600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

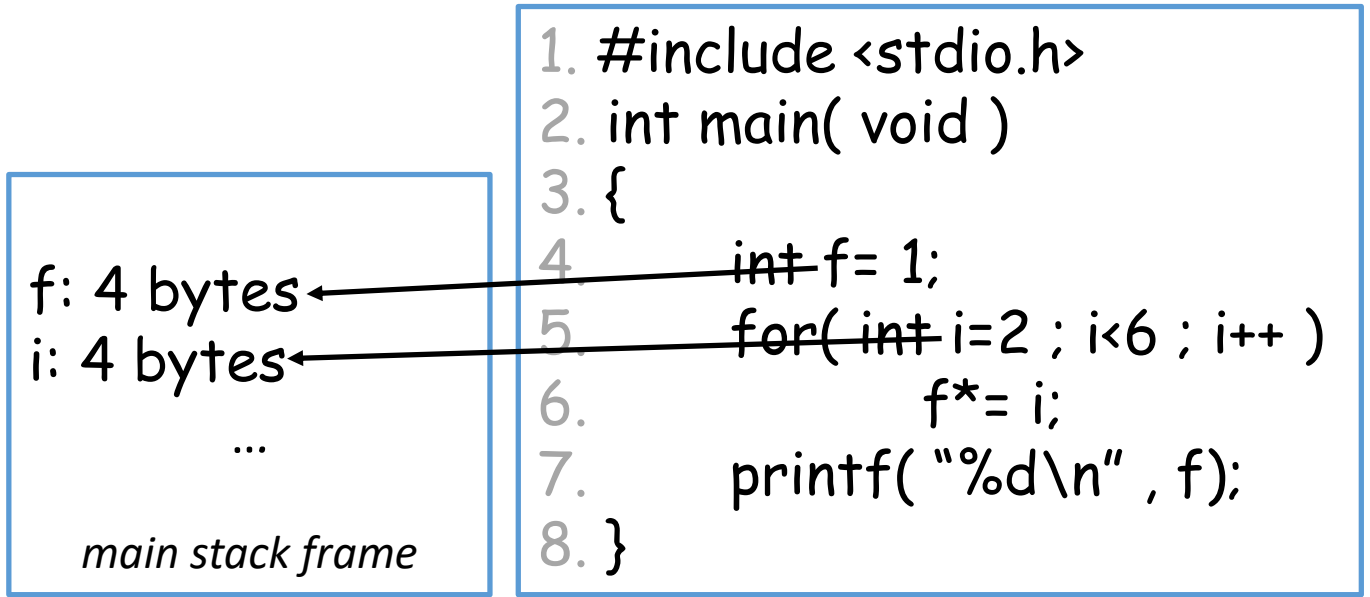
**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

Outline

- Variable lifetime and scope
- File I/O
- gdb

Variable lifetime and scope

- [Recall]
  - When a function is called, it's stack-frame is popped onto the stack
  - Local variables are stored on the frame
  - When a function returns, it's stack-frame is popped off




```
>> ./a.out
120
>>
```

Variable lifetime and scope

- Variables declared in C programs have:
  - *lifetime*: How long is the variable in memory?
    - Both f and i have a lifetime equal to the duration of the main function
      (They come into existence when main's stack frame is created and disappear when it's gone)
  - *scope*: Where is the variable name accessible?
    - f is in scope from the point it is declared to the end of the main function (lines 4-7)
    - i is in scope for the for loop (lines 5-6)

```
1. #include <stdio.h>
2. int main( void )
3. {
4.        int f= 1;
5.        for( int i=2 ; i<6 ; i++ )
6.                f*= i;
7.        printf( "%d\n" , f);
8. }
```

f: 4 bytes
i: 4 bytes
…
main stack frame

```
>> ./a.out
120
>>
```

Variable lifetime and scope

Q: What are the lifetimes of the variables i?

A: Both have a lifetime equal to the duration of the main function

Q: What are the scopes of the variables i?

A: The first comes into scope when it is declared, is hidden during the for loop, and re-emerges after (lines 4, 7)

The second is in scope during the for loop (lines 5-6)

```
1. #include <stdio.h>
2. int main( void )
3. {
4.         int i = 1;
5.         for( int i=0 ; i<2 ; i++ )
6.                 printf( "hi: %d\n" );
7.         printf( "bye: %d\n" , i );
8. }
```

i: 4 bytes

i: 4 bytes

...

main stack frame

```
>> ./a.out
hi: 0
hi: 1
bye: 1
>>
```

Variable lifetime and scope

- Variables declared in C programs have lifetime and scope
  - In general, local variables have lifetime / scope equal to the function's duration (assuming they aren't hidden by an inner variable with the same name)

```
#include <stdio.h>
void foo( int i )
{
    static int count;
    printf( "%d] foo( %d )\n" , count++ ,  i );
}
int main( void )
{
    foo( 1 ) ;
    foo( 7 );
    return 0;
}
```

Variable lifetime and scope

- Variables declared in C programs have lifetime and scope
  - In general, local variables have lifetime / scope equal to the function's duration (assuming they aren't hidden by an inner variable with the same name)
  - But… prefixing the variable declaration with the **static** keyword, extends the lifetime across all calls to that function
    - The variable is automatically initialized to have zero value (the bits in its bytes are cleared)

```c
#include <stdio.h>
void foo( int i )
{
    static int count;
    printf( "%d] foo( %d )\n" , count++ ,  i );
}
int main( void )
{
    foo( 1 ) ;
    foo( 7 );
    return 0;
}
```

```
>> ./a.out
0] foo( 1 )
1] foo( 7 )
>>
```

Variable lifetime and scope

- Variables declared in C programs have lifetime and scope
  - In general, local variables have lifetime / scope equal to the function's duration (assuming they aren't hidden by an inner variable with the same name)
  - But… prefixing the variable declaration with the `static` keyword, extends the lifetime across all calls to that function
  - But the variable is still only scoped within the function

```
#include <stdio.h>
void foo( int i )
{
    static int count;
    printf( "%d] foo( %d )\n" , count++ ,  i );
}
int main( void )
{
    foo( 1 );
    printf( "%d\n" , count );
    return 0;
}
```

Variable lifetime and scope

- Variables declared in C programs have lifetime and scope
  - We can also declare *global* variables outside of any function
    - They have a lifetime equal to the lifetime of the program
      - They are initialized to zero
    - They are accessible to any function following the declaration
      - They can even be made accessible to functions in other files using the **extern** keyword

```
#include <stdio.h>
int count;
void foo( int i )
{
    printf( "%d] foo( %d )\n" , count++ ,  i );
}
int main( void )
{
    foo( 1 );
    printf( "%d\n" , count );
    return 0;
}
```

```
>> ./a.out
0] foo( 1 )
1
>>
```

Beware the global variable

- Usage of global variables is generally discouraged
  - Debugging is harder – less clear which function changed a global variable's value (since it could be any!)
  - Global variables cross boundaries between program modules, undoing benefits of modular code
    - readability
    - testability
  - In general, values should be conveyed via parameter passing and return values

Variable lifetime and scope

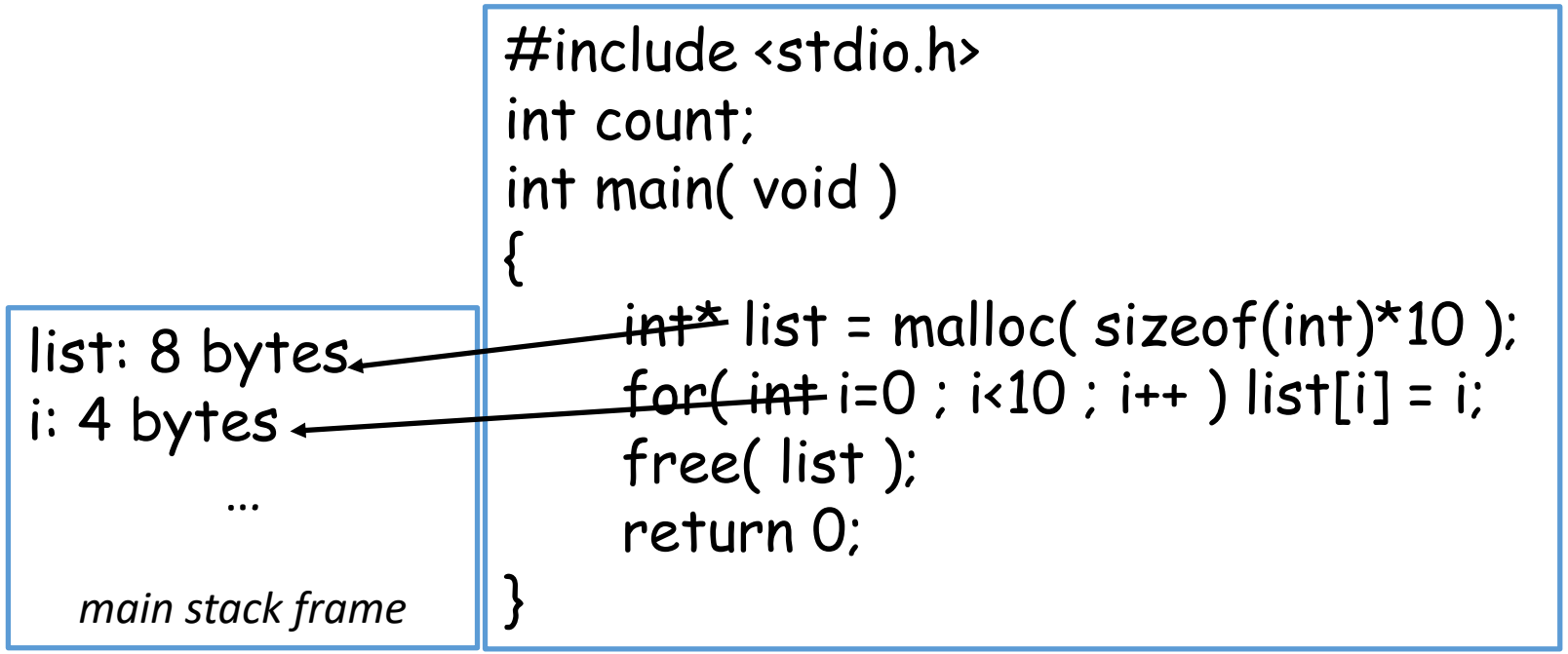
Q: Where do things live?

```
#include <stdio.h>
int count;
int main( void )
{
	int* list = malloc( sizeof(int)*10 );
	for( int i=0 ; i<10 ; i++ ) list[i] = i;
	free( list );
	return 0;
}
```

Variable lifetime and scope

Q: Where do things live?

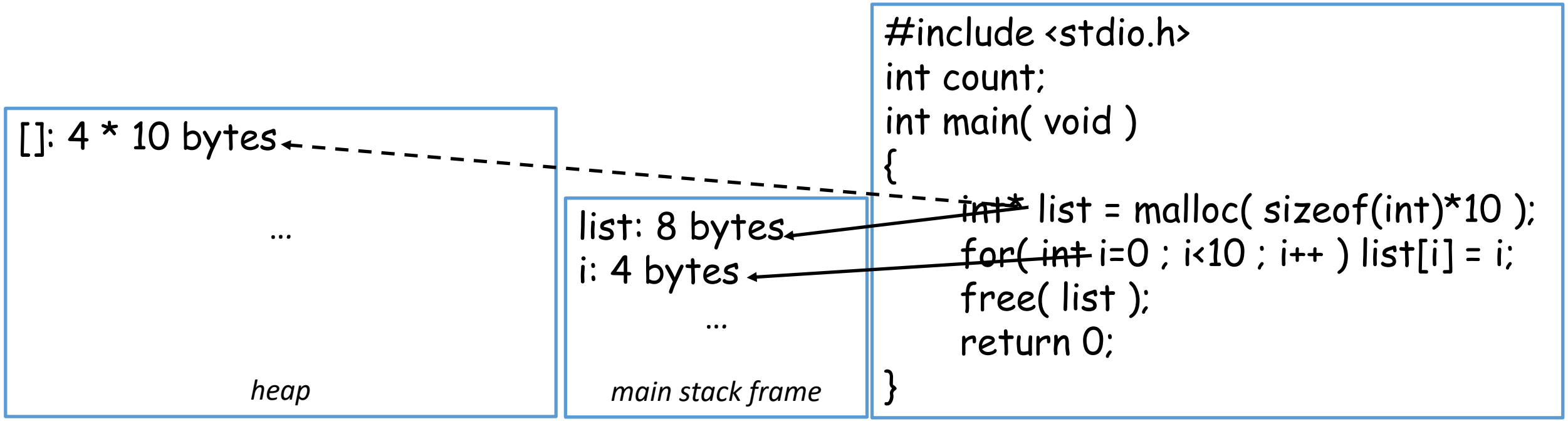
A: Local variables live on the stack



Variable lifetime and scope

Q: Where do things live?

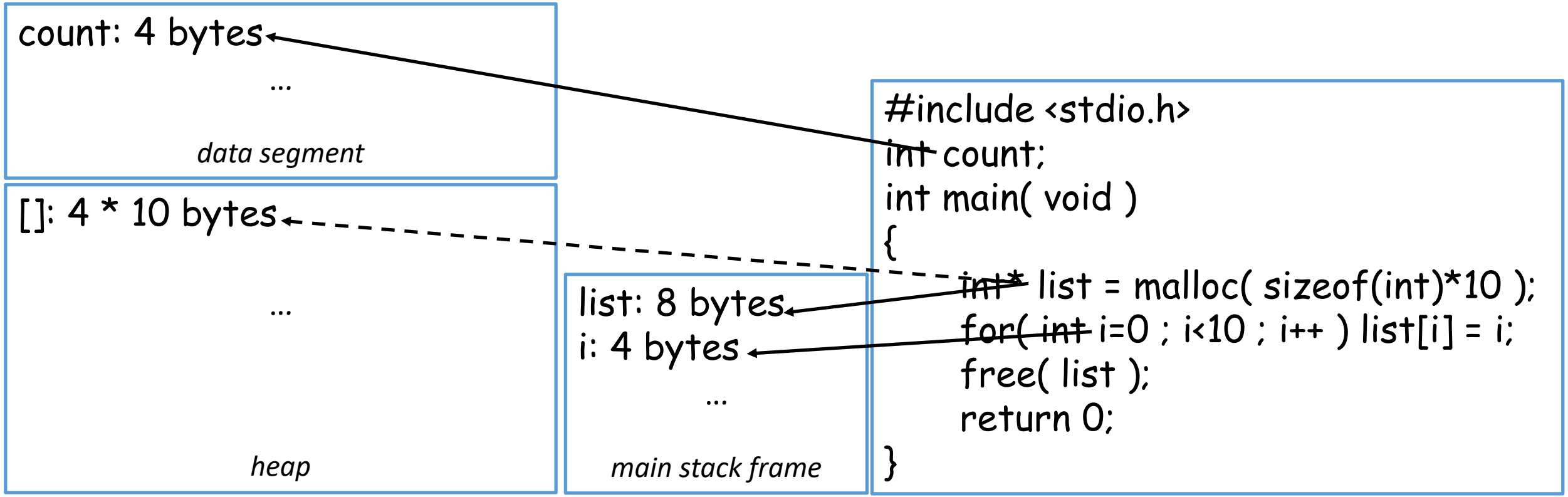
A: Dynamically allocated memory lives on the heap



Variable lifetime and scope

Q: Where do things live?

A: static and global variables live in the data segment



Outline

- Variable life-time and scope
- File I/O
- gdb

File I/O

- To read to / write from the command line, we use the commands
  - `int printf( const char format_str[] , … );`
  - `int scanf( const char format_str[] , … );`

- These are special instances of more general functions:
  - `int printf( format_str[] , … ) = fprintf( stdout , format_str , … );`
  - `int scanf( format_str[] , … ) = fscanf( stdin , format_str , … );`

- `stdout` and `stdin` are instances of file-handles

File-handles

- Different operating systems store data in different ways
- To avoid having to tailor code to the OS, C supports *file-handles*
  - These are abstract representations of objects we can read from / write to
    - Files on disk
    - Command line
    - Sockets across a network
    - etc.

File-handles

- When working with file handles we:
  1. Create a file handle
  2. Access the file's contents
  3. Close the handle

```
#include <stdio.h>
int main( void )
{
	FILE* fp = fopen( "foo.txt" , "w" );
	if( !fp )
	{
		fprintf( stderr , … );
		return 1;
	}
	fprintf( fp , "hello\n" );
	fclose( fp );
}
```

File-handles (opening)

```
FILE* fopen( const char * file_name , const char * mode );
```

```
#include <stdio.h>
int main( void )
{
	FILE* fp = fopen( "foo.txt" , "w" );
	if( !fp )
	{
		fprintf( stderr , … );
		return 1;
	}
	fprintf( fp , "hello\n" );
	fclose( fp );
}
```

File-handles (opening)

FILE* fopen(const char * file_name , const char * mode);

- Input:
  - The name of the file

```
#include <stdio.h>
int main( void )
{
	FILE* fp = fopen( "foo.txt" , "w" );
	if( !fp )
	{
		fprintf( stderr , … );
		return 1;
	}
	fprintf( fp , "hello\n" );
	fclose( fp );
}
```

File-handles (opening)

FILE* fopen(const char * file_name , const char * mode);

- Input:
  - The name of the file
  - The mode in which to open the file
    This is a string consisting of characters indicating access intent
    - ‘r’: read
    - ‘w’: write
    - ‘a’: append
    - ‘b’: binary

```
#include <stdio.h>
int main( void )
{
	FILE* fp = fopen( “foo.txt” , “w” );
	if( !fp )
	{
		fprintf( stderr , … );
		return 1;
	}
	fprintf( fp , “hello\n” );
	fclose( fp );
}
```

File-handles (opening)

<u>FILE*</u> fopen(const char * file_name , const char * mode);

- Input:
  - The name of the file
  - The mode in which to open the file
    This is a string of characters indicating intent
- Output:
  - A pointer to a file-handle
    - We need a pointer because the file-handle tracks the file's state (e.g. where are we in the file) and this will change as we perform I/O

```
#include <stdio.h>
int main( void )
{
	FILE* fp = fopen( "foo.txt" , "w" );
	if( !fp )
	{
		fprintf( stderr , … );
		return 1;
	}
	fprintf( fp , "hello\n" );
	fclose( fp );
}
```

File-handles (opening)

FILE* fopen(const char * file_name , const char * mode);

- Input:
  - The name of the file
  - The mode in which to open the file
    This is a string of characters indicating intent
- Output:
  - A pointer to a file-handle
    - The function returns NULL if the system couldn't open the file
      - reading: file doesn't exist
      - writing: file/directory isn't ours
      - writing: the file is already open

```
#include <stdio.h>
int main( void )
{
	FILE* fp = fopen( "foo.txt" , "w" );
	if( !fp )
	{
		fprintf( stderr , … );
		return 1;
	}
	fprintf( fp , "hello\n" );
	fclose( fp );
}
```

File-handles (opening)

<u>FILE*</u> fopen(const char * file_name , const char * mode);

- Input:
  - The name of the file
  - The mode in which to open the file
    This is a string of characters indicating intent
- Output:
  - A pointer to a file-handle
    - The function returns NULL if the system couldn't open the file
    - ⇒ Check to make sure the command succeeded

```
#include <stdio.h>
int main( void )
{
	FILE* fp = fopen( "foo.txt" , "w" );
	if( !fp )
	{
		fprintf( stderr , … );
		return 1;
	}
	fprintf( fp , "hello\n" );
	fclose( fp );
}
```

File-handles (accessing)

- There are a number of commands for reading from / writing to a file
  - Writing:
    - int fprintf(FILE* fp , const char format_str[] , …);
      - Writes a formatted string to the specified file-handle
      - Returns the number characters written (a negative value if the write failed)

```
#include <stdio.h>
int main( void )
{
	FILE* fp = fopen( "foo.txt" , "w" );
	if( !fp ) …
	fprintf( fp , "hello\n" );
	fclose( fp );
}
```

File-handles (accessing)

- There are a number of commands for reading from / writing to a file
  - Writing:
    - int fputc(int character , FILE * fp);
      - Writes a single character to the specified file-handle
      - Returns the character written (EOF* if the write failed)

```
#include <stdio.h>
int main( void )
{
	char str[] = "hello";
	FILE* fp = fopen( "foo.txt" , "w" );
	if( !fp ) …
	for( int i=0 ; i<strlen(str) ; i++ ) fputc( str[i] , fp );
	fclose( fp );
}
```

*EOF is the end-of-file character

File-handles (accessing)

- There are a number of commands for reading from / writing to a file
  - Reading:
    - int fscanf(FILE* fp , const char format_str[] , …);
      - Reads a formatted string from the specified file-handle
      - Returns the number of variables successfully set

```
#include <stdio.h>
int main( void )
{
	char word[512];
	FILE* fp = fopen( "foo.txt" , "r" );
	if( !fp ) …
	while( fscanf( fp , "%s" , word )==1 ) printf( "Read: %s\n" , word );
	fclose( fp );
}
```

File-handles (accessing)

- There are a number of commands for reading from / writing to a file
  - Reading:
    - int fscanf(FILE* fp , const char format_str[] , …);
      - Reads a formatted string from the specified file-handle
      - Returns the number of variables successfully set

```
#include <stdio.h>
int main( void )
{
	char word[512];
	FILE* fp = fopen( "foo.txt" , "r" );
	if( !fp ) …
	while( fscanf( fp , "%s" , word )==1 ) printf( "Read: %s\n" , word );
	fclose( fp );
```

[Recall] This function could be unsafe if we read in a string longer than **str**

File-handles (accessing)

- There are a number of commands for reading from / writing to a file
  - Reading:
    - int fgets(char * str , int num, FILE* fp);
      - Reads characters from a file-handle until:
        - The string buffer is filled
        - A new line is reached
        - EOF is reached
      - Returns str
        (NULL if the read failed)

```
#include <stdio.h>
int main( void )
{
	char str[512];
	FILE* fp = fopen( "foo.txt" , "r" );
	if( !fp ) …
	while( fgets( str , 512 , fp ) ) printf( "%s" , str );
	fclose( fp );
}
```

File-handles (accessing)

- There are a number of commands for reading from / writing to a file
  - Reading:
    - int fgetc(FILE* fp);
      - Reads a single character from the specified file-handle
      - Returns the character written (EOF if the read failed)

```
#include <stdio.h>
int main( void )
{
	char c;
	FILE* fp = fopen( "foo.txt" , "r" );
	if( !fp ) …
	while( ( c=fgetc( fp ) )!=EOF ) printf( "%c" , c );
	fclose( fp );
}
```

File-handles (closing)

```
int fclose( FILE * fp );
```

- Input:
  - The file-handle
- Output:
  - Returns 0 if the file was successfully closed (EOF if it wasn't)

```
#include <stdio.h>
int main( void )
{
	char c;
	FILE* fp = fopen( "foo.txt" , "r" );
	if( !fp ) …
	while( ( c=fgetc( fp ) )!=EOF ) printf( "%c" , c );
	fclose( fp );
}
```

`stdin`, `stdout`, and `stderr`

- C defines three file-handles:
  - standard input (`stdin`): the command prompt, for reading
  - standard output (`stdout`): the command prompt, for writing
  - standard error (`stderr`): the command prompt, for writing error messages

stdin, stdout, and stderr

stdout and stderr are both file-handles that allow writing to the command prompt

```
#include <stdio.h>
int main( void )
{
	fprintf( stdout, "This is not an error message\n" );
	fprintf( stderr , "This is an error message\n" );
	return 0;
}
```

```
>> ./a.out
This is not an error message
This is an error message
>>
```

stdin, stdout, and stderr

stdout and stderr are both file-handles that allow writing to the command prompt

- These are separate file-handles! (e.g. You can redirect them separately)

```
#include <stdio.h>
int main( void )
{
	fprintf( stdout, "This is not an error message\n" );
	fprintf( stderr , "This is an error message\n" );
	return 0;
}
```

```
>> ./a.out > foo.txt
This is an error message
>>
```

stdin, stdout, and stderr

stdout and stderr are both file-handles that allow writing to the command prompt

- These are separate file-handles! (e.g. You can redirect them separately)

```
#include <stdio.h>
int main( void )
{
	fprintf( stdout, "This is not an error message\n" );
	fprintf( stderr , "This is an error message\n" );
	return 0;
}
```

```
>> ./a.out > foo.txt
This is an error message
>> more foo.txt
This is not an error message
>>
```

Outline

- Variable life-time and scope
- File I/O
- gdb

Debugging

- A *debugger* is a tool for removing errors (“bugs”) from software, especially useful for software crashes (e.g., segmentation faults)
- Allows you to run your code in an environment where you can pause its execution and check values of variables
- More versatile than using `printf` statements throughout your code
  - Can decide what variables to inspect and what to evaluate, on-the-fly
- gdb is the GNU debugger
  - See tutorials on class Tools page, minimally work through the first one

Getting started with **gdb**

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

```
>> ./a.out
-48
>>
```

Getting started with **gdb**

- Use the `-g` flag to compile an executable `a.out` which can be used by gdb

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

```
>> gcc -std=c99 -pedantic -Wall -Wextra -g foo.c
>>
```

Getting started with **gdb**

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

- Use the `-g` flag to compile an executable `a.out` which can be used by gdb
- Launch gdb to debug the program

```
>> gcc -std=c99 -pedantic -Wall -Wextra -g foo.c
>> gdb ./a.out
...
```

Getting started with **gdb**

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

- Use the `-g` flag to compile an executable `a.out` which can be used by gdb
- Launch gdb to debug the program
- This will put you in the gdb environment and you will have a new prompt where you can enter debugging commands

```
>> gcc -std=c99 -pedantic -Wall -Wextra -g foo.c
>> gdb ./a.out
...
(gdb)
```

Getting started with **gdb**

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

- Use the `-g` flag to compile an executable `a.out` which can be used by gdb
- Launch gdb to debug the program
- This will put you in the gdb environment and you will have a new prompt where you can enter debugging commands
- When you're finished using gdb, exit by typing "`quit` " or just "q"

```
>> gcc -std=c99 -pedantic -Wall -Wextra -g foo.c
>> gdb ./a.out
...
(gdb) quit
>>
```

Getting started with **gdb**

- Within the debugger:
  - `list`: display source code with line numbers

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

```
(gdb) list
1       #include <stdio.h>
2       int main( void )
3       {
4         char f = 1;
5         for( int i=1 ; i<7 ; i++ )
6           f *= i;
7         printf( "%d\n" , f);
8   }
(gdb)
```

Getting started with **gdb**

- Within the debugger:
  - `break <line number>`: add a breakpoint at the specified line number

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

```
(gdb) break 6
Breakpoint 1 at 0x40050b: file foo.c, line 6.
(gdb)
```

Getting started with **gdb**

- Within the debugger:
  - `break <line number>`: add a breakpoint at the specified line number
  - `run <command line args>`: execute the program (with specified arguments) until a breakpoint is reached

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

```
(gdb) break 6
Breakpoint 1 at 0x40050b: file foo.c, line 6.
(gdb) run
...
Breakpoint 1, main () at foo.c:6
6           f *= i;
(gdb)
```

Getting started with **gdb**

- Within the debugger:
  - `break <line number>`: add a breakpoint at the specified line number
  - `run <command line args>`: execute the program (with specified arguments) until a breakpoint is reached
  - `print [<format>] <expr>`: outputs the [formatted] value of the expression (often a single variable)

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

```
(gdb) break 6
Breakpoint 1 at 0x40050b: file foo.c, line 6.
(gdb) run
...
Breakpoint 1, main () at foo.c:6
6           f *= i;
(gdb) print /d f
$1 = 1
(gdb)
```

Getting started with **gdb**

- Within the debugger:
  - `break <line number>`: add a breakpoint at the specified line number
  - `run <command line args>`: execute the program (with specified arguments) until a breakpoint is reached
  - `print [<format>] <expr>`: outputs the (formatted) value of the expression (often a single variable)
  - `continue`: resume execution until next breakpoint is reached

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

```
(gdb) break 6
Breakpoint 1 at 0x40050b: file foo.c, line 6.
(gdb) run
...
Breakpoint 1, main () at foo.c:6
6           f *= i;
(gdb) print /d f
$1 = 1
(gdb) continue
Continuing.

Breakpoint 1, main () at foo.c:6
6           f *= i;
(gdb)
```

Getting started with **gdb**

- Within the debugger:
  - `break <line number>`: add a breakpoint at the specified line number
  - `run <command line args>`: execute the program (with specified arguments) until a breakpoint is reached
  - `print [<format>] <expr>`: outputs the (formatted) value of the expression (often a single variable)
  - `continue`: resume execution until next breakpoint is reached

```
#include <stdio.h>
int main( void )
{
        char f = 1;
        for( int i=2 ; i<7 ; i++ )
                f*= i;
        printf( "%d\n" , f);
}
```

```
(gdb) break 6
Breakpoint 1 at 0x40050b: file foo.c, line 6.
(gdb) run
...
Breakpoint 1, main () at foo.c:6
6           f *= i;
(gdb) print /d f
$1 = 1
(gdb) continue
Continuing.

Breakpoint 1, main () at foo.c:6
6           f *= i;
(gdb) print /d f
$2 = 2
(gdb)
```

Getting started with **gdb**

- Managing breakpoints
  - `break <line number>`: add a breakpoint at the specified line number
  - `delete <breakpoint num>`: remove specified breakpoint
  - `info breakpoints`: display current list of breakpoints
  - It is also possible to set conditional breakpoints,
    e.g., if we want to pause execution at line 31 only if variable `x` is positive:

```
(gdb) break 31 if x >0
```

Getting started with **gdb**

- Advancing the debugger:
  - `continue`: resume execution until next breakpoint is reached
  - `step`: single-steps execution forward, one source line at a time
  - `next`: single-steps execution forward, but treats function call as a single item
  - `finish`: runs until end of the current function

Getting started with **gdb**

- Other commands
  - `watch <variable>`: pauses program whenever this variable's value changes, and outputs its old and new values
  - `backtrace`: produces a stack trace of function calls that lead to a segmentation fault

Tips:

- Command names can be shortened as long as there is no ambiguity, e.g. type “b 5 ” to set a breakpoint at line 5, or “p x ” to print the value of x
- Hitting return re-executes the last command

Piazza → Resources section → Resources tab → Exercise 4-1