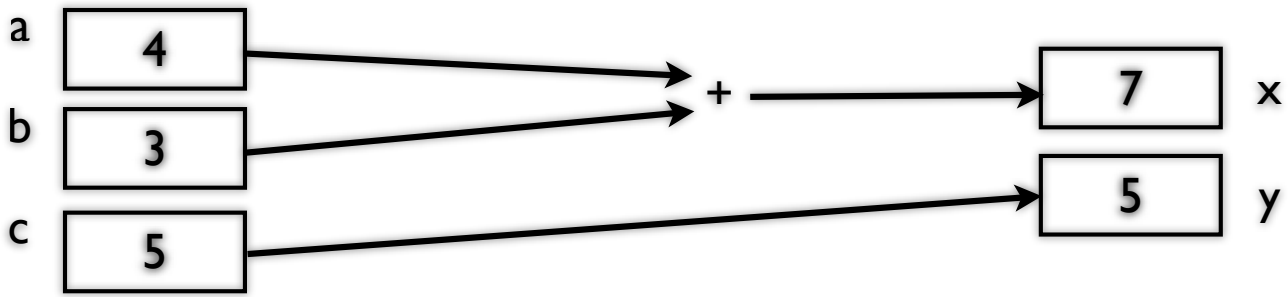


Void Type

- The type for the result of a function that does not return anything
- The type of a pointer which does not point to any particular type
  - For example, **`void *x;`**
  - The pointer contains an address and the compiler has no idea what type of object the pointer points to
- To indicate that a function does not take any parameters
  - **`int foo(void)`**
  - **`int foo() // equivalent to the above`**

Call by Value

- Arguments to functions are always passed by value
  - When a function is called, the value of each real argument is assigned to the corresponding formal parameter, and any changes in a formal parameter does not affect the value of the associated real argument
- The expression passed as an argument to a function is first evaluated, and its value is passed to the function




```
foo(a + b, c);
```

```
int foo(int x, int y)
```

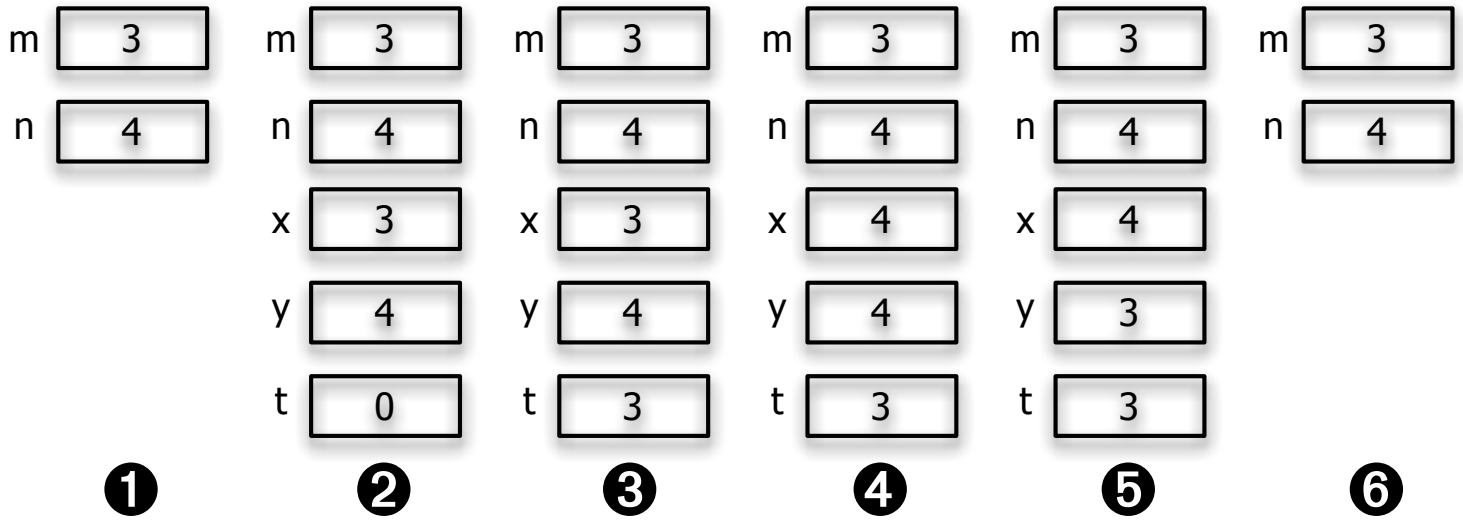
Call by Value (contd.)

```
#include <stdio.h>

int sum( int n )
{
    int s = 0;
    for ( ; n > 0; n--)
        s += n;
    return s;
}

int main( void )
{
    int n = 5;
    printf( "sum = %d\n", sum( n ) );
    printf( "n = %d\n", n );
    return 0;
}
```

Swap

```
#include <stdio.h>

void swap(int x, int y);

int main(void)
{
    int m = 3, n = 4;       // ❶

    swap(m, n);             // ❻
printf("m = %d, n = %d\n", m, n);

    return 0;
}
```

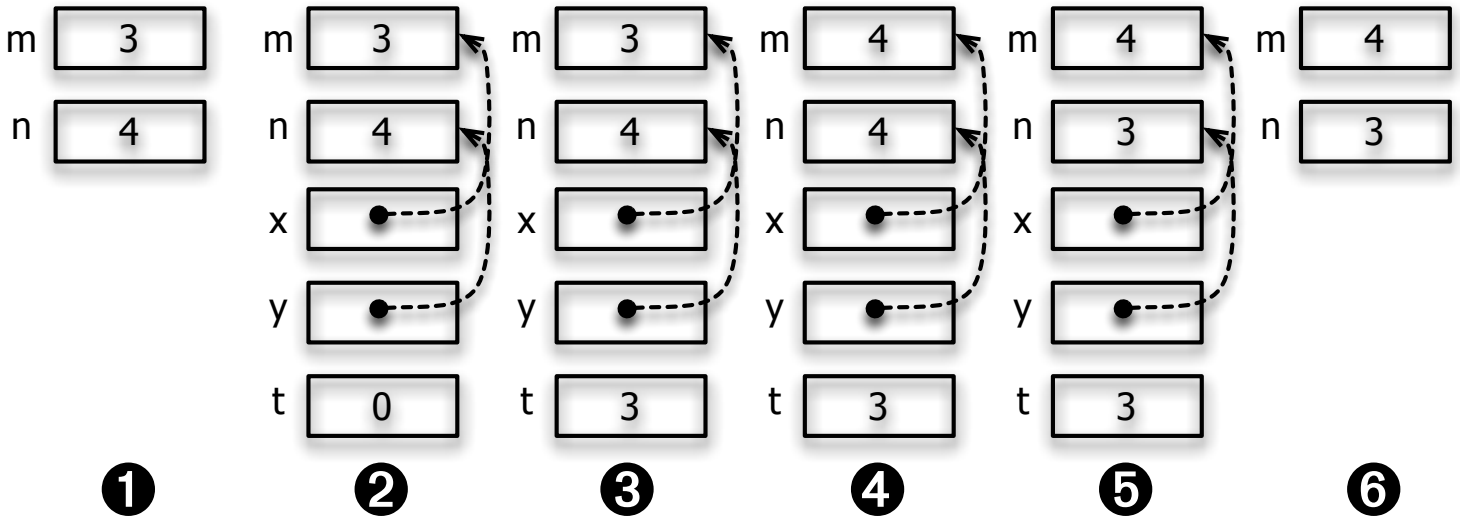
```
void swap(int x, int y)
{
    int t;                  // ❷

    t = x;                  // ❸
    x = y;                  // ❹
    y = t;                  // ❺

    return;
}
```



Swap (contd.)

```
#include <stdio.h>

void swap(int *x, int *y);

int main(void)
{
    int m = 3, n = 4;       // ❶

    swap(&m, &n);           // ❻
    printf("m = %d, n = %d\n", m, n);

    return 0;
}
```

```
void swap(int *x, int *y)
{
    int t;                  // ❷

    t = *x;                 // ❸
    *x = *y;                // ❹
    *y = t;                 // ❺

    return;
}
```



Local Variables

- Local variables
  - A variable declared inside a function
  - Function parameters
  - A variable declared inside a block
- Not visible to other functions
- Each local variable in a function or a block comes into existence when the function is called or the block is entered, and it disappears when the function or the block is exited
- The memory space for a local variable is allocated in the associated function's stack frame
  - Called automatic variables
  - Exists solely for the duration of the stack frame

Non-local Variables in C

- Memory spaces for non-local variables are allocated in the data section of the memory
  - `x: .word 12`
  - `y: .word`
- Exist for all time from the beginning of program execution to the end of the execution
- Accessible to all functions

```c
#include <stdio.h>

int area(int w, int h) {
    return w * h;
}

int main(void) {
    int width = 5, height;

    height = 10;
    printf("width: %d, height: %d, area: %d\n",
           width, height, area(width, height));

    return 0;
}
```

```c
#include <stdio.h>

int width = 5;
int height;

int area(void) {
    return width * height;
}

int main(void) {

    height = 10;
    printf("width: %d, height: %d, area: %d\n",
           width, height, area(width, height));

    return 0;
}
```

Declarations and Definitions

- Declarations tell us about the types of things, while definitions tell us about their values

Scope Rules

- A variable declared inside a function is a local variable
- Each local variable in a function comes into existence when the function is called, and it disappears when the function is exited
  - Temporary storage for the lifetime of the function or block
    - Its memory is allocated in the function's stack frame
  - Such a variable is called an ***automatic*** variable
  - Default and implicit

```
int foo(int n){
    int i;
    …
}
```

Scope Rules (contd.)

- Function parameters (e.g., variable n in foo) are also automatic variables
- If we put **`static`** in the declaration of a local variable, it remains in existence throughout the program
- Variable i in the example below is called a ***static local*** variable

```
int foo(int n)
{
    static int i;
    …
}
```

Scope Rules (contd.)

- A variable declared outside any function is an external variable
  - Permanent storage for the lifetime of the program
  - Implicit
- The scope of a variable or a function is the part of the program within which the variable or the function can be used
  - The scope of a local variable is the function in which the variable is declared
  - The scope of an external variable or a function is from the point of its declaration to the end of the file
    - An external variable can be accessed by all functions that follow its declaration

Scope Rules (contd.)

- Because of the scope rule, a function should be defined before it is used (called)
- However, C allows a function to be declared before it is defined
  - Forward declaration

```
#include <stdio.h>

int foo(int n);

int main(void)
{
    int x;
    …
    foo(x);
    …
}

int foo(int n)
{
    …
}
```

```
#include <stdio.h>

int foo(int n)
{
    …
}

int main(void)
{
    int x;
    …
    foo(x);
    …
}
```

Scope Rules (contd.)

- When we have a newly declared variable in the scope of another variable x with the same name, new x hides old x in the scope of new x

```
#include <stdio.h>
int i;              /* external variable */
int foo(int n);     /* forward declaration */

int main(void)
{
    int x;
    …
    x = i;          /* i refers to the external variable i */
    …
}

int foo(int n)      /* function definition */
{
    int i;
    …
    i = n;
    …
}
```

Static Scope and Block Structure

- A block is a grouping of declarations and statements
  - Enclosed with { and }
- A declaration D belongs to a block B if B is the most closely nested block containing D
  - D is located within B, but not within any block that is nested within B
- Static-scope rule
  - If declaration D of name x belongs to block B, then the scope of D is all of B, except for any blocks B' nested to any depth within B, in which x is redeclared
  - A name x is redeclared in B' if some other declaration D' of the same name x belongs to B'

Blocks in C

```
main() {
  int a = 1;
  int b = 1;
  {
    int b = 2;
    {
      int a = 3;
      printf("%d%d\n", a, b);
    }
    {
      int b = 4;
      printf("%d%d\n", a, b);
    }
    printf("%d%d\n", a, b);
  }
  printf("%d%d\n", a, b);
}
```

Sequential Flow of Control

- Statements in a program are executed one after another in the order they placed in the program

If Statement

- The if statement has the following form:
  - **if (** *expression* **)** statement1 **else** statement2
  - The else part is optional
- If *expression* is evaluated to true, then statement1 is executed
- If it is false, statement2 is executed
  - If it is false and there is no else part, the if statement does nothing

```
    CMP R5,#0   @ R5 contains the value of expression
    BEQ Else
    ...         @ code for statement1
    B   Done
Else:
    ...         @ code for statement2
Done:
```

If Statement (contd.)

- The following if statement sets abs to be the absolute value of x

```
if (x > 0) abs = x;
else abs = -x;
```

- The following example sets min to be the minimum of i and j

```
min = i;
if (j < min) min = j;
```

If Statement (contd.)

- statement1 or statement2 should be a single statement
- Compound statement
  - A series of declarations and statements surrounded by braces
    - { declarations statements }
  - A compound statement itself is a (single) statement
  - A block

```
if (x < 0) {
   temp = x;
   x = y;
   y = temp;
}
```

Cascaded if

- A cascaded if statement can be used to write a multi-way decision
- What does the program below do?

```
#include <stdio.h>
int main(void)
{
    char op;
    int a = 20, b = 4;

    printf("Select operator (+, -, *, /) : ");
    scanf("%c", &op);

    if (op == '+') printf("20 %c 4 = %d\n", op, a+b);
    else if (op == '-') printf("20 %c 4 = %d\n", op, a-b);
    else if (op == '*') printf("20 %c 4 = %d\n", op, a*b);
    else if (op == '/') printf("20 %c 4 = %d\n", op, a/b);
    else printf("Wrong operator!\n");

    return 0;
}
```

Switch Statement

- The switch statement also describes a multi-way decision that tests whether an expression matches one of several values
- The default case is optional
  - If the default case does not exist and none of the cases match, then no action will take place

```
switch(op) {
case '+': printf("20 %c 4 = %d\n", op, a+b);
          break;
case '-': printf("20 %c 4 = %d\n", op, a-b);
          break;
case '*': printf("20 %c 4 = %d\n", op, a*b);
          break;
case '/': printf("20 %c 4 = %d\n", op, a/b);
          break;
default: printf("Wrong operator!\n");
          break;
}
```

Switch Statement (contd.)

- The **`break`** statement causes an immediate exit from the switch statement
- If break is absent in a case, execution falls through to the next case

```
switch(op) {
case '+': printf("20 %c 4 = %d\n", op, a+b);
          break;
case '-': printf("20 %c 4 = %d\n", op, a-b);
          break;
case '*': printf("20 %c 4 = %d\n", op, a*b);
          break;
case '/': printf("20 %c 4 = %d\n", op, a/b);
          break;
default: printf("Wrong operator!\n");
          break;
}
```

For Statement

- The for statement is a convenient way to write a loop that has a counting variable
  - for (expr1; expr2; expr3) statement
    - expr1 is an initialization that is executed only once at the beginning of the loop
    - expr2 is a termination condition
    - expr3 is executed at the end of each loop iteration

```
    @ code for expr1
Loop:
    @ code for expr2
    @ R5 contains the value of expr2
    CMP R5,#0
    BEQ Done
    @ code for statement
    @ code for expr3
    B   Loop
Done:
```

For Statement (contd.)

- A for loop that computes the sum of integers from1 to 100

```
#include <stdio.h>
int main(void)
{
    int i, sum = 0;

    for (i=1; i<=100; i++)
        sum = sum + i;

    printf("1 + 2 + .... + 99 + 100 = %d\n", sum);
    return 0;
}
```

While Statement

- The while statement has the following form
  - while (expr) statement
    - expr is evaluated
    - If it is true (non-zero), statement is executed and expr is evaluated again
    - This process continues until expr becomes false (zero)

```
Loop:
    @ code for expr
    @ R5 contains the value of expr
    CMP R5,#0
    BEQ Done
    @ code for statement
    B   Loop
Done:
```

While Statement (contd.)

- Finding the minimum integer i such that 1+2+…+ i > 1000
- In general, for loops are better if we know the number of iterations in advance, and while loops are better otherwise

```
#include <stdio.h>
int main(void)
{
    int i = 0, sum = 0;

    while (sum <= 1000) {
        i = i + 1;
        sum = sum + i;
    }
    printf("minimum i which satisfies (1+2+…+i) > 1000 is %d\n", i);
            return 0;
}
```

Infinite Loops

- Sometimes one may need an infinite loop which can be written as follows:
  - while (1) statement

Do Statement

- The do statement is similar to the while statement, but expression is tested after the loop body (i.e., statement)
  - do statement while (expression);
- the loop body is executed at least once

Break Statement

- Causes an exit from the innermost enclosing loop or switch statement

```
while(1) {
    scanf("%lf", &x);
    if (x < 0.0)
        break;
    printf("%f\n", x*x);
}
```

Continue Statement

- Causes the current iteration of a loop to stop and the next iteration of the loop begin
- Printing all integers less than 100 that are not divisible by 3

```
#include <stdio.h>
int main(void)
{
    int i;

    for (i=0; i<100; i++)
    {
        if ( i % 3 == 0) continue;
        printf("%d ", i);
    }
    printf("\n");
    return 0;
}
```

Goto Statement

- goto label ;
  - Control is unconditionally transferred to a labeled statement
- In general, goto should be avoided
  - Using many gotos makes the program hard to read

```
…
goto error;
…
error: {
     printf("error\n" );
     exit(1);
}
```

exit(int status)

- In stdlib
- Never returns
- Terminates the program
  - 0 (EXIT_SUCCESS) stands for successful program completion
  - 1 (EXIT_FAILURE) stands for unsuccessful program completion

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
  int a = 0;

  if (a == 1)
    exit(1);
  else
    exit(0);

  a = 1;
}
```

getchar() and putchar()

- getchar()
  - Gets a value from the keybord and returns the value
- putchar(c)
  - The value of c is written to the standard output in the format of a character
- Both defined in stdio.h

```
#include <stdio.h>

void main()
{
    int c;
    while ( ( c = getchar() ) != EOF ) {
        putchar( c );
    }
}
```

Programming Example

- Write a program that gets a positive integer n and print the following shape in n lines

(when n = 4)

```
*
**
***
****
```

- The main idea is to have nested for loops, and to make each iteration of the outer for loop print each line of the shape

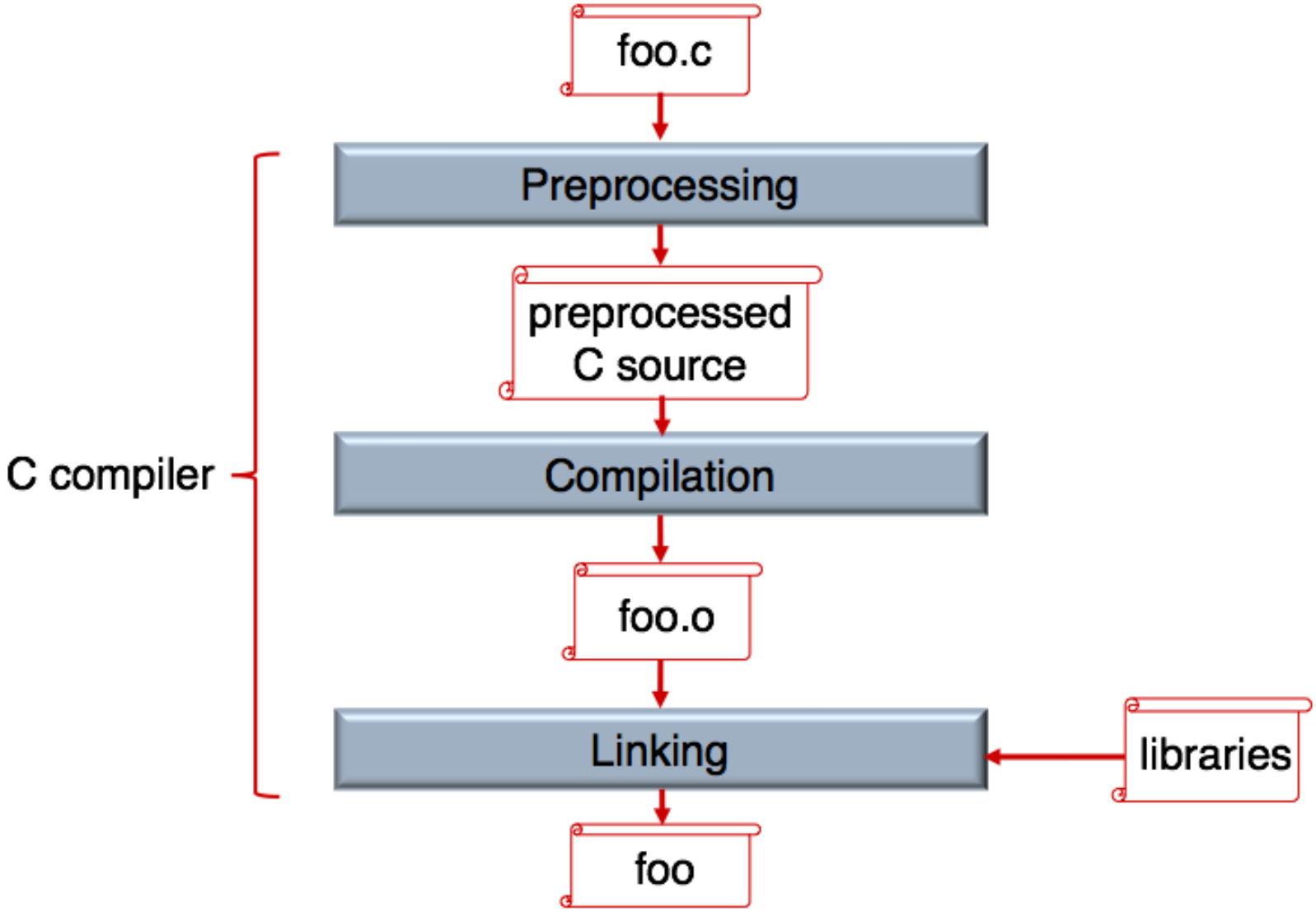
Programming Example (contd.)

```
#include <stdio.h>


int main(void)
{
   int n, i, j;

   scanf("%d", &n);
   for (i=1; i<=n; i++) {
      for (j=1; j<=i; j++)
         printf("*");
      printf("\n");
   }
   return 0;
}
```

Exercise

- Write a program to check the proper pairing of parentheses
  - Input: “()(()())”
  - Output: “Yes”
  - Input: “((())()))”
  - Output: “No”

Typical Compilation Phases (revisited)



Object Files

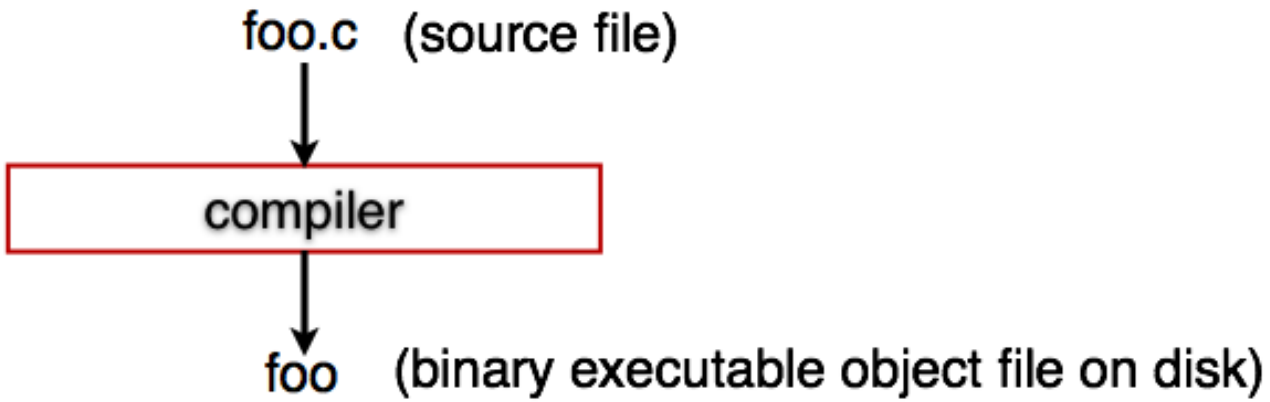
- Addresses of instructions and data are not fixed in an object file
- To determine the address later with a linker, it contains relocation information

Executable Files

- An executable file is generated by combining one or more object files and libraries
- The addresses of instructions and data are determined by linker

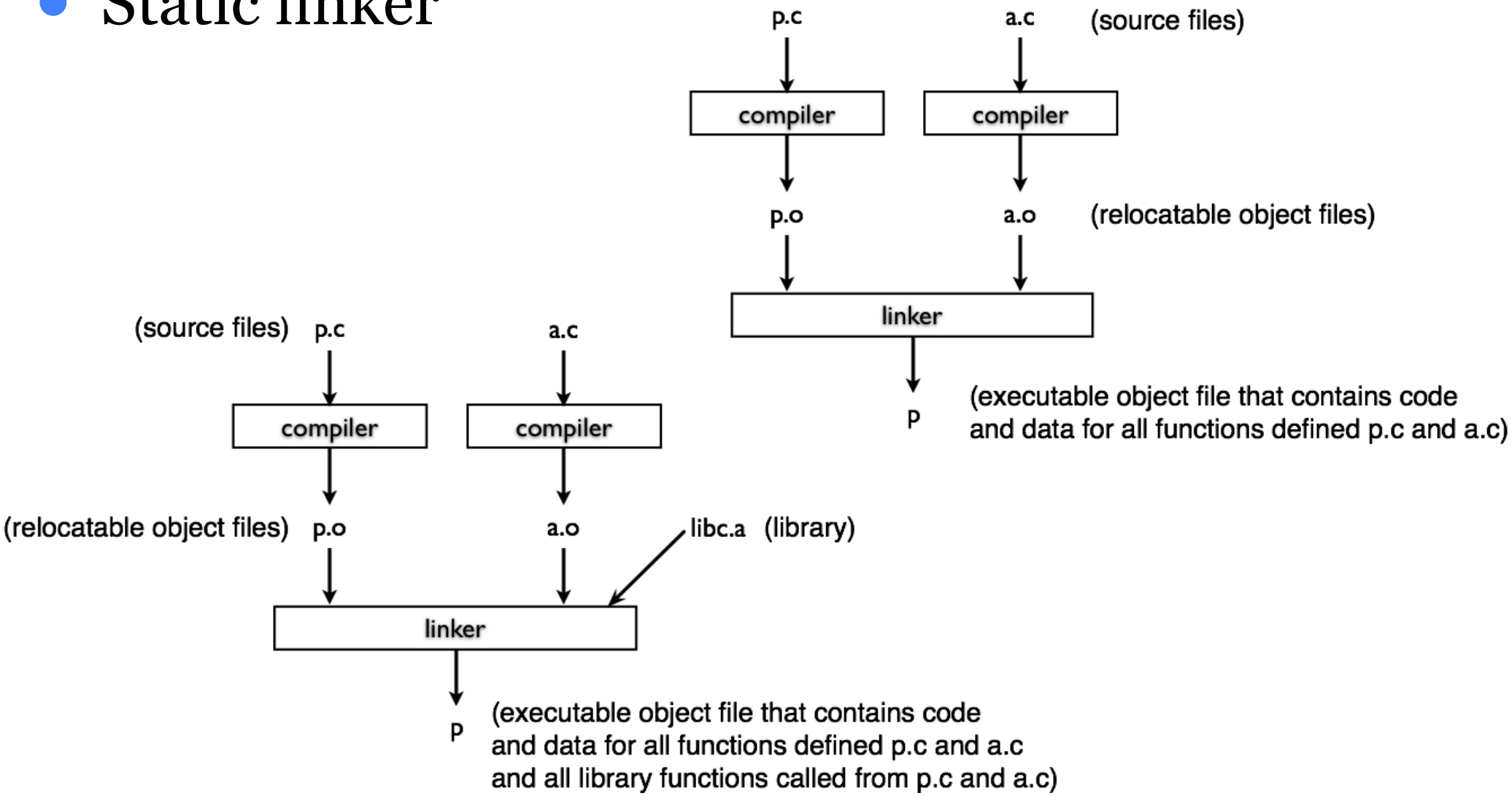
A Simplistic Program Translation Scheme

- Problems
  - Efficiency: small change requires complete recompilation
  - Modularity: hard to share common functions (e.g., printf)



A Better Scheme Using a Linker

- Static linker



A Better Scheme Using a Linker (contd.)

- Modularity
  - Program can be written as a collection of smaller source files, rather than one monolithic mass
  - Can build libraries of common functions (more on this later)
    - E.g., Math library, standard C library
- Efficiency
  - Time
    - Change one source file, compile, and then relink
    - No need to recompile other source files
  - Space
    - Libraries of common functions can be aggregated into a single file
    - Yet executable files and running memory images contain only code for the functions they actually use

Extern Storage Class

- Variables declared outside a function
  - Permanent storage for the lifetime of the program
- Implicit
- All functions have external storage class
- The keyword extern tells the compiler to look for the variable in this file or some other file
  - E.g., extern int a; in foo.c

Extern Storage Class (contd.)

```
/*****************/
/* main.c */
/*****************/
int a, b, c;
void foo(void);

int main(void)
{
    a = b = c = 1;
    foo();
    printf("%d %d %d\n", a, b, c);

    return 0;
}
```

```
/*****************/
/* foo.c */
/*****************/

void foo(void)
{
    extern int a;
    int b, c;
    a = b = c = 4;

    return;
}
```