HIGH-LEVEL PROGRAMMING I

Intro to C Programming (Part 2/3) by Prasanna Ghali

Outline

- What is C?
- C's Strengths
- Why not C?
- History of C
- Procedural Paradigm
- Organization of C Programs
- Writing and Calling Functions
- Creating and Running a C Program
- C Programs with Multiple Source Files
- Header Files: Interface vs. Implementation

What is C? (1/3)

- C is a high-level language
  - Provides constructs for structured programming
  - Supports functions and procedural paradigm
  - Supports user-definable derived data types
- C is a (relatively) low-level language
  - Supports full range of native machine types
  - Supports access to machine-level addresses
  - Provides operations that correspond closely to machine's built-in instructions

What is C? (2/3)

- C is a small language and is therefore easy to learn
  - Provides limited set of features compared to other languages
  - Uses library for commonly required features including input/output, file management, memory management, math, …
  - We'll use about 34 keywords

we've come across these keywords

```
signed, unsigned, char, short, int, long,
float, double, void, if, else, while, return
```

What is C? (3/3)

- C is a general purpose programming language
  - Can be used to program variety of different applications
  - Most widely used systems programming language for implementing operating systems, compilers, databases, …
  - De facto standard for programming embedded systems
- Provides excellent foundation for learning ALL other languages

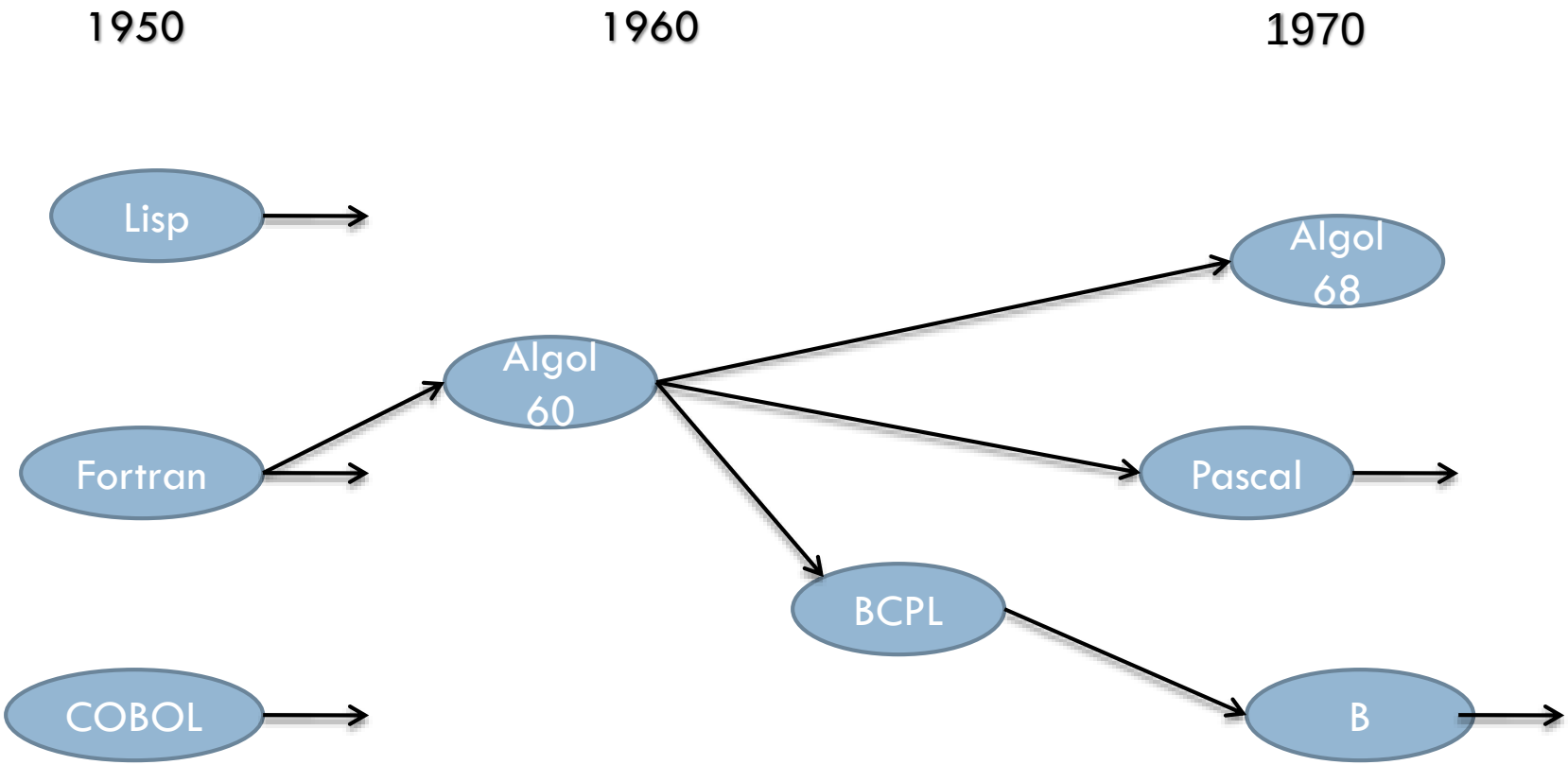
C's Strengths

- Simple
- Efficient
- Integration with Unix/Linux
- Portable
- Standard library
- Powerful
- Flexible

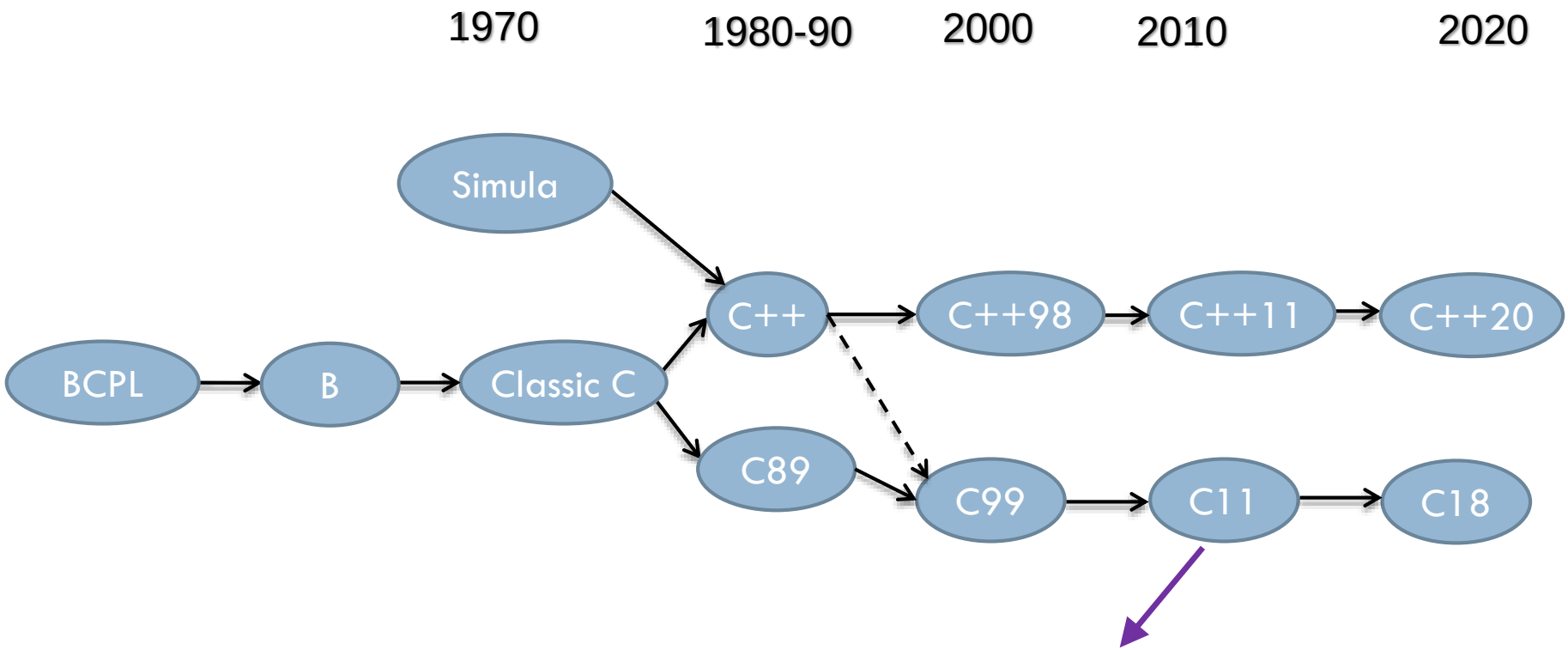
Why Not C?

- C programs can be error-prone
  - Permissiveness and flexibility may provide too wide latitude for some programmers
- C programs can be terse
  - Brevity may be difficult for some programmers
- Large scale C programs can be difficult to maintain
  - Language itself doesn't provide support for modular and object-oriented paradigms

History of C (1/2)



History of C (2/2)



This course will use C11

Consider Complex Problem (1/2)

- You have to send flowers to your grandmother (who lives in Japan) for her birthday
  - Plant flowers
  - Water flowers
  - Pick flowers
  - Fly to Japan with flowers

Consider Complex Problem (2/2)

- You have to send flowers to your grandmother (who lives in Japan) for her birthday
  - Plant flowers
    - Make trip to nursery to make purchases
    - Prepare soil in pot
    - Plant seeds in pot
    - …
  - Water flowers
  - Pick flowers
  - Fly to Japan with flowers

Another Complex Problem (1/2)

- You're asked to organize catering for a wedding
  - Make up guest list
  - Invite guests
  - Select appropriate menu
  - Book reception hall
  - …

Another Complex Problem (2/2)

- You're asked to organize catering for a wedding
  - Make up guest list
    - Get list from groom
    - Get list from bride
    - Check for conflicts
      - Check with bride about groom's list
      - Check with groom about bride's list
      - Check final list with groom's parents
      - Check final list with bride's parents
      - …
  - Invite guests
    - …
  - Select appropriate menu
  - Book reception hall
  - …

Procedural Programming Paradigm (1/2)

- Breaking down tasks into smaller subtasks is good plan of attack for solving complex programming problems too
  - Each "large" task is decomposed into smaller subtasks and so forth
  - Process is continued until subtask can be implemented by single ***algorithm***
- Synonyms for this strategy: ***top-down design, procedural abstraction, functional decomposition, divide-and-conquer, stepwise refinement***

Procedural Programming Paradigm (2/2)

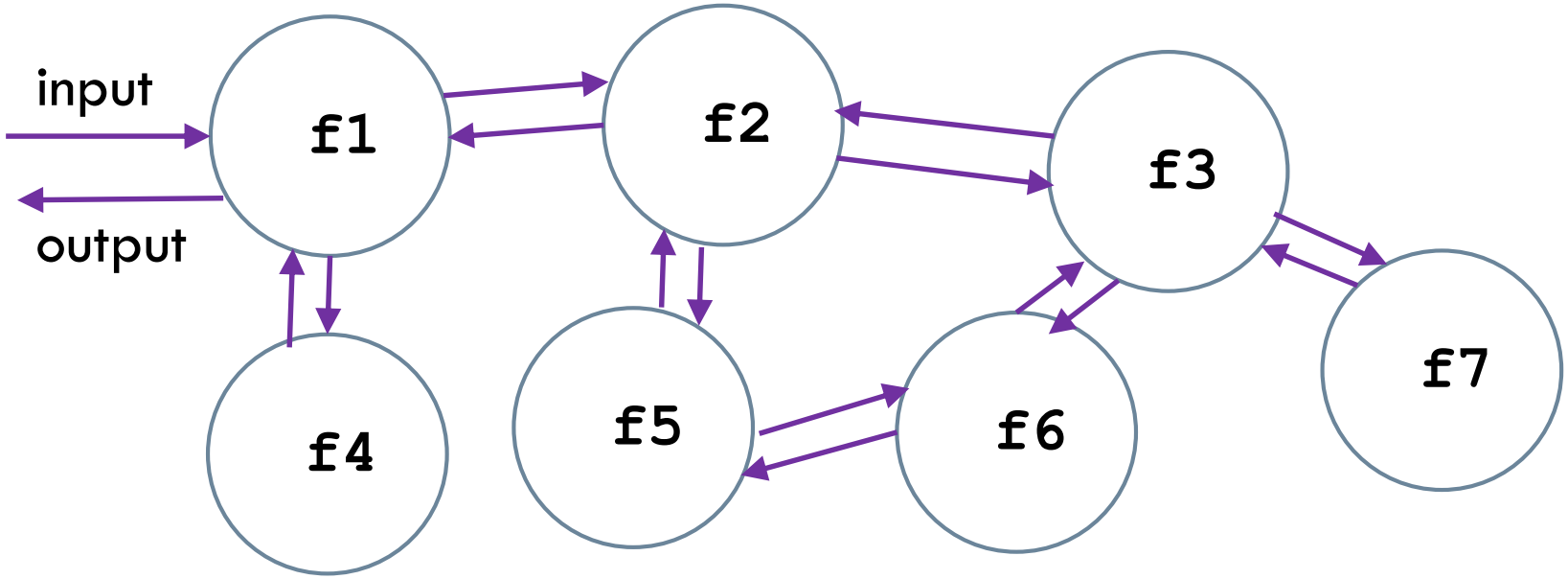
- In C/C++, algorithm is packaged into building block called ***function***
  - Other languages refer to function as ***procedure*** or ***subroutine*** or ***method***
- Program is organized into these smaller, independent units called ***functions***

Advantages of Functions

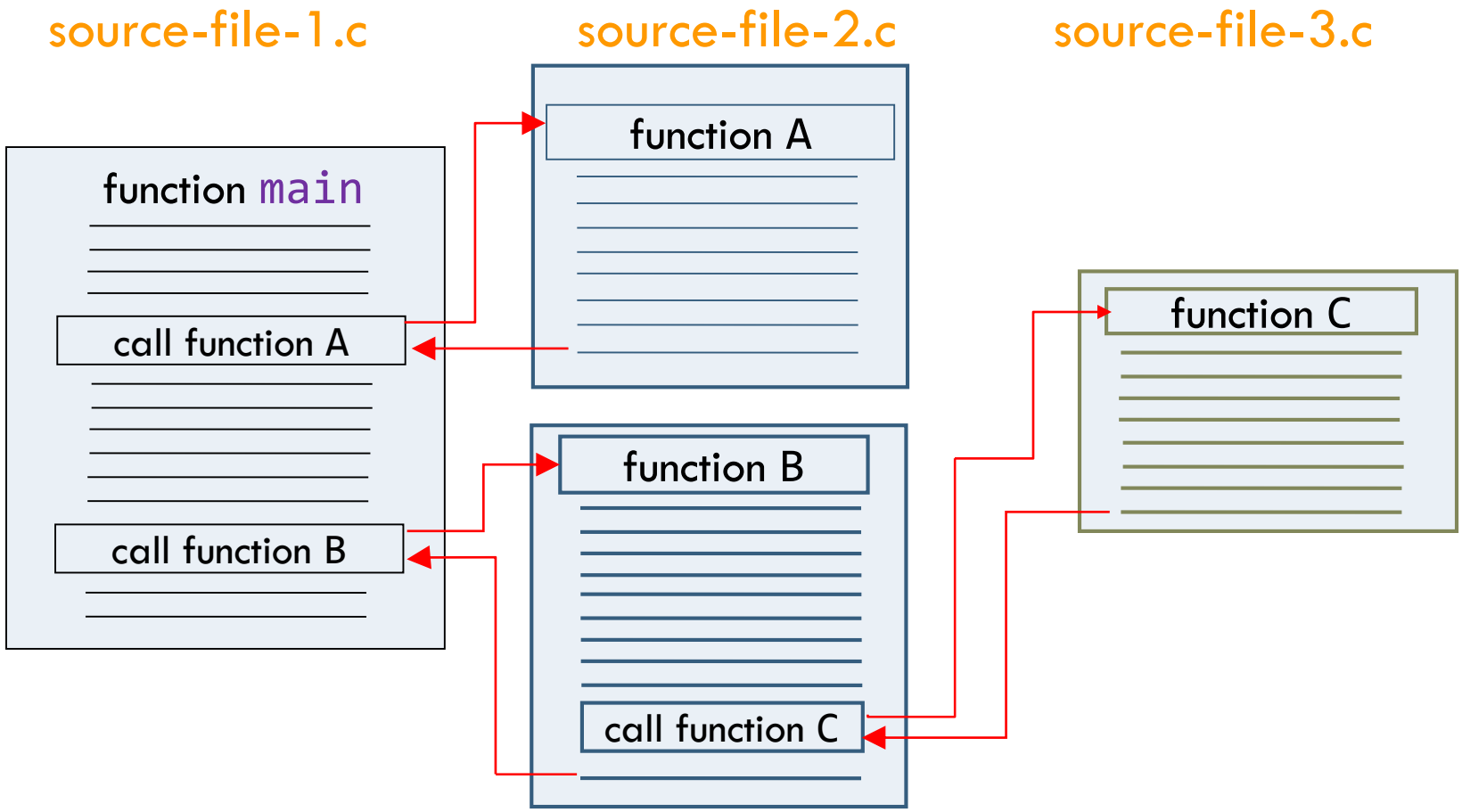
- Divide and conquer approach to complexity
  - Divide complicated whole into simpler and more manageable units
  - Standalone, independent functions are easier to understand, implement, maintain, test and debug
- Cost and pace of development
  - Different people can work on different functions simultaneously
- Building blocks for programs
  - Write function once and use it many times in same program or many other programs

Organization of C Programs (1/3)

- C was designed to use procedural paradigm to solve programming problems
  - Program is *synchronization* of different functions to solve problem



Organization of C Programs (2/3)



- Every C program must have *one and only one* function called `main` → not a C/C++ keyword!!!

Organization of C Programs (3/3)

- Related functions are organized into a source file
- Think of C program as one or more source files with each source file containing one or more related functions

```
// source-file-1.c
preprocessing directives

function prototypes

data declarations (global)

return-type
main(parameter declarations)
{
  data declarations (local)
  statements
}

other functions
```

...

```
// source-file-n.c
preprocessing directives

function prototypes

data declarations (global)

return-type function-name
(parameter declarations)
{
  data declarations (local)
  statements
}

other functions
```

How to ~~Write~~ Define a Function (1/3)

1) Every function *must* have a name
2) C has rules for specifying names

specifies *type* of output value

Comma-separated list of input values

{ and } are used to group statements that implement function

```
return-type function-name(parameter-list)
{
   statements
}
```

In C, each `statement` is *expression* followed by ;

How to ~~Write~~ Define a Function (2/3)

this variable in parameter list is called *formal parameter* or just *parameter* i.e., `num` is parameter of type `int`

1

```
int abs(int num)
{
  if (num < 0) {
    num = -num;
  }
  return num;
}
```

2

C statements are terminated by ;

3

Statement says "function will return an output value `num` of type `int`"

How to ~~Write~~ Define a Function (3/3)


```
int max(int x, int y)
{
  if (x > y) {
    return x;
  } else {
    return y;
  }
}
```

Simplest C Program: Does Nothing!!!

Every C program *must* have one and only function named `main`

1

Here, we're saying that `main` takes *no* input value(s)

2

`main` *must* return an output value of type `int`

3

`void` is keyword indicating no value!!!

4

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

{ and } are used to group statements that implement function

5

8

`return` is keyword used in a statement to terminate function or to return output value

6

7

Statement says "function will return an output value `0` of type `int`"

Every C statement must terminate with `;`

How to Call a Function

this *variable* in parameter list is called *parameter*

1

```
int abs(int num) {
  if (num < 0) {
    num = -num;
  }
  return num;
}

int main(void) {
  int x = -10;
  x = abs(x);
  return 0;
}
```

2

3

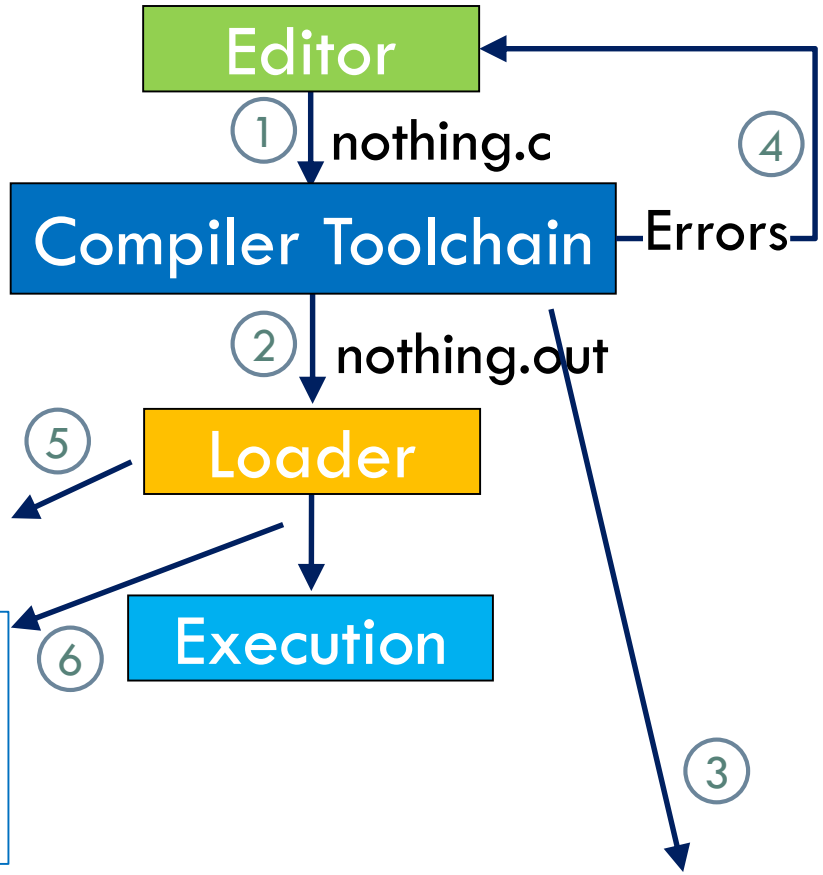
1) At run-time, client function `main` calls function `abs` using function call operator ( )
2) Argument `x` is evaluated to value `-10`
3) Result of evaluation is used to initialize parameter `num` to value `-10`
4) Function `abs` returns value `10` back to client before terminating
5) So, result of calling function `abs` is value `10` of type `int` – this value is then assigned to `x`

variable `x` is declared type `int` and initialized with value -10

Function `abs` is called by using its name followed by ( ) that enclose a value (`x` here)
Value `x` in function call operator is called *argument*

Creating and Running a C Program


```c
// this is file nothing.c
int main(void)
{
  return 0;
}
```



OS program that loads executable nothing.out into memory

Program nothing.out begins execution once loader has installed executable in memory

```
gcc -std=c11 -pedantic-errors -Wstrict-prototypes
-Wall -Wextra -Werror nothing.c -o nothing.out
```

gcc Options

- `-std=c11` Uses C11 standard
- `-pedantic-errors` Gives an error (not just a warning) if code is not following C11 standard
- `-Wstrict-prototypes` Disallows things allowed in old C standards – we want C code to be compatible with C++
- `-Wall` Warns about anything that compiler finds shady
- `-Wextra` Warns about more shady things than `-Wall`
- `-Werror` Converts warnings to errors so that code with warnings is never successfully compiled
- `-o nothing.out` Specifies name of output file; otherwise file name defaults to `a.out`

```
gcc -std=c11 -pedantic-errors -Wstrict-prototypes
-Wall -Wextra -Werror nothing.c -o nothing.out
```

Next Simplest C Program: Print a Greeting!!! (1/2)

1) Strange syntax that is not a part of C!!!
2) Any line that begins with # is directive to another program called *preprocessor*.
3) Think of preprocessor as editor that modifies C source file according to directives that begin with # character

1) C has no facilities for I/O.
2) Instead, it has set of libraries to support I/O, math, date & time, and many other functionalities.
3) Functionalities in each library are *declared* in a standard header file.
4) To use a particular library's functions, add preprocessor `#include` command

1

2

```
// this is file hello.c
#include <stdio.h>

int main(void)
{
  printf("Hello World\n");
  return 0;
}
```

Next Simplest C Program: Print a Greeting!!! (2/2)


```c
// this is file hello.c
#include <stdio.h>

int main(void)
{
  printf("Hello World\n");
  return 0;
}
```

1

1) File `stdio.h` is called *header file*
2) It contains *prototype* of function `printf`
3) Makes name `printf` and function's parameter list and return type known to compiler
4) C/C++ require all names used in source file to be *declared* before their first use
5) Preprocessor will replace 1st line with contents of header file
6) < and > delimiting header file name simply tell compiler where to find the file

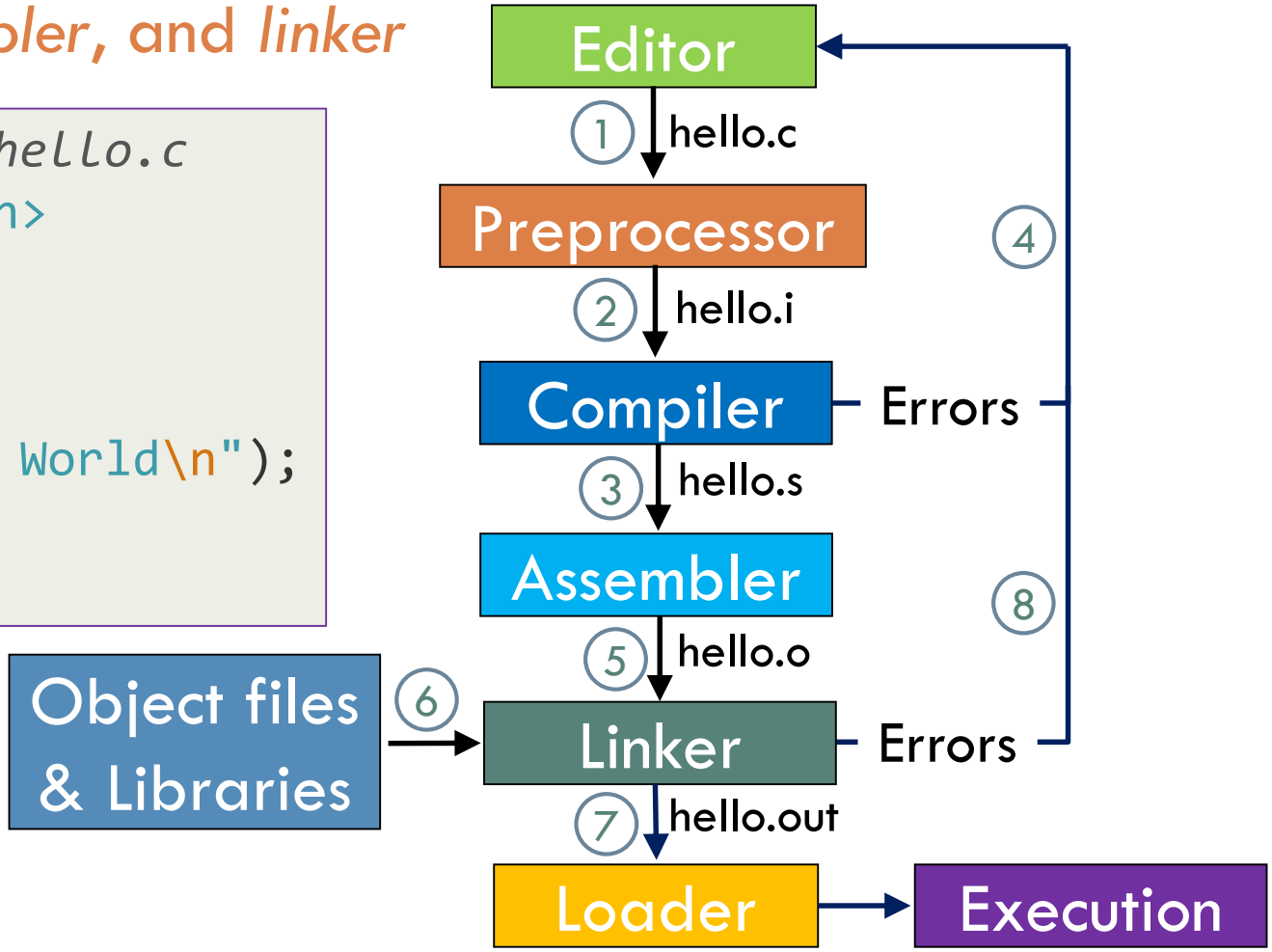
2

1) Function `printf` is part of standard C library
2) It is used to print information to standard output (the screen)
3) It is one of the most complex functions in the library
4) Argument `"Hello World\n"` represents a string (sequence of characters) with `\n` representing a newline

Compilation Phases (1/2)

Compiler toolchain consists of four phases: *preprocessor*, *compiler*, *assembler*, and *linker*

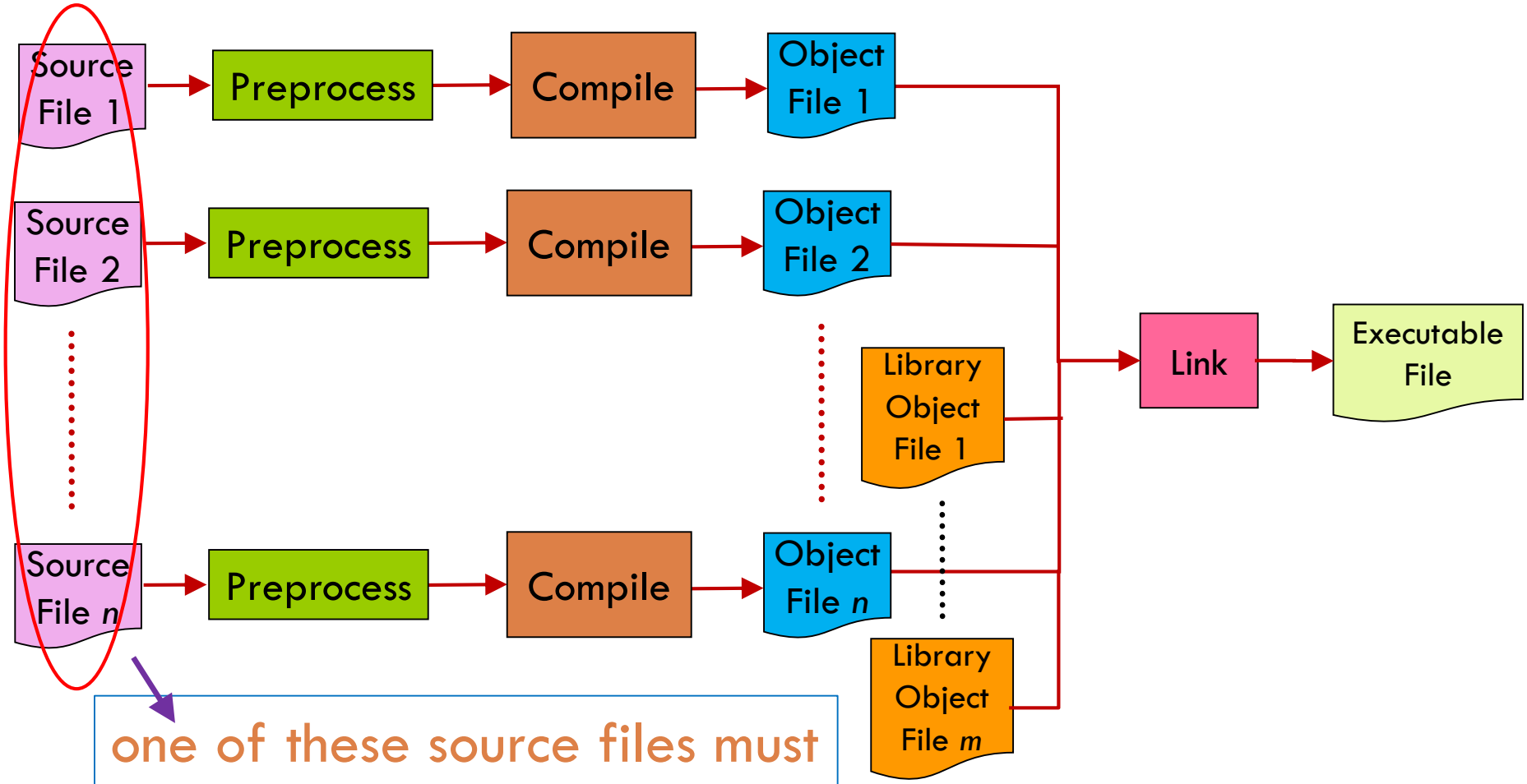
```c
// this is file hello.c
#include <stdio.h>

int main(void)
{
  printf(" Hello World\n");
  return 0;
}
```



Compilation Phases (2/2)

- Preprocess only (I'm not using all required gcc options for brevity)
  - `gcc -std=c11 -E hello.c -o hello.i`
- Compile only
  - `gcc -std=c11 -S hello.i -o hello.s`
- Assemble: `gcc -c hello.s -o hello.o`
- Link: `gcc hello.o -o hello.out`
- Execute: `./hello.out`

Creating a C Program (1/2)



one of these source files must define function `main`

Creating a C Program (2/2)

- Compiler consumes *each* source file individually without being aware of presence of other source files!!!
- Linker consumes *all* object files together to create executable file

Multiple Source Files (1/6)

- Deconstruct `hello.c` into two source files and one header file

```
// this is file hello.c
#include <stdio.h>

int main(void)
{
  printf("Hello World\n");
  return 0;
}
```

Multiple Source Files (2/6): hello-decl.h


```
// this is file hello-decl.h
#include <stdio.h>

// hello prints a greeting to standard output
void hello(void);
```

1

File stdio.h is included to provide *function prototype (or declaration)* of standard C library function printf

2

Declaration of identifier hello says "hello is a function that takes no input and returns no output"

Multiple Source Files (3/6): hello-defn.c


```c
#include "hello-decl.h"

// definition of function hello
void hello(void) {
  printf("Hello World!!!\n");
}
```

1

2

1) File hello-decl.h is included to provide *function prototype (or declaration)* of standard C library function printf and function hello

2) Pair of double quote delimiters " tells preprocessor to search for header file in current directory

Compile only!!!

3

```
gcc -std=c11 -pedantic-errors -Wstrict-prototypes
-Wall -Wextra -Werror -c hello-defn.c -o hello-defn.o
```

Multiple Source Files (4/6): driver.c


```c
#include "hello-decl.h"

int main(void) {
  hello();
  return 0;
}
```

1

File `hello-decl.h` is included to provide *function prototype (or declaration)* of function `hello`

2

1) Compiler doesn't care where or how function `hello` is defined
2) Compiler only cares whether call to `hello` *matches* declaration in `hello-decl.h`

3

```
gcc -std=c11 -pedantic-errors -Wstrict-prototypes
-Wall -Wextra -Werror -c driver.c -o driver.o
```

Multiple Source Files (5/6): driver.c


```c
#include "hello-decl.h"
#include <stdio.h>
int main(void) {
  hello();
  return 0;
}
```

1 File hello-decl.h is included to provide *function declaration (or prototype)* of function hello

2 File hello-decl.h includes C standard library file stdio.h

3

1) Can have multiple declarations of function printf!!!
2) But, there can be *only one definition* of a function (and a variable)
3) Otherwise, linker will not create executable.

4

```
gcc -std=c11 -pedantic-errors -Wstrict-prototypes
-Wall -Wextra -Werror -c driver.c -o driver.o
```

Multiple Source Files (6/6): Linking Object & Library Files

- Two object files:
  - `driver.o` with definition of function `main`
  - `hello-defn.o` with definition of function `hello`
- C standard library: `libc.a` with definition of function `printf`
  - Find location of C standard library file thro' this command: **`gcc -print-file-name=libc.a`**
- Link these files into executable `hello.out`

```
gcc driver.o hello-defn.o -o hello.out
```

Interface versus Implementation


```c
// this is file hello-decl.h
#include <stdio.h>

void hello(void);
```

1

1) Header file specifies *function prototypes*
2) Function prototype is an *interface* that only specifies *what* the function does

```c
// this is file hello-defn.c
#include "hello-decl.h"

// definition of function hello
void hello(void) {
  printf("Hello World!!!\n");
}
```

2

1) Source file specifies *function definitions*
2) Function definition is an *implementation* that specifies *how* function accomplishes purpose advertised by interface

Mathematical Functions


```c
#include <math.h>
#include <stdio.h>

int main(void) {
  double px = 0.0, py = 0.0;
  double qx = 3.0, qy = 4.0;
  double w = qx - px, h = qy - py;
  double dist = sqrt(w*w + h*h);
  printf("Distance is %f\n", dist);
  return 0;
}
```

1

File `math.h` is included to provide *function prototype (or declaration)* of standard C library function `sqrt`

2

Call to `printf` displays following line:

```
Distance is 5.000000
```

3

```
gcc -std=c11 -pedantic-errors -Wstrict-prototypes
-Wall -Wextra -Werror dist.c -o dist.out -lm
```

Shorthand for link object files with library file `libm.a`

4

Summary

- C program consists of source files with each file consisting of functions
- Function is encapsulation of algorithm
  - Think of function as statements grouped together and given a name
- C programs must have one and only one function called `main`
  - `main` function is starting point of execution of C program
- Each source file must be *individually compiled* into object file
- Object files are *linked together* into an executable
- Must explicitly link to C math standard library functions