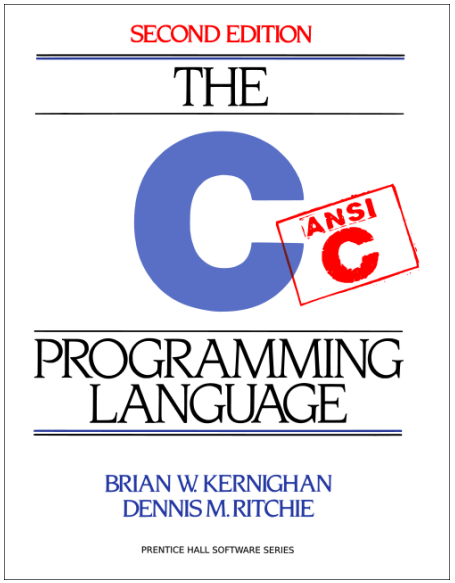
# CS270

# C Programming Tutorial: In class notes

UNIX®

Celebrating 40 years uptime




UNIVERSITY of WISCONSIN
LA CROSSE™

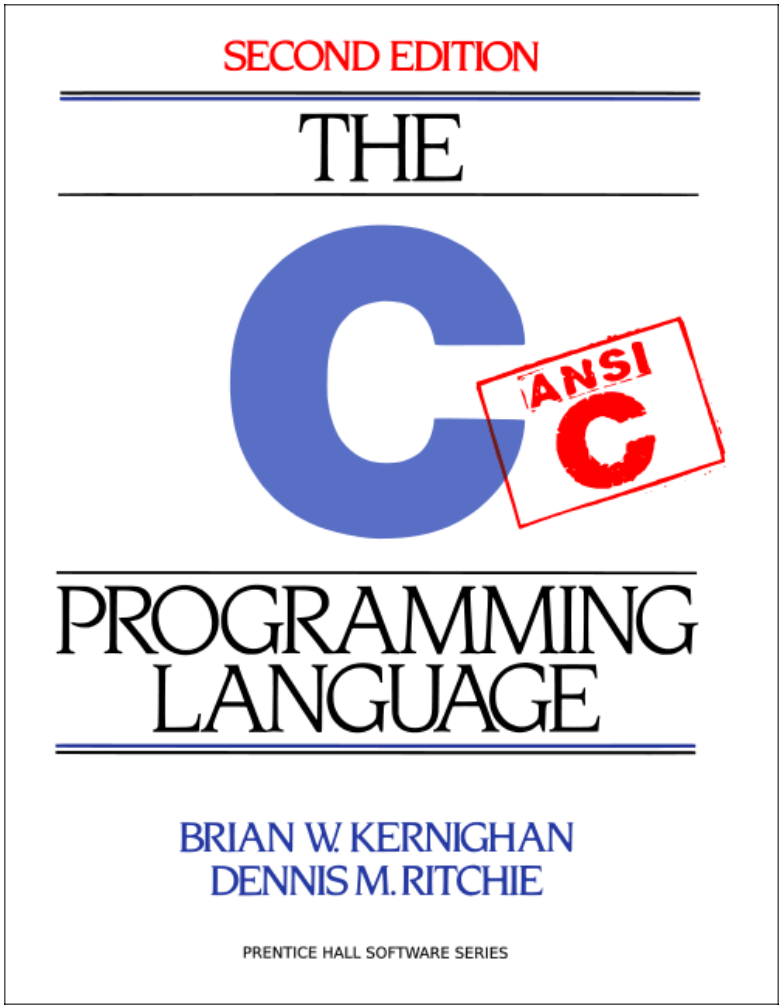
Professor Foley

# Announcements

- Portfolios that were not picked up are graded and can be picked up from my office.
- Thursdays from now on will be in CS Lab Wing 016
- Final Exam will be Saturday 12/17 at 7:45 – 9:45am
  - Exam will be in ***Centennial 3215***
  - Comprehensive
  - MIPS card
  - Calculator
  - 1 sheet of notes, standard letter sized paper, 1 sided, hand written.

# Preparation for Lab

- Complete the command-line tutorial at codecademy: https://www.codecademy.com/learn/learn-the-command-line
- Prepare your development environment
  - If you have a Mac, you should have a C compiler accessible from the command-line.
  - If you have a Linux system, you should have a C compiler accessible from the command-line.
  - If you have a Windows system, you can download and install Cygwin or another UNIX-like environment that can run C programs. Alternatively, you can run a virtual machine with Linux.
  - You may also do your development in the CS Lab.
  - There may be other alternative solutions, but it is best to make sure that your code runs on the CS Lab machines.

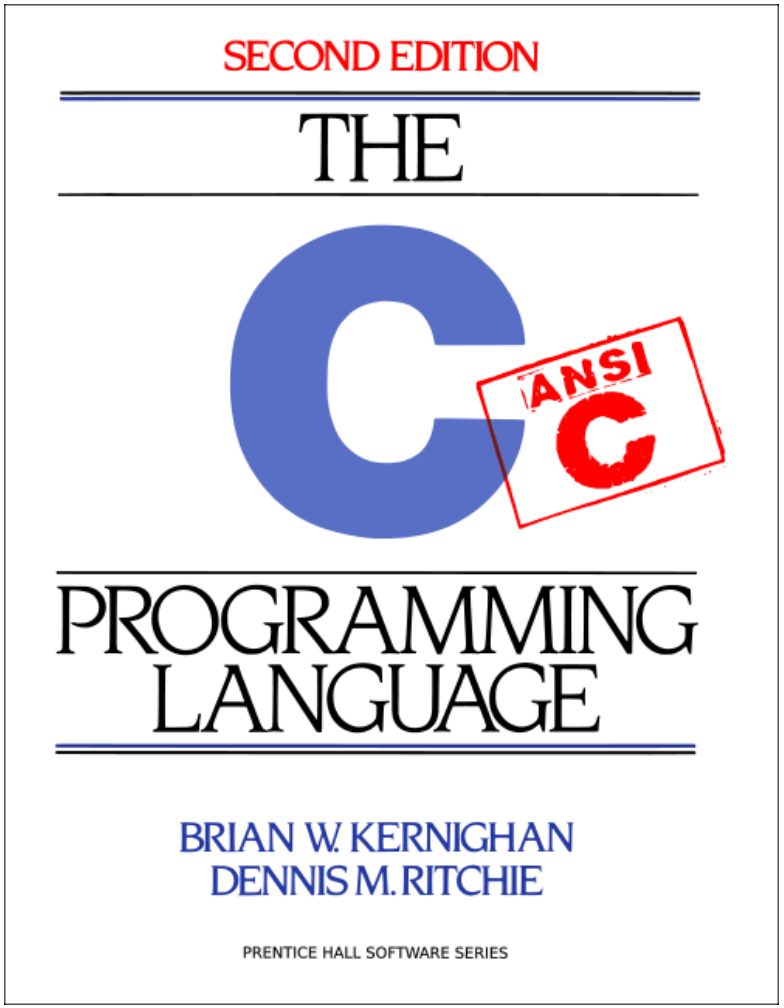



# C Programming

Kernighan, B.W., Ritchie, D.M., "The C Programming Language (Second Edition)." 1988.

# C vs. Java vs. MIPS

- Java is a high level language which is designed to be object-oriented. It is a descendant of C, so there are some similarities between the languages.
- C is also a high level language because it is not an assembly language, however it is not object-oriented. There are no objects at all.
- C is useful for several reasons, some of which are:
  - It helps you to understand how data is manipulated in higher-level languages like Java.
  - It helps you appreciate the convenience of OO.
  - It is used in systems software, embedded software, and high performance applications because it is a smaller and faster language which is closer to assembly.




# Variables & Control Structures

# C Programming: Primitive Datatypes & Variables

Same as Java!
Except void *

- Primitive Datatypes in C (there are no classes – that's C++)
  - `char` Single byte. Capable of holding one character.
  - `int` An integer. (usually 32 bits)
  - `float` A single-precision floating point.
  - `double` A double-precision floating point.
  - `void*` A memory address

**Style Req.:**
Declare all local variables at the top of the function, and all global variables at the top of the file.

```
#include <stdio.h>
int main(int argc, char **argv) {
    int x, y;
    int z = 10;
    double result = 0;

    x = 3;
    y = 8;
    result = (x + y) / z;
    printf("Result %f\n", result);

    return 0;
}
```

Result 1.0

```
#include <stdio.h>
int main(int argc, char **argv) {
    int x, y;
    int z = 10;
    double result = 0;

    x = 3;
    y = 8;
    result = (x + y) / ((double)z);
    printf("Result %f\n", result);

    return 0;
}
```

Result 1.1

# C Programming: I/O

- Output
  - `printf(<format string>, <arguments>);`
    Example: `printf("Result %d + %6d = %8d\n", x, y, result);`
  - `fprintf(<stream>, <format string>, <arguments>);`
    Example: `fprintf(stderr, "Result %d / %d = %6.2f\n", x, y, result);`
- Predefined file streams
  - `stdout` OUT Standard Output
  - `stderr` OUT Standard Error
  - `stdin` IN Standard in

This is useful for managing output in assignments.

| Special Characters | |
|---|---|
| \n | newline |
| \t | tab |
| \\ | \ |
| \" | " |
| \r | Carriage return |
| \f | Form Feed |

| Format | |
|---|---|
| %d | integer |
| %f | float, double |
| %c | char |
| %s | char * (String) |

| Precision ( _ = space) | |
|---|---|
| %3d | ___<br>_12 |
| %-3d | ___<br>12_ |
| %6.2f | ___.__<br>__1.00<br>_13.14 |

# C Programming: Selection Logic (Just like Java)

```
if ( condition ) {
  clauseStatements;
}
```

```
if ( condition ) {
  then_clause;
}
else {
  else_clause;
}
```

```
if ( condition1 ) {
  clause1;
  if ( condition2 ) {
    clause2;
  }
  else {
    if ( condition3 ) {
      clause3;
    }
    else {
      else_clause3;
    }
  }
}
else {
  else_clause;
}
```

```
if ( condition1 ) {
  clause1;
}
else if ( condition2 ) {
  clause2;
}
else if ( condition3 ) {
  clause3;
}
else {
  else_clause;
}
```

# C Programming: Selection Logic

Same as Java!

```
if ( condition1 ) {
  clause1;
}
else if ( condition2 ) {
  clause2;
}
else if ( condition3 ) {
  clause3;
}
else {
  else_clause;
}
```

```
#include <stdio.h>

int main(int argc, char **argv) {
  int i = 12;

  if( 0 == i % 2 ) {
    printf("%2d is even!\n", i);
  }
  else if( 0 == i % 3 ) {
    printf("%2d is odd and divisible by 3!\n", i);
    break;
  }
  else {
    printf("%2d is odd!\n", i);
  }

  return 0;
}
```

# C Programming: Selection Logic

Same as Java!

```
switch ( statement ) {
  case constant1:
    clause1;
    break;
  case constant2:
    clause2;
    break;
  case constant3:
    clause3;
    break;
  case default:
    default_clause;
    break;
}
```

```
#include <stdio.h>

int main(int argc, char **argv) {
  int i = 12;

  switch( i ) {
    case 1:
      printf("Option 1\n");
      break;
    case 2:
    case 3:
      printf("Option 2 or 3\n");
      break;
    case 4:
    default:
      printf("Unrecognized Option!\n");
  }

  return 0;
}
```

# C Programming: Repetition (Just like Java)

Same as Java!

```
while ( condition ) {
  loopStatements;
}
```

```
int i = 0;
while( i < 10 ) {
  printf("Index %d\n", i);
}
```

```
do {
  loopStatements;
} while( condition );
```

```
int i = 0;
do {
  printf("Select an index\n");
  i = get_user_input();
} while( i > 0 );
```

```
for ( init ; condition ; progress ) {
  loopStatements;
}
```

```
int i;
for( i = 0; i < 10; i++ ) {
  printf("Counter %d\n", i);
}
```

# C Programming: Repetition

Declaration needs to be outside loop for strict compilers.

```
for ( init ; condition ; progress ) {
  loopStatements;
}
```

```
for( int i = 0; i < 10; i++ ) {
  printf("Counter %d\n", i);
}
```

```
shell$ gcc prog.c -o prog
prog.c: In function 'main':
prog.c:5:5: error: 'for' loop initial declarations are only allowed in C99 mode
prog.c:5:5: note: use option -std=c99 or -std=gnu99 to compile your code
```

Loop variable should be declared outside of the for loop initializer.

```
int i;
for( i = 0; i < 10; i++ ) {
  printf("Counter %d\n", i);
}
```

# C Programming: I/O

`printf(<format string>, <arguments>);`

```
int x = 10;
printf("x = (%d)\n", x);
```

```
shell$ ./prog
x = (10)
```

```
int x = 10;
printf("x = (%5d)\n", x);
```

```
shell$ ./prog
x = (   10)
```

```
int x = 10;
printf("x = (%-5d)\n", x);
```

```
shell$ ./prog
x = (10   )
```

```
double x = 10 / 3.0;
printf("x = (%f)\n", x);
```

```
shell$ ./prog
x = (3.33333)
```

```
double x = 10 / 3.0;
printf("x = (%3.2f)\n", x);
```

```
shell$ ./prog
x = (3.33)
```

```
double x = 10 / 3.0;
printf("x = (%6.2f)\n", x);
```

```
shell$ ./prog
x = (  3.33)
```

```
double x = 10 / 3.0;
printf("x = (%-6.2f)\n", x);
```

```
shell$ ./prog
x = (3.33  )
```

| Format | |
|---|---|
| %d | integer |
| %f | float, double |
| %c | char |
| %s | char * (String) |

| Precision ( _ = space) | |
|---|---|
| %3d | ___<br>_12 |
| %-3d | ___<br>12_ |
| %6.2f | ___.__<br>__1.00<br>_13.14 |

| Special Characters | |
|---|---|
| \n | newline |
| \t | tab |
| \\ | \ |
| \" | " |
| \r | Carriage return |
| \f | Form Feed |

# C Programming: I/O

`printf(<format string>, <arguments>);`

More formatting with escaping special characters and printing strings.

```
int x = 13, y = 5;
int result = x % y;
printf("%d %% %d = %d\n", x, y, result);
```

```
shell$ ./prog
13 %  5 =  3
```

```
char str[] = "Hello";
char pun = '!';
printf("%s%c\n", str, pun);
```

```
shell$ ./prog
Hello!
```

```
char str[] = "Hello";
char pun = '!';
printf("\"%s%c\"\n", str, pun);
```

```
shell$ ./prog
"Hello!"
```

```
char str[] = "Hello";
char pun = '!';
printf("\"%7s%3c\"\n", str, pun);
```

```
shell$ ./prog
"  Hello  !"
```

| Format | |
|---|---|
| %d | integer |
| %f | float, double |
| %c | char |
| %s | char * (String) |

| Special Characters | |
|---|---|
| \n | newline |
| \t | tab |
| \\ | \ |
| \" | " |
| \r | Carriage return |
| \f | Form Feed |

# Hello World!

```
/*
 * Samantha Foley
 * Example Program
 *
 * A Hello World C program
 */

#include <stdio.h>

int main(int argc, char **argv) {
  // Print to stdout
  printf("Hello World\n");
  return 0;
}
```

**Libraries**

**Main Function**
Arguments are command line parameters.

**Print formatted output to stdout**

**Value to return from this function**
(to the shell in this case)

```
shell$ ls
hello.c
shell$ gcc hello.c –o hello
shell$ ./hello
Hello World!
```

# C Programming: Header Files & Libraries

- #include
  - Insert at that location the contents of the file specified.
- #include <stdio.h>
  - System header file.
  - Designated with <>
- #include "support.h"
  - User defined header file
  - Designated with ""

prog.c

```
#include "support.h"

int main(int argc, char **argv) {
    int start = 10, result;

    result = sum_up_to_x(start);

    printf("Sum from 0 to %2d is %3d\n",
           start, result);
    return 0;
}
```

support.h

```
#include <stdio.h>

int sum_up_to_x(int x);
```

# C Programming: Compiling

- GNU C Compiler: **gcc**
  - `gcc hello.c –o hello`
    Compile and link the program. Place results in hello (-o).
  - `gcc hello.c –g –O0 –o hello`
    Compile and link the program.
    Include debugger symbols (-g), and turn off optimizations (-O0).
  - `gcc hello.c –Wall –o hello`
    Compile and link the program.
    Turn on various warning checks (a good thing to leave on) (-Wall).
  - `gcc hello.c –c –o hello.o`
    Compile, but do not link the program (-c, -o generates **`hello.o`**).

- Suggested compiling command:
  - `gcc hello.c –Wall –g –O0 –o hello`

http://linux.die.net/man/1/gcc

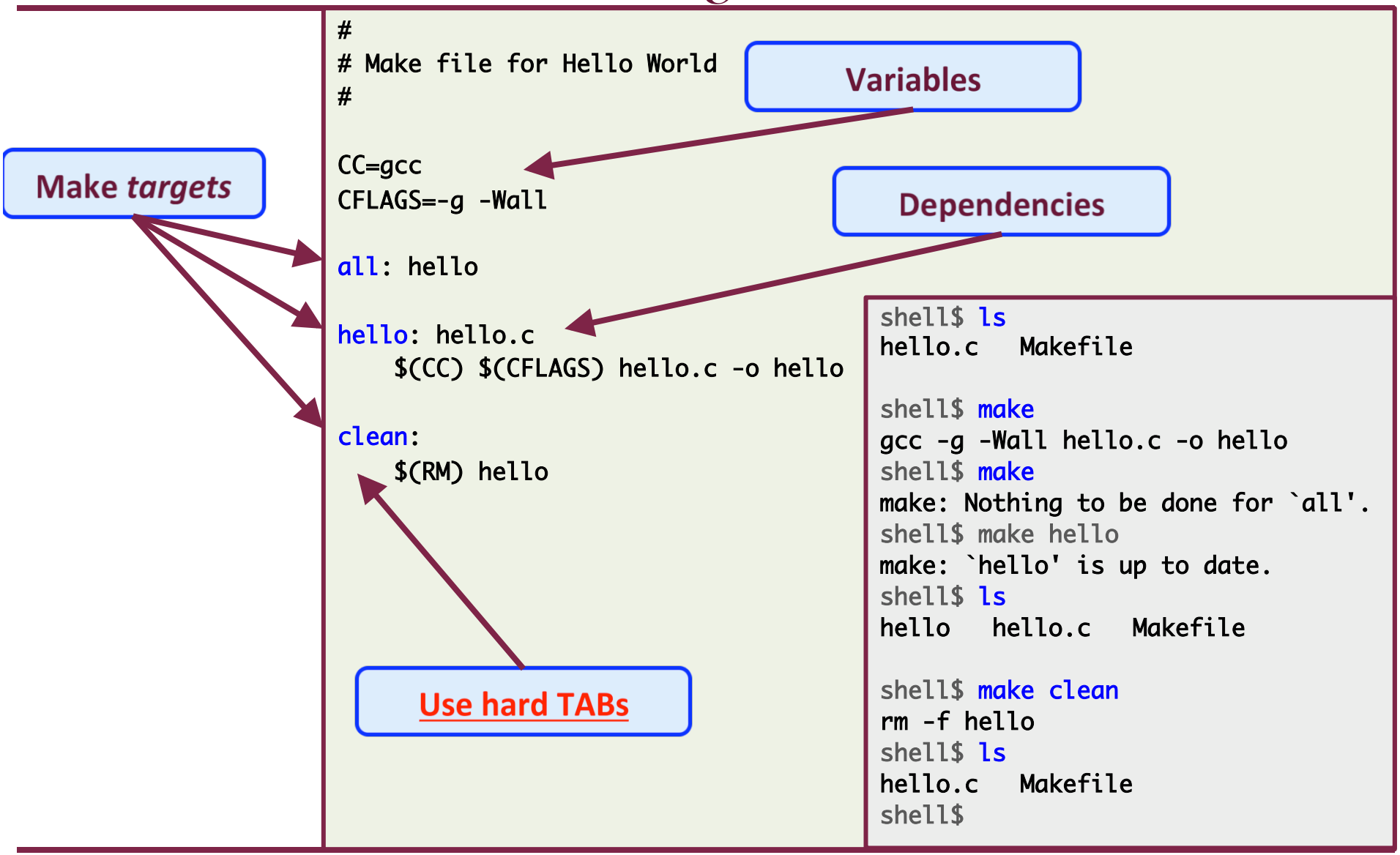
# Makefile

- GNU `make` is a tool that controls the generation of executables from program source(s).
- A `Makefile` describes how to build your program.
  - Ease of Use:
    Makefiles allow the end user can build & install your software without knowing the details of how to do so.
  - Dependency Tracking:
    Make is smart enough to only re-compile those sources that depend on sources that have changed since the last build – Improved build times.

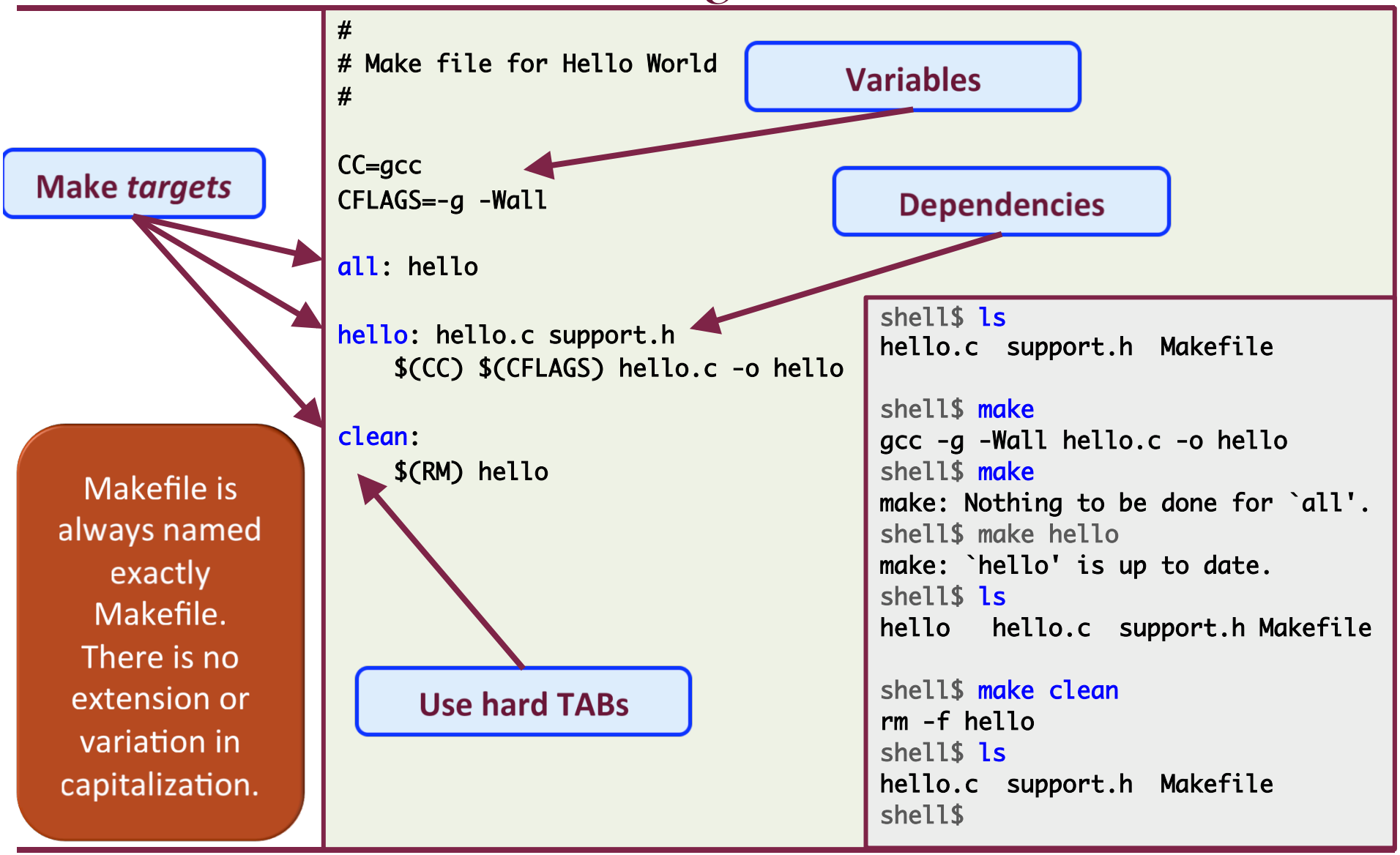
```
shell$
shell$ gcc -o prog prog.c
shell$
shell$ rm prog
shell$
shell$ gcc -o prog prog.c
shell$
shell$ gcc -o prog prog.c
shell$
```

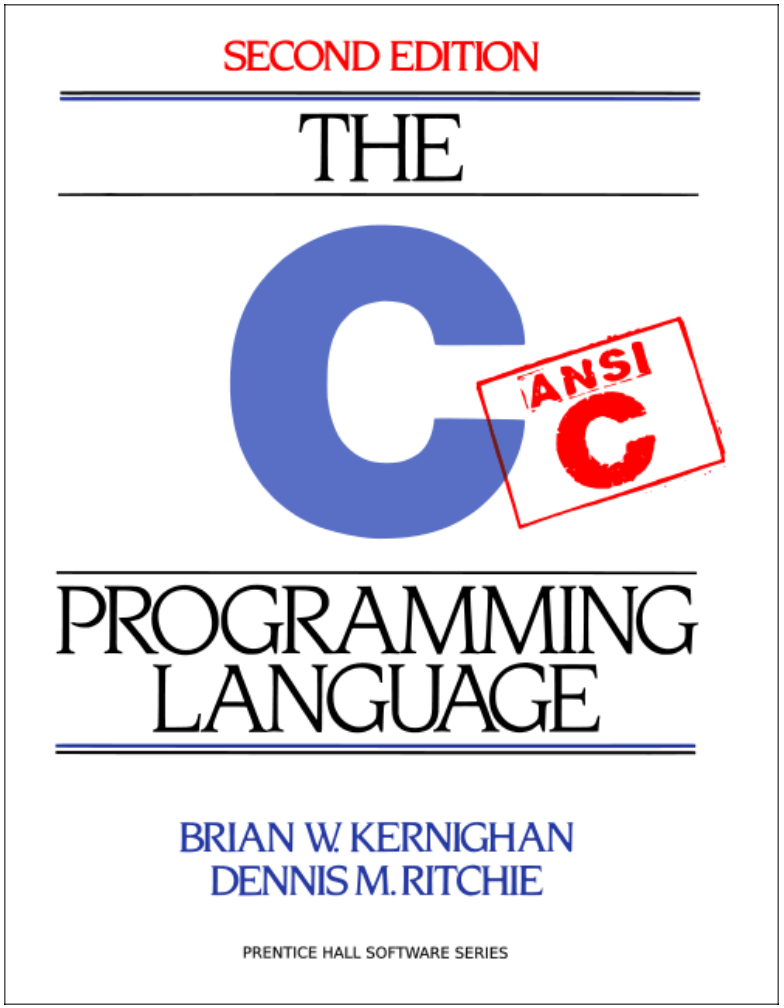
```
shell$ make
gcc -o prog prog.c
shell$ make clean
rm prog
shell$ make
gcc -o prog prog.c
shell$ make
make: Nothing to be done for `all'.
shell$
```

# Making a Makefile:
# Basic Makefile for a single .c file



# Making a Makefile:
# Basic Makefile for a single .c file & .h file






# Command Line Arguments

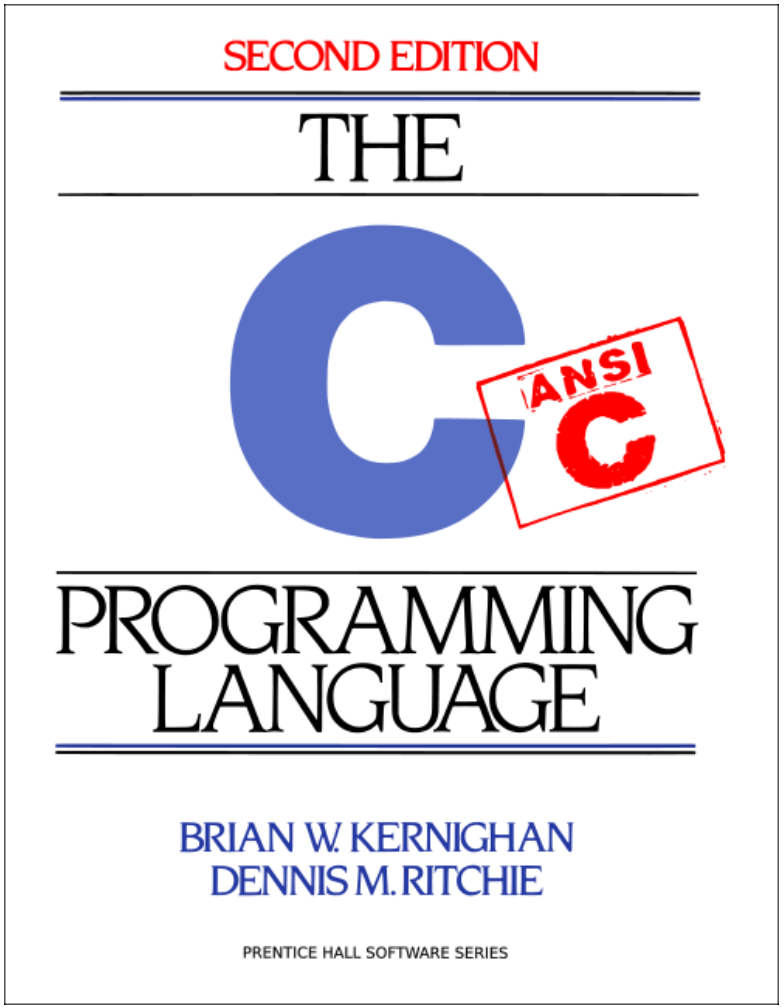
# C Programming: Command Line Arguments

Example program: hello/cli.c

- main() provides as input an array of command line arguments.
  - `int main(int argc, char **argv)`
  - `int argc`
    Number of command line arguments
  - `char **argv` –or– `char *argv[]`
    Array of pointers to strings

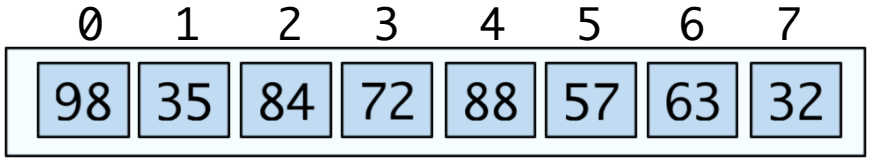
```
int main(int argc, char **argv) {
  int i;

  for( i = 0; i < argc; ++i ) {
    printf("%s -- %d\n", argv[i], i);

  }

  return 0;
}
```

```
shell$ make
gcc -o cmd_line -Wall -g -O0 cmd_line.c
shell$ ./cmd_line arg1 arg2 arg3
./cmd_line -- 0
arg1 -- 1
arg2 -- 2
arg3 -- 3
shell$
```


SECOND EDITION
THE
C
ANSI
C
PROGRAMMING
LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

# Arrays

# Arrays




- Declare an array

```
int array1[ 10 ];
int array2[] = {12, 5, 47}; // Allocate and assign values
```

~~`int array1[];`~~

```
int *array1 = NULL;
```

- Access an element of an array

```
array1[3];
```

- Assign a value to an element of an array

```
array2[1] = 17 * 13;
```

- Access the length of an array
  - You need to track this yourself in another variable.

```
int array1_len = 10, array2_len = 3;
```

# 1-Dimensional and 2-Dimensional Arrays

- Arrays are allocated as contiguous memory segments.
  - `int var[32];`
    32 element array, indexed from [0-31], single dimensional array
  - `int var[2][4];`
    8 element array, 2 rows, 4 columns : C is a row-major language
  - `var[0][1] = 12;`
    Set element in the first row, second column to 12.

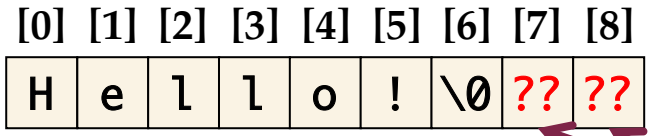
```
#include <stdio.h>
int main(int argc, char **argv) {
  int var[2][4] = { {1,2,3,4},
                    {5,6,7,8} };
  int rows, cols;

  for(rows = 0; rows < 2; rows++) {
    for(cols = 0; cols < 4; cols++) {
      printf("%d\t", var[rows][cols]);
    }
    printf("\n");
  }
  return 0;
}
```

```
shell$ gcc -o prog prog.c
shell$ ./prog
1   2   3   4
5   6   7   8
shell$
```

# Strings

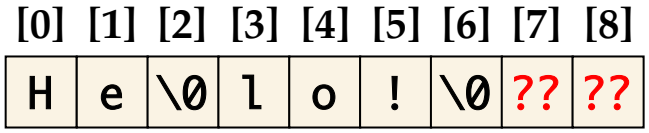
Example program: str/traverse.c

- Strings are just arrays of characters in C
  - Terminated with the **NUL** ('\0') character
  - `char str[9] = "Hello!";`
    Allocate 9 characters of space, and initialize the array. Terminate the string with the NUL character ('\0').




**Uninitialized characters = random data**

- ~~`str = "Bye";`~~
  - Cannot direct assign a quoted string to a character array outside of initialization.
- `str[2] = '\0';`




  - Moving the NUL character does not clear the rest of the string, just changes where you want it to end.

Checkout the next slides for useful library functions and pointers to examples.

# String Library

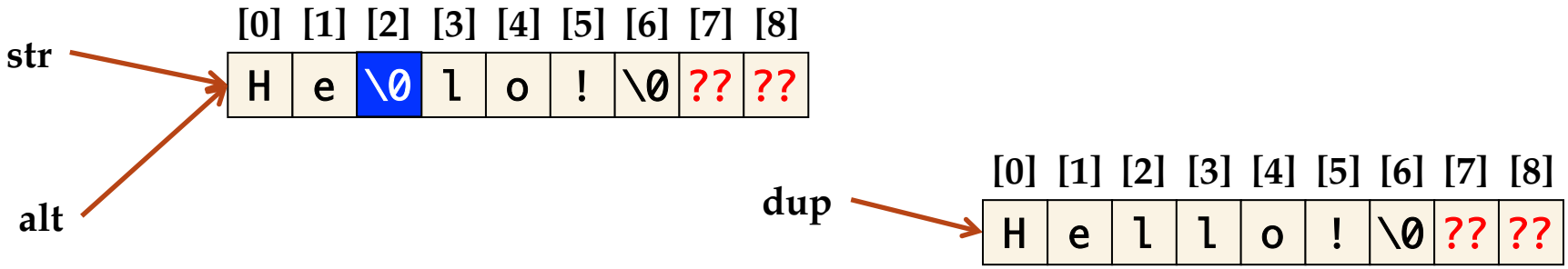
```
#include <string.h>
```

Example: strfn/dup_str.c
Useful library functions!

- `char *strdup(char *str)`
  Return a duplication of the string (allocates new memory)

```
char str[9] = "Hello!";
char *dup = NULL;
char *alt = NULL;

alt = str;
dup = strdup(str);
str[2] = '\0';

printf("String (len = %lu) \"%s\"\n", strlen(str), str);
printf("String (len = %lu) \"%s\"\n", strlen(dup), dup);
printf("String (len = %lu) \"%s\"\n", strlen(alt), alt);
```

```
shell$ ./prog
String (len = 2) "He"
String (len = 6) "Hello!"
String (len = 2) "He"
```



# String Tokenization

Example: strtok/strptr.c
Useful library functions!

strtok() will modify the input string!

- `char * strtok(char * str, char * sep)`
  - `strtok(str, " ")`
    The first call to strtok you provide the 'string' to parse.
  - `strtok(NULL, " ")`
    To keep parsing the same string, pass NULL
  - strtok() will return NULL when finished parsing

```
int main(int argc, char *argv[]) {
  char str[32] = "This is an  example string";
  char *str_ptr = NULL;
  int counter = 0;
  printf("Parsing the String: [%s]\n", str);
  for( str_ptr  = strtok(str, " ");
       NULL    != str_ptr;
       str_ptr  = strtok(NULL, " ") ) {
    printf("Index %2d) [%s]\n",
           counter, str_ptr);
    ++counter;
  }
  return 0;
}
```

This can be tricky at first, be sure you understand the example before using in your own code.

```
shell$ ./token
Parsing the String: [This is an  example string]
Index  0) [This]
Index  1) [is]
Index  2) [an]
Index  3) [example]
Index  4) [string]
shell$
```

# Input Processing Character-by-Character

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

- `int fgetc( FILE *stream )`
  Access the next character from the stream
  Returns `EOF` on end of input/file.
- System defined `FILE` streams
  - `stdout` – Standard Output
  - `stderr` – Standard Error
  - `stdin` – Standard Input

```
char value = ' ';

do {
    value = fgetc( stdin );
    if( value != EOF ) {
        printf("Character: \"%c\"\n", value);
    }
} while( value != EOF );

printf("All Done!\n");
```

```
shell$ ./prog
abc
Character: "a"
Character: "b"
Character: "c"
Character: "
"
Test!!
Character: "T"
Character: "e"
Character: "s"
Character: "t"
Character: "!"
Character: "!"
Character: "
"
All Done!
shell$
```

`CTRL-D` will send the `EOF` marker to the program

# Input Processing

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

- `char * fgets(char *buf, int size, FILE *stream )`
  Read at most "`size-1`" characters from the stream and store it in the string `buf`. Stops reading after a newline or `EOF`.
  - Returns `NULL` on error or when end-of-file occurs.

```
char buffer[64];
char *fgets_rtn = NULL;
do {
  fgets_rtn = fgets( buffer, 64, stdin );
  if( NULL != fgets_rtn) {
    printf("Input: \"%s\"\n", buffer);
  }
} while( fgets_rtn != NULL );

printf("All Done!\n");
```

```
shell$ ./prog
abc
String: "abc
"
Test!!
String: "Test!!
"
All Done!
shell$
```

# Input Processing

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

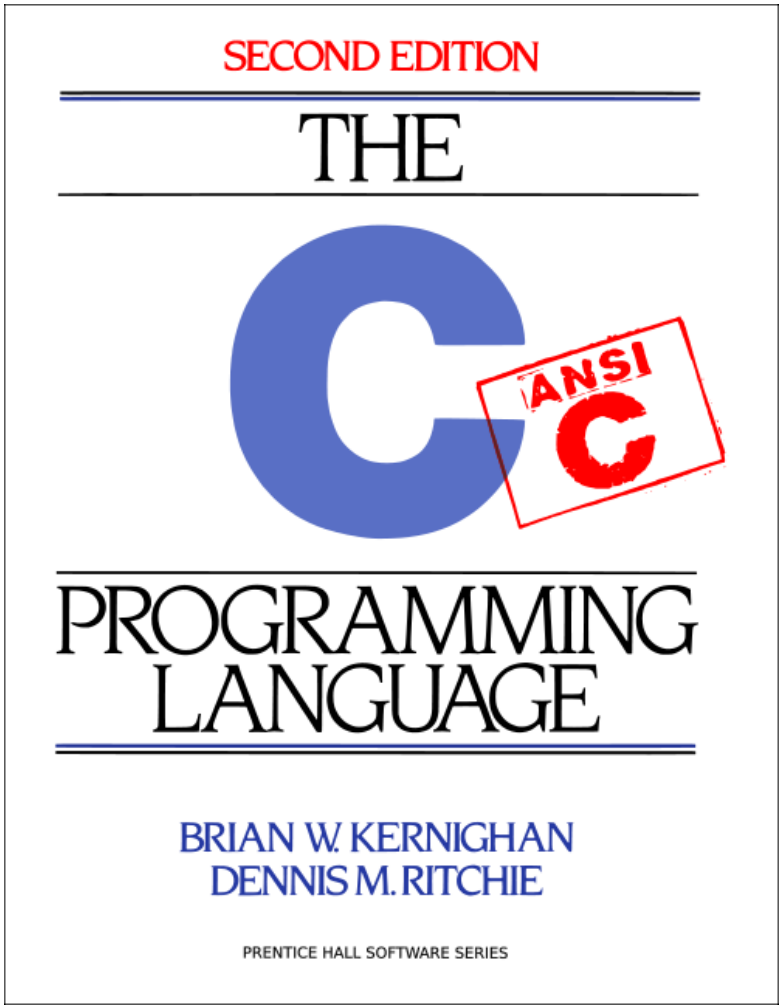
Example: fgets/extract.c

- `char * fgets(char *buf, int size, FILE *stream )`
  Read at most "`size-1`" characters from the stream and store it in the string `buf`. Stops reading after a newline or `EOF`.
  - Returns `NULL` on error or when end-of-file occurs.

```
char buffer[64];
char *fgets_rtn = NULL;
do {
  fgets_rtn = fgets( buffer, 64, stdin );
  if( NULL != fgets_rtn) {
    /* Strip off the new line */
    if( '\n' == buffer[ strlen(buffer)-1 ] ) {
      buffer[strlen(buffer)-1] = '\0';
    }

    printf("Input: \"%s\"\n", buffer);
  }
} while( fgets_rtn != NULL );
printf("All Done!\n");
```

```
shell$ ./prog
abc
String: "abc"
Test!!
String: "Test!!"
All Done!
shell$
```

What if we type in more characters than the buffer will hold?


SECOND EDITION
THE
C
ANSI
C
PROGRAMMING
LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

# Functions

# C Programming: Functions

- Functions have a return type, name, and 0 or more arguments.
  - Function declarations specifies the function signature:
    You must declare a function before you define a function.

```
void print_hello();
int sum_up_to_x(int x);
```

  - Function definitions specify the code behind the function

```
void print_hello() {
    printf("Hello World!\n");
}

int sum_up_to_x(int x) {
    int result = 0, i;
    if( x < 0 ) {
        return -1;
    }
    for( i = 0; i <= x; ++i ) {
        result += i;
    }
    return result;
}
```

**Style Req.:** Declare all functions in a header file.

# C Programming: Global and Local Scope

prog.c

```
#include "support.h"

int main(int argc, char **argv) {
    int start = 10, result;

    valid_everywhere = 10;

    result = sum_up_to_x(start);

    printf("Sum from 0 to %2d is %3d\n",
           start, result);
    return 0;
}

int sum_up_to_x(int x) {
    int result = 0, i;
    if( x < 0 ) {
        return -1;
    }
    for( i = 0; i <= x; ++i ) {
        result += i * valid_everywhere;
    }
    return result;
}
```

support.h

```
#include <stdio.h>

int valid_everywhere;

int sum_up_to_x(int x);
```

**Global scope can be accessed anywhere in the program!**

***Convention is to reduce/eliminate the use of global variables in your program.***

# C Programming: Global and Local Scope

prog.c

```
#include "support.h"

int main(int argc, char **argv) {
    int start = 10, result;

    valid_everywhere = 10;

    result = sum_up_to_x(start);

    printf("Sum from 0 to %2d is %3d\n",
           start, result);
    return 0;
}

int sum_up_to_x(int x) {
    int result = 0, i;
    if( x < 0 ) {
        return -1;
    }
    for( i = 0; i <= x; ++i ) {
        result += i * valid_everywhere;
    }
    return result;
}
```

support.h

```
#include <stdio.h>

int valid_everywhere;

int sum_up_to_x(int x);
```

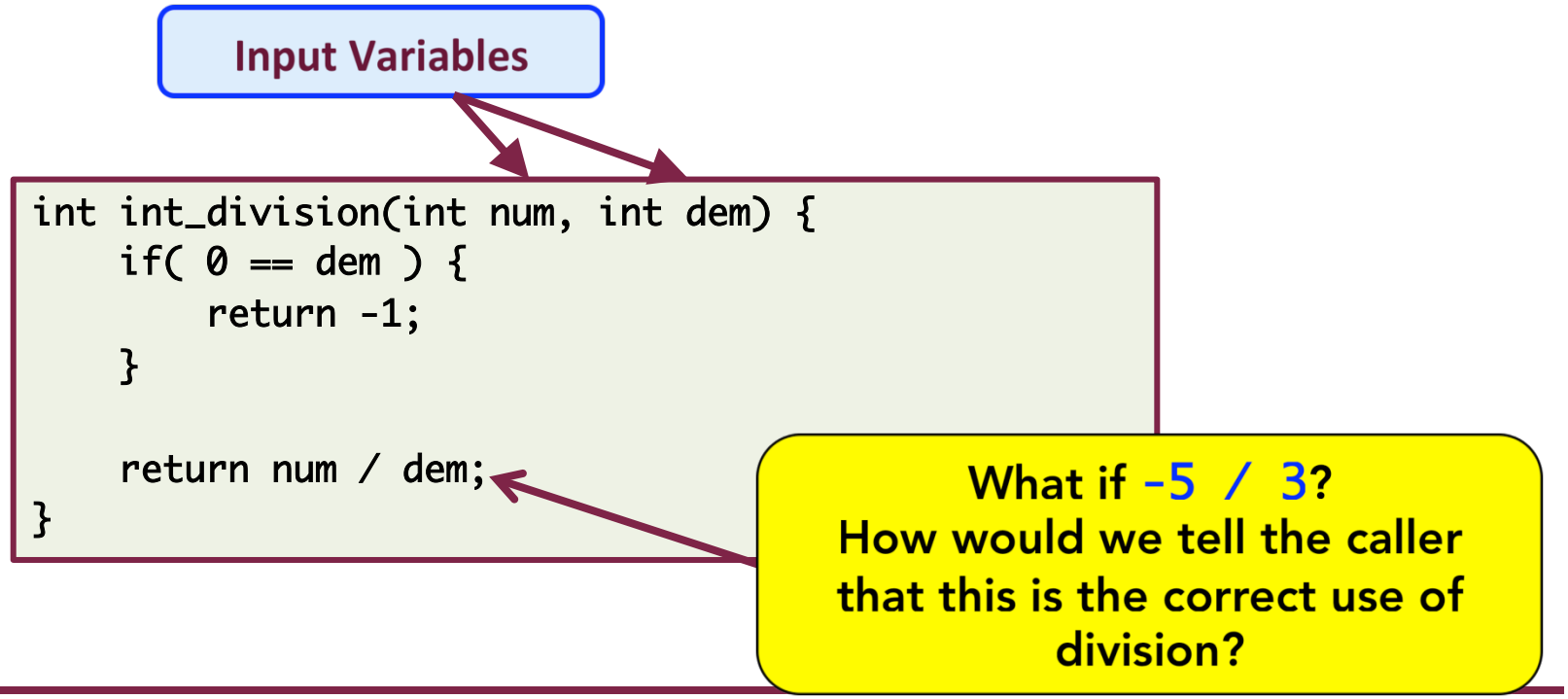
**Style Req.:**
Declare all local variables at the top of the function, and all global variables at the top of the file.

**Convention:**
***Declare all variables at the top of the function.***

# Functions

Example: divide/division.c

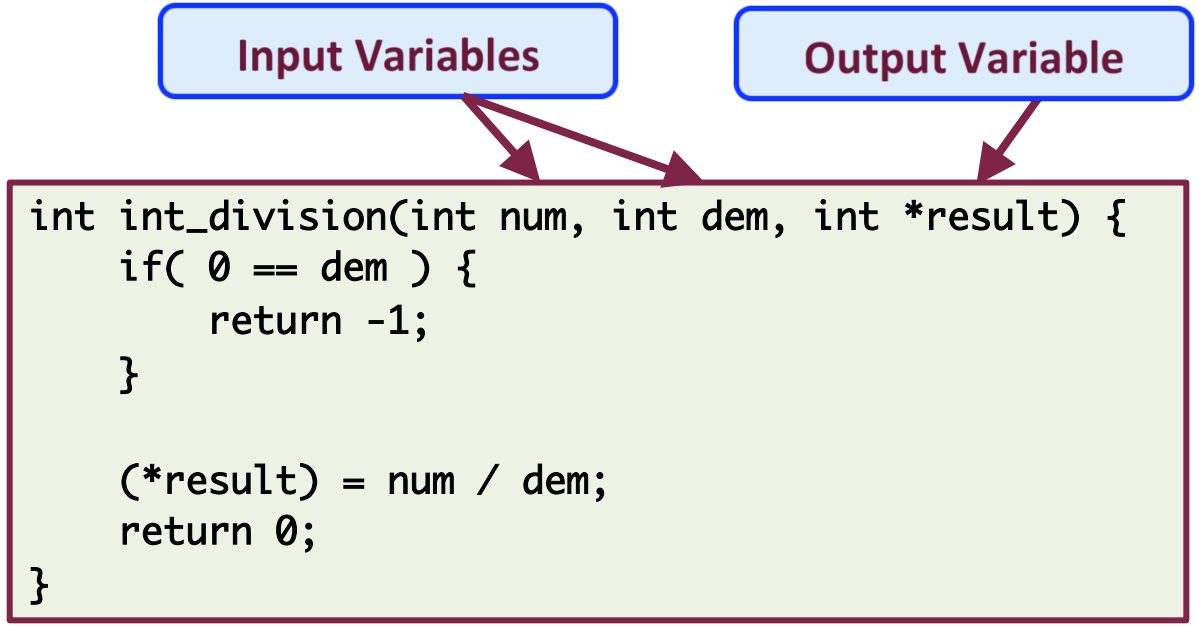
- Conventions:
  - Function and variables separated by '_' (e.g., `my_foo()` not `myFoo()`)
  - Function arguments
    First set of arguments are input variables, followed by output variables.



# Functions

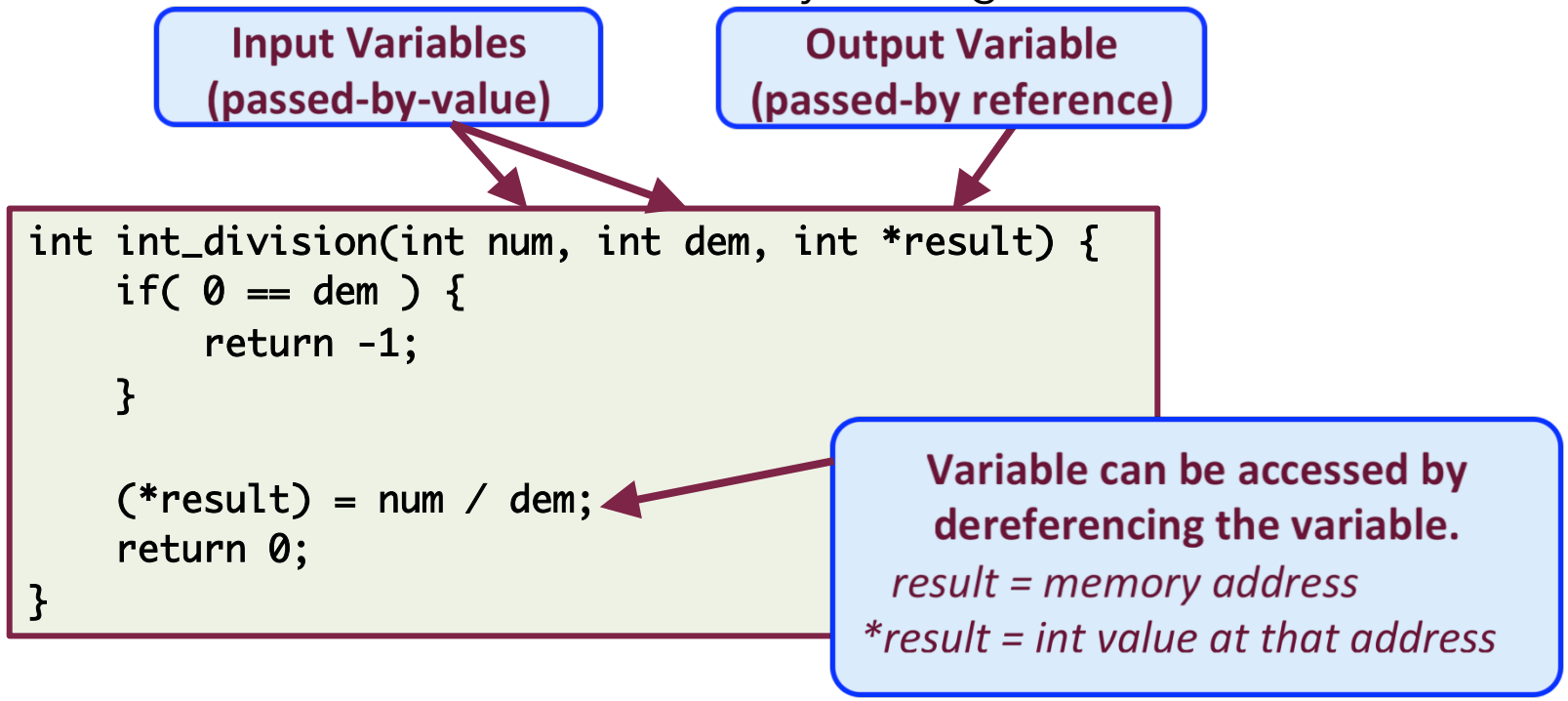
Example: divide/division.c

- Conventions:
  - Function and variables separated by '_' (e.g., `my_foo()` not `myFoo()`)
  - Function arguments
    First set of arguments are input variables, followed by output variables.
  - Functions return an `int`.
    Return `0` upon success, or a negative value indicating error conditions.



# Functions

- Pass-by-value: Variable is copied into the function.
  - A copy of the value stored in the variable is passed, not the memory.
  - All modifications to that variable are local to function.
- Pass-by-reference: A reference to the variable is passed to the function.
  - Memory address of the variable is passed, not a copy of the value.
  - Modifications to the variable reference modify the original.



# Functions

Binky pointer video
https://www.youtube.com/watch?v=i49_SNt4yfk

```
int int_division(int num, int dem, int *result);

int int_division(int num, int dem, int *result) {
    if( 0 == dem ) {
        return -1;
    }

    (*result) = num / dem;
    return 0;
}
```

**When passing-by-reference you need to pass the address of the variable.**

*result = int variable*

*&result = memory address of the variable*

**When accessing a variable passed-by-reference you need to dereference the variable.**

*result = memory address*

**result = int value at that address*

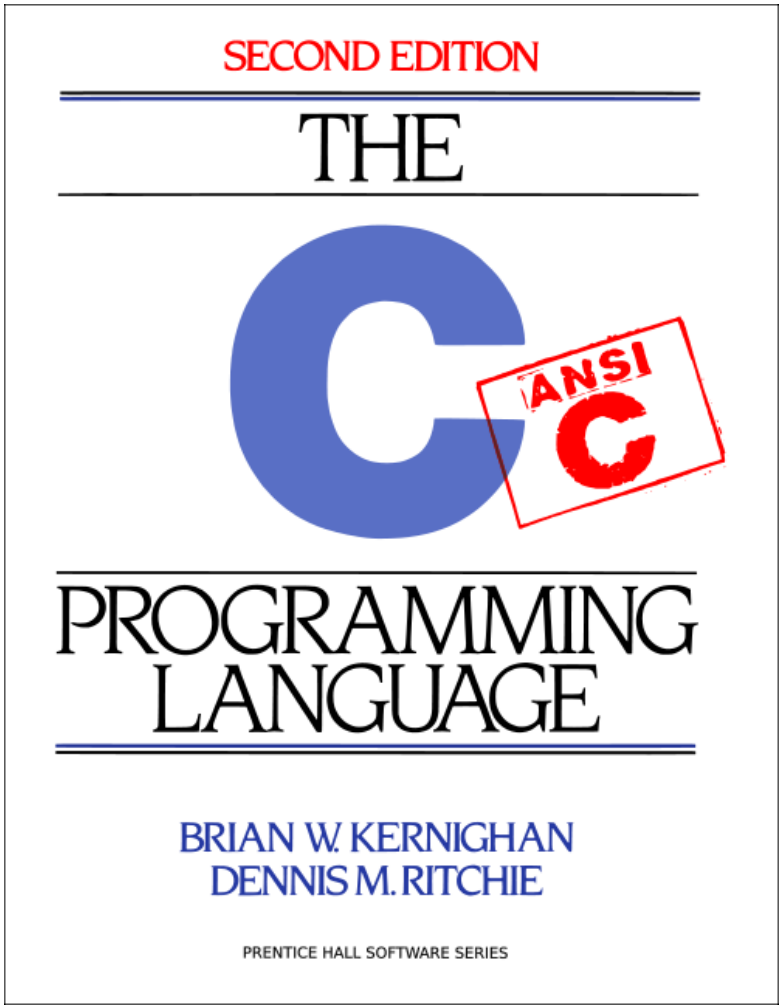
```
int main(int argc, char **argv) {
    int ret, result;
    int lhs = 10;
    int rhs = 3;

    ret = int_division(lhs, rhs, &result);
    if( ret != 0 ) {
        fprintf(stderr, "Error: Divide by 0\n");
    } else {
        printf("%3d / %3d = %3d\n",
               lhs, rhs, result);
    }
    return 0;
}
```

# Pass-by-Reference Rules of Thumb Variables

- Passing a variable by reference allows the function to change the original variable.
  - It needs to know where the original lives in memory!
- When passing the variable:
  - Add & in front of the variable to access it's memory location.
- When receiving the variable:
  - Add a * in front of the variable in the function signature
- When referencing the variable:
  - Add a * in front of the variable and surround with () to bind the * close to the variable.

```
int int_division(int num, int dem, int *result);

int main(int argc, char **argv) {
    int lhs = 10;
    int rhs = 3;
    int result;


    int_division(lhs, rhs, &result);

    return 0;
}


int int_division(int num, int dem, int *result) {
    if( 0 == dem ) {
        return -1;
    }

    (*result) = num / dem;
    return 0;
}
```

C and pointer resources:
http://cslibrary.stanford.edu/




# Arrays and Functions

Remember that a C-style string is really just an array of characters

# Passing Arrays to Functions

Examples: mem/*.c

- Arrays are passed-by-reference to functions
  - Memory address of the array is passed, not a copy of the array!
  - Any modifications to the array in the function will persist when the function returns.

```
void display(int ary[], int len);

int main(int argc, char **argv) {
    int ary3[] = {10, 20, 30};

    display(ary3, 3);
    printf("----------------------------\n");
    display(ary3, 3);

    return 0;
}
```

**Alternative Syntax:**
`void display(int *ary, int len)`

```
void display(int ary[], int len) {
    int i;
    if( ary == NULL ) {
        printf("Error: Array is NULL\n");
        return;
    }
    for(i = 0; i < len; ++i ) {
        printf("%2d = %d\n", i, ary[i]);
        ary[i] += 2;
    }
}
```

```
shell$ ./prog
 0 = 10
 1 = 20
 2 = 30
----------------------------
 0 = 12
 1 = 22
 2 = 32
```

# Pass-by-Reference Rules of Thumb Arrays

- If you want to change the contents of the array, then you do not need to explicitly pass it by reference.
  - An array is just a memory reference!
- When passing the variable:
  - Pass it as normal
    (It is already a memory address)
- When receiving the variable:
  - The * in front of the variable indicates the array
- When referencing the variable:
  - Use the [] operators instead of the * to access the values.

```
int fill_str(char *str, char c);

int main(int argc, char **argv) {
    char str[9] = "Hello!";

    printf("Before: %s\n", str);
    fill_str(str, 'a');
    printf("After : %s\n", str);

    return 0;
}


int fill_str(char *str, char c) {
    int i;

    for(i = 0; i < strlen(str); ++i) {
        str[i] = c;
    }

    return 0;
}
```

```
shell$ ./array
Before: Hello!
After : aaaaaa
```

# Returning Arrays from Functions

Examples: mem/*.c

- Returning arrays is fairly complicated

~~`int[] get_array(int len);`~~

```
int * get_array(int len);
```

This is returning a pointer to a **temporary** variable on the **stack**!

```
int* get_array(int len);

int main(int argc, char **argv) {
    int *ary1 = NULL;

    ary1 = get_array(5);
    display(ary1, 5);

    return 0;
}
```

```
shell$ ./prog
 0 = 150
 1 = 151
 2 = 152
 3 = 153
 4 = 154
```

```
int * get_array(int len) {
    int i;
    int local[len];

    for(i = 0; i < len; ++i) {
        local[i] = 100 + i + (len * 10);
    }

    return local;
}
```

```
shell$ ./prog
 0 = 150
 1 = 0
 2 = 0
 3 = 0
 4 = 4196334
```

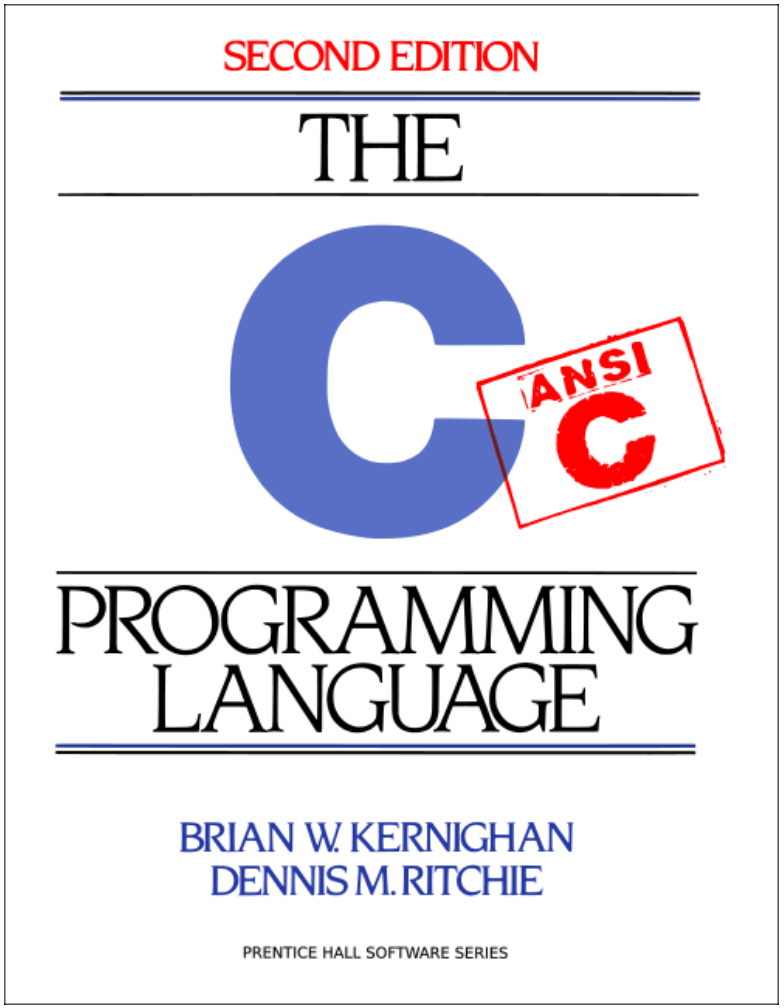
**DANGER!!!!!**

# Pass-by-Reference Rules of Thumb Arrays (Advanced)

Examples: mem/*.c

- If you want to change the size of the array or create a new one then you do need to explicitly pass it by reference.
  - You need the memory address of the array itself!
- When passing the variable:
  - Add & in front of the variable to access it's memory location.
- When receiving the variable:
  - Add an additional * in front of the variable in the function signature
- When referencing the variable:
  - Add a * in front of the variable and surround with () to bind the * close to the variable.

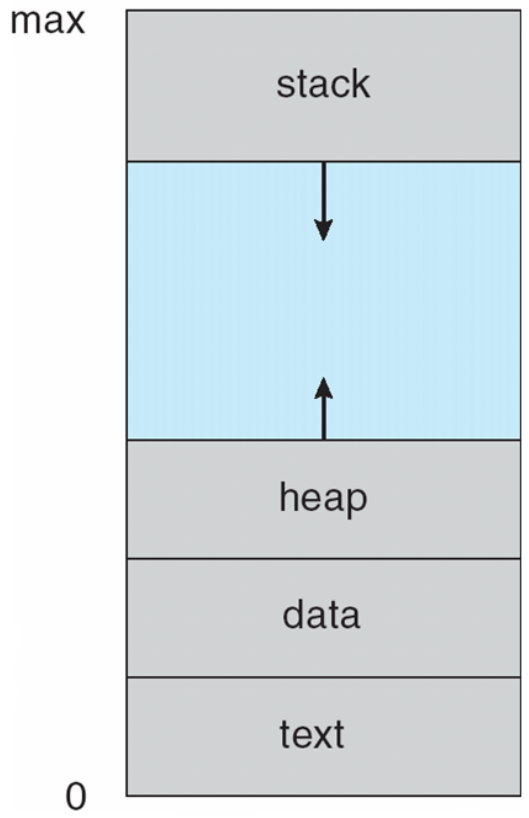
```
int dup_substr(char *src, char **dest, int pos);

int main(int argc, char **argv) {
    char *src  = "Magic";
    char *dest = "foo";
    int pos = 1;

    dup_substr(src, &dest, pos);

    return 0;
}

int dup_substr(char *src, char **dest, int pos) {
    int d_pos = 0, s_pos;

    (*dest) = strdup(src); // creates a new array
    s_pos = pos;
    while(s_pos < strlen(src) ){
        (*dest)[d_pos] = src[s_pos];
        d_pos++;
        s_pos++;
    }
    (*dest)[d_pos] = '\0';

    return 0;
}
```




# Pointers & Memory

# Program in Memory

- Composed of:
  - Program Text
  - Program Counter
    - Next instruction to execute
  - Data Section
    - Storage of global variables
  - Stack
    - Function parameters, local variables
  - Heap
    - Dynamically allocated memory



# String Library

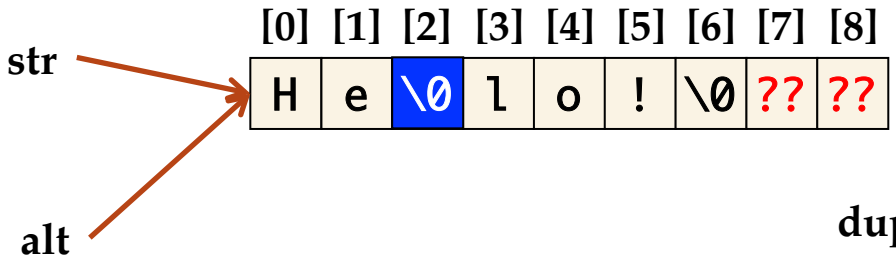
```
#include <string.h>
```

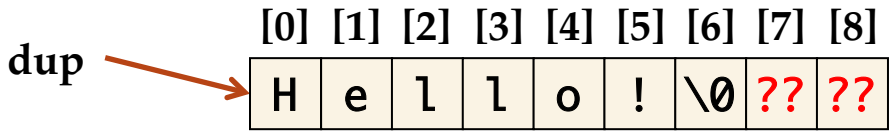
Allocates space on the **heap**, but only strings…

- `char *strdup(char *str)`
  Return a duplication of the string (allocates new memory)

```
char str[9] = "Hello!";
char *dup = NULL;
char *alt = NULL;

alt = str;
dup = strdup(str);
str[2] = '\0';

printf("String (len = %lu) \"%s\"\n", strlen(str), str);
printf("String (len = %lu) \"%s\"\n", strlen(dup), dup);
printf("String (len = %lu) \"%s\"\n", strlen(alt), alt);
```

```
shell$ ./prog
String (len = 2) "He"
String (len = 6) "Hello!"
String (len = 2) "He"
```



# Dynamic Memory Allocation

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

- `void * malloc(size_t size)`
  Allocates `size` bytes and returns a pointer to the allocated memory
- `size_t sizeof( datatype )`
  Returns the size of the specified datatype, measured in bytes
- `void free( void * ptr )`
  Free the memory space pointed to by `ptr` (must have been previously allocated by `malloc`)

```
int *ary = NULL;
int len = 10;

ary = (int*) malloc( sizeof(int) * len );
if( NULL == ary ) {
    return -1;
}
// ... Use the 10 element array as normal ...

free(ary);
ary = NULL;
```

# Dynamic Memory Allocation

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

- `void * malloc(size_t size)`
  Allocates `size` bytes and returns a pointer to the allocated memory
- `void * realloc(void *ptr, size_t size)`
  Change the size of the memory pointed to by `ptr` to `size` bytes. The contents of the memory are unchanged. Returns a pointer to the new location in memory.
  - If `ptr` is `NULL`, then it behaves the same as `malloc`.

```
double *ary = NULL;
int i, len = 0;

for(i = 0; i < 10; ++i ) {
  // Extend the array
  ary = (double*)realloc( ary, sizeof(double) * (len+1) );
  if( NULL == ary ) {
    exit(-1);
  }
  // Save the value
  ary[len] = i * 3.14;
  len++;
}
```

# Returning Arrays from Functions

```
int* get_array(int len);

int main(int argc, char **argv) {
    int *ary1 = NULL;

    ary1 = get_array(5);
    display(ary1, 5);

    return 0;
}
```

```
shell$ ./prog
 0 = 150
 1 = 0
 2 = 0
 3 = 0
 4 = 4196334
```

```
int * get_array(int len) {
    int i;
    int local[len];

    for(i = 0; i < len; ++i) {
        local[i] = 100 + i + (len * 10);
    }

    return local;
}
```

This is returning a pointer to a **temporary** variable on the **stack**!

# Returning Arrays from Functions

```
int* get_array(int len);

int main(int argc, char **argv) {
    int *ary1 = NULL;

    ary1 = get_array(5);
    display(ary1, 5);

    if( NULL != ary1 ) {
        free(ary1)
        ary1 = NULL;
    }
    return 0;
}
```

We take the responsibility of freeing that memory when we are done!

```
shell$ ./prog
 0 = 150
 1 = 151
 2 = 152
 3 = 153
 4 = 154
```

```
int * get_array(int len) {
    int i;
    int *local = NULL;

    local = (int*)malloc( sizeof(int) * len);
    if( NULL == local ) {
        return NULL;
    }

    for(i = 0; i < len; ++i) {
        local[i] = 100 + i + (len * 10);
    }

    return local;
}
```

This is returning a pointer to a variable on the **heap**!

# Passing the Array by Reference

```
int get_array(int len, int **ary);

int main(int argc, char **argv) {
    int *ary1 = NULL;

    get_array(5, &ary1);
    display(ary1, 5);

    if( NULL != ary1 ) {
        free(ary1)
        ary1 = NULL;
    }
    return 0;
}
```

```
shell$ ./prog
 0 = 150
 1 = 151
 2 = 152
 3 = 153
 4 = 154
```

```
int get_array(int len, int **ary) {
    int i;

    (*ary) = (int*)malloc(sizeof(int) * len);
    if( NULL == (*ary) ) {
        // Print an error
        return -1;
    }

    for(i = 0; i < len; ++i) {
        (*ary)[i] = 100 + i + (len * 10);
    }

    return 0;
}
```

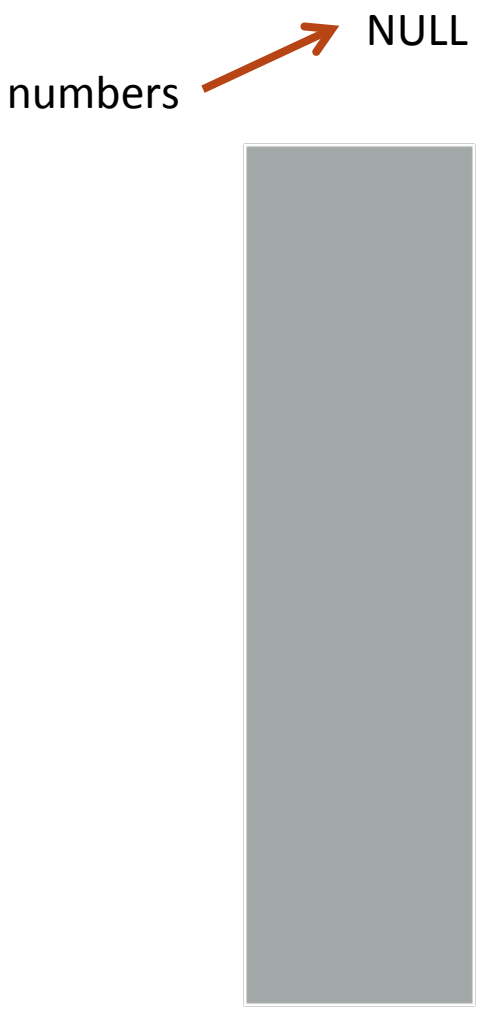
# Memory Management

```
int main(int argc, char **argv) {
  int *numbers;









  return 0;
}
```

# Memory Management

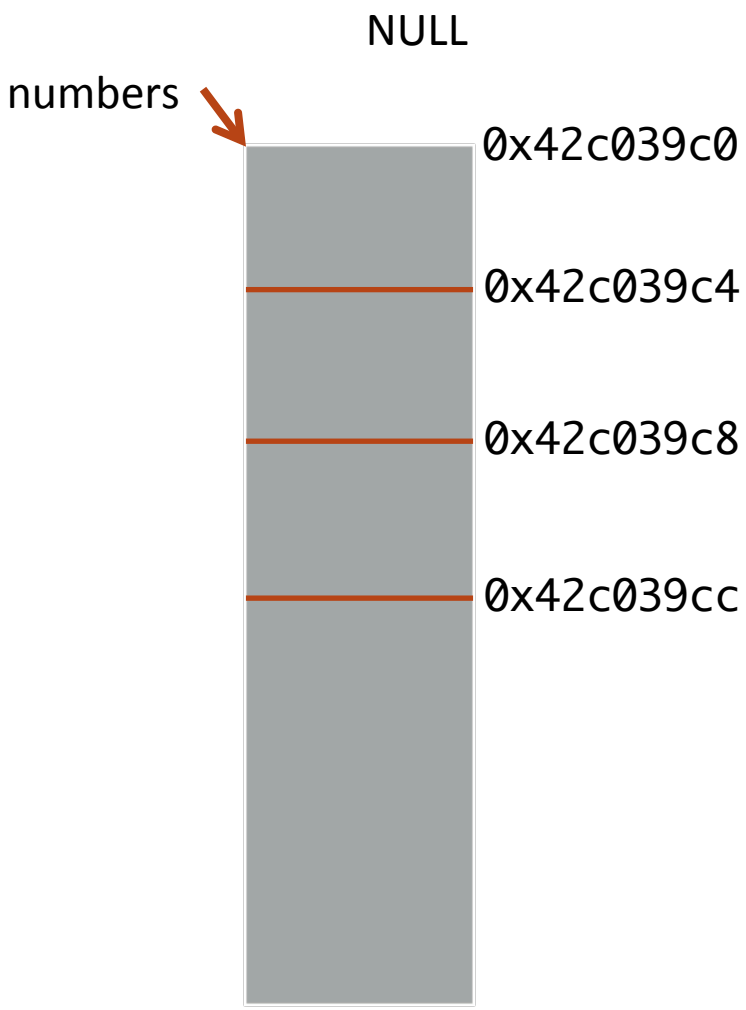
```
int main(int argc, char **argv) {
  int *numbers = NULL;




  return 0;
}
```

**Safety Notice:**
Initialize all pointers to NULL when you declare them!



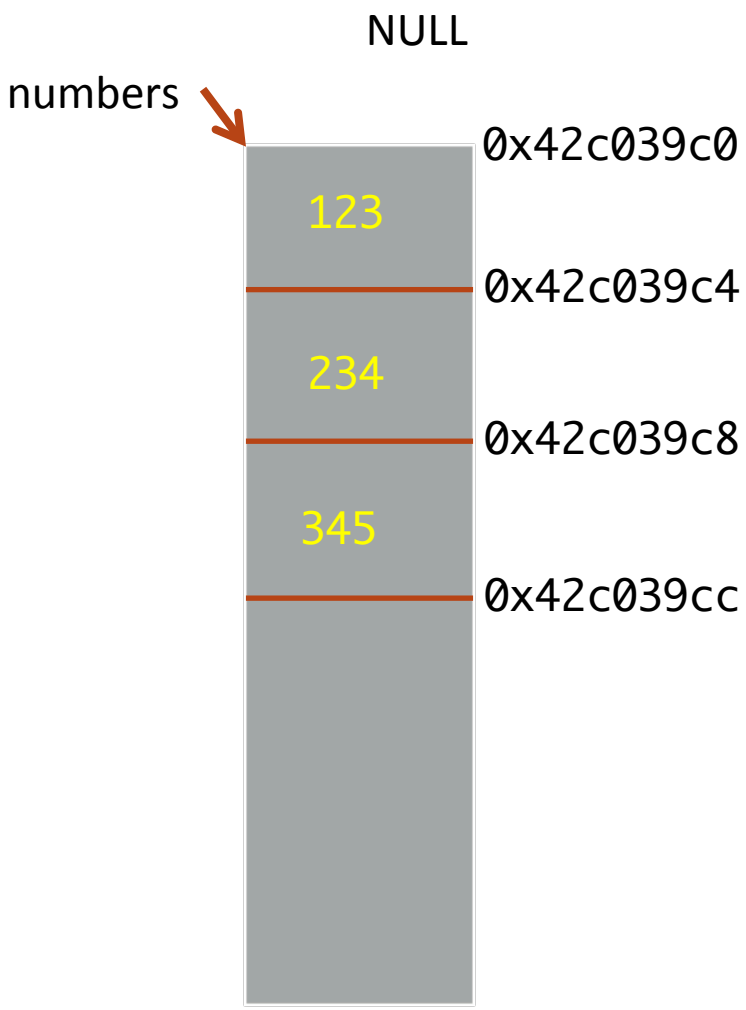
# Memory Management

```
int main(int argc, char **argv) {
  int *numbers = NULL;

  numbers = (int *) malloc( sizeof(int) * 4 );
  if( NULL == numbers ) {
    fprintf(stderr, "Memory Error!\n");
    exit(-1);
  }









  return 0;
}
```



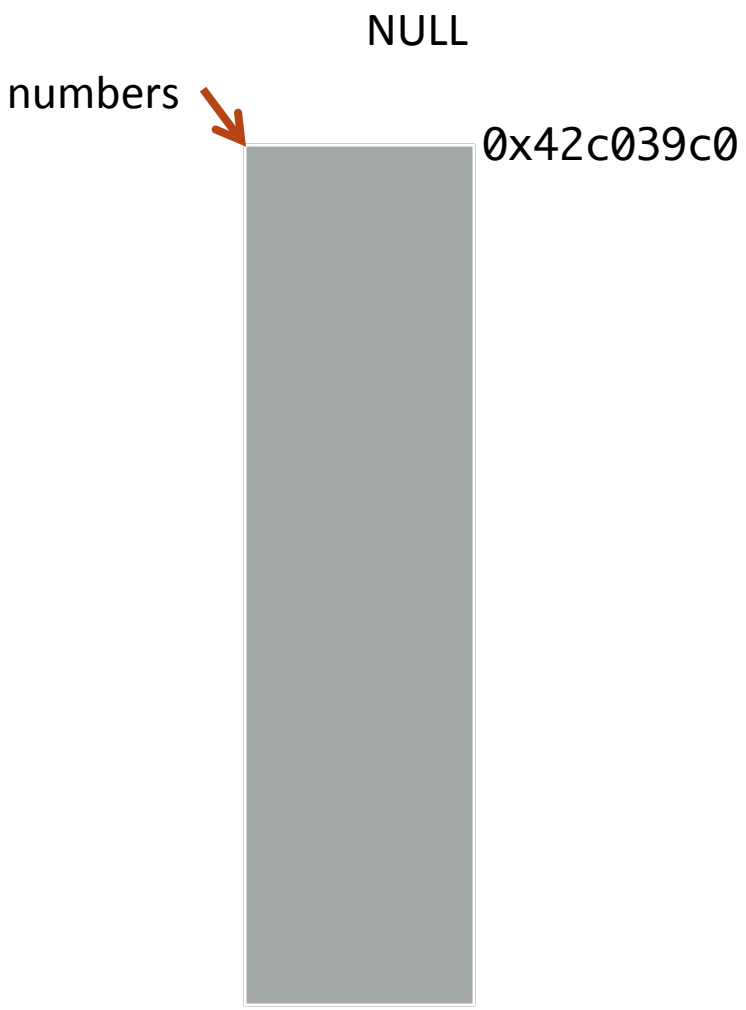
# Memory Management

```
int main(int argc, char **argv) {
  int *numbers = NULL;

  numbers = (int *) malloc( sizeof(int) * 4 );
  if( NULL == numbers ) {
    fprintf(stderr, "Memory Error!\n");
    exit(-1);
  }

  numbers[0] = 123;
  numbers[1] = 234;
  numbers[2] = 345;









  return 0;
}
```



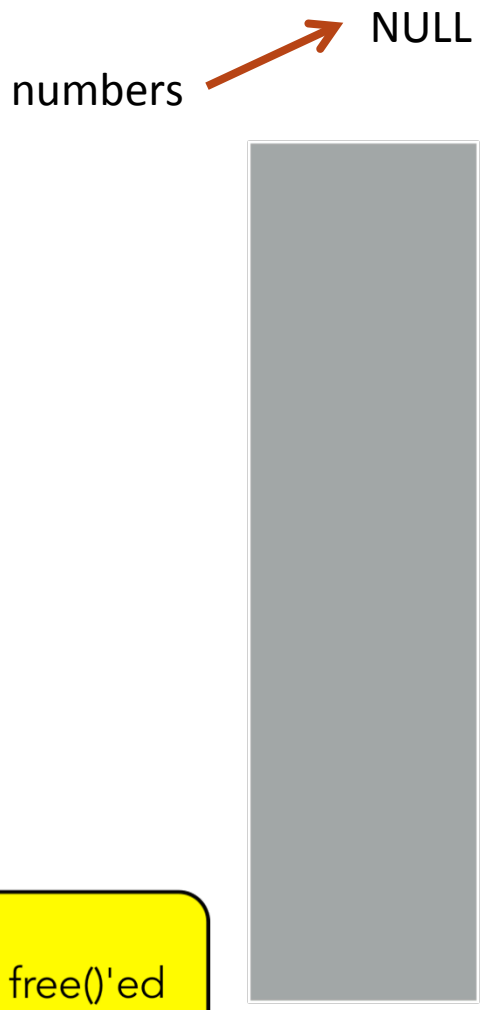
# Memory Management

```
int main(int argc, char **argv) {
  int *numbers = NULL;

  numbers = (int *) malloc( sizeof(int) * 4 );
  if( NULL == numbers ) {
    fprintf(stderr, "Memory Error!\n");
    exit(-1);
  }

  numbers[0] = 123;
  numbers[1] = 234;
  numbers[2] = 345;

  if( NULL != numbers ) {
    free(numbers);

  }

  return 0;
}
```



# Memory Management

```
int main(int argc, char **argv) {
  int *numbers = NULL;

  numbers = (int *) malloc( sizeof(int) * 4 );
  if( NULL == numbers ) {
    fprintf(stderr, "Memory Error!\n");
    exit(-1);
  }

  numbers[0] = 123;
  numbers[1] = 234;
  numbers[2] = 345;

  if( NULL != numbers ) {
    free(numbers);
    numbers = NULL;
  }

  return 0;
}
```




**Safety Notice:**
After the memory has been free()'ed set the pointer back to NULL!

# C Programming: fgets & Memory Management

```
int main(int argc, char **argv) {
    char *buffer = NULL, *fgets_rtn = NULL;
    // Allocate the Buffer
    buffer = (char*) malloc( sizeof(char) * 32 );
    if( NULL == buffer ) {
        fprintf(stderr, "Error: Cannot allocate memory!\n");
        exit(-1);
    }
    // Read input
    fgets_rtn = fgets(buffer, 32, stdin);
    if( NULL == fgets_rtn) {
        return 0;
    }
    /* Strip off the new line */
    if( '\n' == buffer[ strlen(buffer) - 1] ) {
        buffer[ strlen(buffer) - 1] = '\0';
    }
    printf("Input: %s\n", buffer);
    if( NULL != buffer ) {
        free(buffer);
        buffer = NULL;
    }
    return 0;
}
```

```
shell$ make
gcc -o prog prog.c
shell$ ./prog
Hello World!
Input: Hello World!
shell$
```

# Dynamic Memory Allocation Safety Tips

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

- When declaring a pointer, always set it to `NULL`
- After calling `malloc`, always check for `NULL`
  - Indicates out-of-memory
  - Often indicates bugs in your program (resulting in bad parameters)
- When finished using the memory always call `free`
  - After calling `free` set the pointer back to `NULL`

```
int *ary = NULL;
int len = 10;

ary = (int*) malloc( sizeof(int) * len );
if( NULL == ary ) {
    return -1;
}

// Use the 10 element array as normal...

if( NULL != ary ) {
    free(ary);
    ary = NULL;
}
```

# Returning Arrays from Functions

Examples: mem/*.c

- Returning arrays is fairly complicated

~~`int[] get_array(int len);`~~

```
int * get_array(int len);
```

This is returning a pointer to a **temporary** variable on the **stack**!

```
int* get_array(int len);

int main(int argc, char **argv) {
    int *ary1 = NULL;

    ary1 = get_array(5);
    display(ary1, 5);

    return 0;
}
```

```
int * get_array(int len) {
    int i;
    int local[len];

    for(i = 0; i < len; ++i) {
        local[i] = 100 + i + (len * 10);
    }

    return local;
}
```

```
shell$ ./prog
 0 = 150
 1 = 151
 2 = 152
 3 = 153
 4 = 154
```

```
shell$ ./prog
 0 = 150
 1 = 0
 2 = 0
 3 = 0
 4 = 4196334
```

**DANGER!!!!!**

# Returning Arrays from Functions

```
int* get_array(int len);

int main(int argc, char **argv) {
    int *ary1 = NULL;

    ary1 = get_array(5);
    display(ary1, 5);

    if( NULL != ary1 ) {
        free(ary1)
        ary1 = NULL;
    }
    return 0;
}
```

We take the responsibility of freeing that memory when we are done!

```
shell$ ./prog
 0 = 150
 1 = 151
 2 = 152
 3 = 153
 4 = 154
```

```
int * get_array(int len) {
    int i;
    int *local = NULL;

    local = (int*)malloc( sizeof(int) * len);
    if( NULL == local ) {
        return NULL;
    }

    for(i = 0; i < len; ++i) {
        local[i] = 100 + i + (len * 10);
    }

    return local;
}
```

This is returning a pointer to a variable on the **heap**!

# Passing the Array by Reference

```
int get_array(int len, int **ary);

int main(int argc, char **argv) {
    int *ary1 = NULL;

    get_array(5, &ary1);
    display(ary1, 5);

    if( NULL != ary1 ) {
        free(ary1)
        ary1 = NULL;
    }
    return 0;
}
```

```
shell$ ./prog
 0 = 150
 1 = 151
 2 = 152
 3 = 153
 4 = 154
```

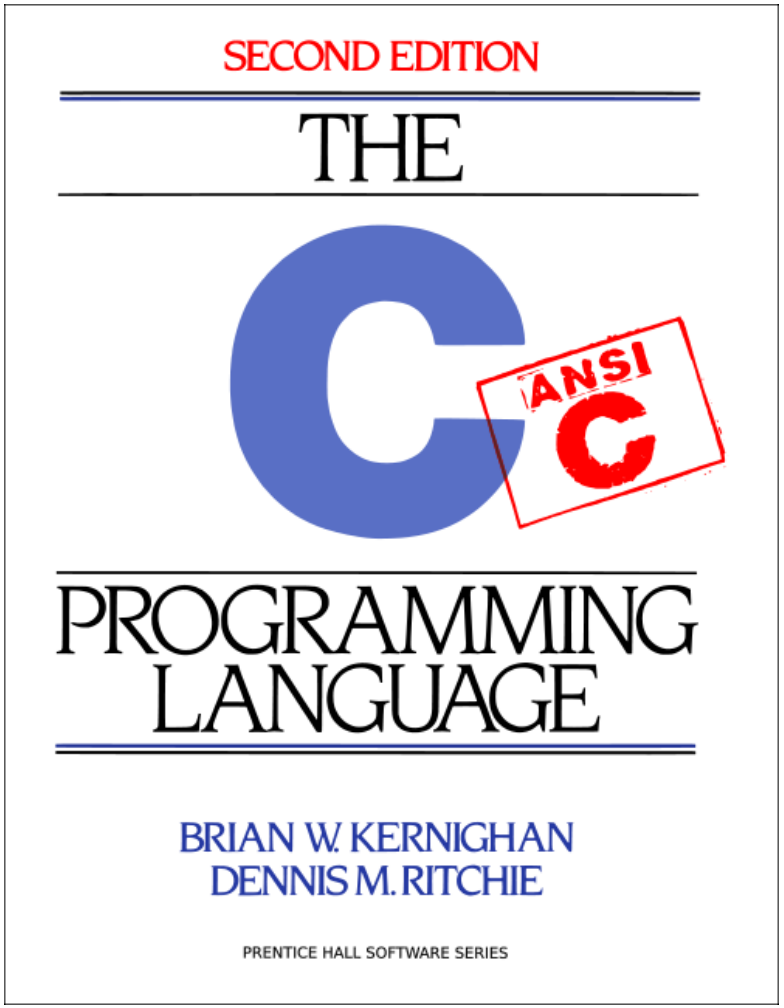
```
int get_array(int len, int **ary) {
    int i;

    (*ary) = (int*)malloc(sizeof(int) * len);
    if( NULL == (*ary) ) {
        // Print an error
        return -1;
    }

    for(i = 0; i < len; ++i) {
        (*ary)[i] = 100 + i + (len * 10);
    }

    return 0;
}
```

# Pass-by-Reference Rules of Thumb Arrays (Advanced)

Examples: mem/*.c

- If you want to change the size of the array or create a new one then you do need to explicitly pass it by reference.
  - You need the memory address of the array itself!
- When passing the variable:
  - Add & in front of the variable to access it's memory location.
- When receiving the variable:
  - Add an additional * in front of the variable in the function signature
- When referencing the variable:
  - Add a * in front of the variable and surround with () to bind the * close to the variable.

```
int dup_substr(char *src, char **dest, int pos);

int main(int argc, char **argv) {
    char *src  = "Magic";
    char *dest = "foo";
    int pos = 1;

    dup_substr(src, &dest, pos);

    return 0;
}

int dup_substr(char *src, char **dest, int pos) {
    int d_pos = 0, s_pos;

    (*dest) = strdup(src); // creates a new array
    s_pos = pos;
    while(s_pos < strlen(src) ){
        (*dest)[d_pos] = src[s_pos];
        d_pos++;
        s_pos++;
    }
    (*dest)[d_pos] = '\0';

    return 0;
}
```




# struct, enum, & union

# Structures

Examples: struct/*.c

- A structure is a collection of one or more other variables with possibly different types.

```
#include <stdio.h>

struct point_t {
    double x;
    double y;
};
typedef struct point_t point_t;

int main(int argc, char **argv) {
    point_t pt;

    pt.x = 10;
    pt.y = 7;

    printf("Point: (%6.2f, %6.2f)\n", pt.x, pt.y);

    return 0;
}
```

Structure containing two `double` variables

Create an alias for `struct point_t` to just `point_t`

The dot operator (`.`) allows you to access a field of the structure.

```
shell$ ./example
Point: ( 10.00,   7.00)
```

# Structures – Passed-by-reference

Examples: struct/*.c

```c
#include <stdio.h>

struct point_t {
    double x;
    double y;
};
typedef struct point_t point_t;

int create_point(double x, double y, point_t *pt);

int main(int argc, char *argv[]) {
    point_t pt;

    create_point(10, 7, &pt);

    return 0;
}

int create_point(double x, double y, point_t *pt) {

    (*pt).x = x;
    pt->y = y;

    return 0;
}
```

Step 2: Add a *

Step 1: Add an &

Step 2: Add a *

Step 3: Add parenthesis and * when you want to access the variable behind the reference

Structures allow you to use the alternative arrow notation for pointers to structures

# Structures of Structures

Examples: struct/*.c

```
struct point_t {
    double x;
    double y;
};
typedef struct point_t point_t;

struct rectangle_t {
    double width;
    double height;
    point_t upper_left;
    point_t lower_right;
};
typedef struct rectangle_t rectangle_t;
```

Must declare the structure before using it.

# Enumerated Types

- A named or unnamed collection of constant integer variables sometimes with predefined values.
  - If no value is assigned then one is added to each consecutive variable defined.

```
enum {
    OP_ADDITION,
    OP_SUBTRACTION,
    OP_MULTIPLICATION,
    OP_DIVISION,
    OP_MODULUS,
    OP_ERROR
};
```

**Unnamed**
Can only use the constants defined

```
enum op_type_t {
    OP_ADDITION,
    OP_SUBTRACTION,
    OP_MULTIPLICATION,
    OP_DIVISION,
    OP_MODULUS,
    OP_ERROR
};
typedef enum op_type_t op_type_t;
```

**Named**
Can use the constants defined, and the datatype

# Unnamed Enumerated Types

```
enum {
    OP_ADDITION,
    OP_SUBTRACTION,
    OP_MULTIPLICATION,
    OP_DIVISION,
    OP_MODULUS,
    OP_ERROR
};

int main(int argc, char *argv[]) {
    int alt;

    alt = OP_SUBTRACTION;

    printf("Alt = %d\n", alt);
    printf("---------\n");
    printf("ADD = %d\n", OP_ADDITION);
    printf("SUB = %d\n", OP_SUBTRACTION);
    printf("MUL = %d\n", OP_MULTIPLICATION);
    printf("DIV = %d\n", OP_DIVISION);
    printf("MOD = %d\n", OP_MODULUS);
    printf("ERR = %d\n", OP_ERROR);

    return 0;
}
```

Unless otherwise specified, the first constant is 0, and each consecutive variable is one more than the previous.

```
shell$ ./example
Alt = 1
---------
ADD = 0
SUB = 1
MUL = 2
DIV = 3
MOD = 4
ERR = 5
```

# Unnamed Enumerated Types

```
enum {
    OP_ADDITION,
    OP_SUBTRACTION,
    OP_MULTIPLICATION = 10,
    OP_DIVISION,
    OP_MODULUS,
    OP_ERROR
};

int main(int argc, char *argv[]) {
    int alt;

    alt = OP_SUBTRACTION;

    printf("Alt = %d\n", alt);
    printf("---------\n");
    printf("ADD = %d\n", OP_ADDITION);
    printf("SUB = %d\n", OP_SUBTRACTION);
    printf("MUL = %d\n", OP_MULTIPLICATION);
    printf("DIV = %d\n", OP_DIVISION);
    printf("MOD = %d\n", OP_MODULUS);
    printf("ERR = %d\n", OP_ERROR);

    return 0;
}
```

You can specify a value for any of the variables.

```
shell$ ./example
Alt = 1
---------
ADD = 0
SUB = 1
MUL = 10
DIV = 11
MOD = 12
ERR = 13
```

It just resets the counter.

# Unnamed Enumerated Types

```
enum {
    OP_ADDITION,
    OP_SUBTRACTION,
    OP_MULTIPLICATION = 10,
    OP_DIVISION,
    OP_MODULUS,
    OP_ERROR = -1
};

int main(int argc, char *argv[]) {
    int alt;

    alt = OP_SUBTRACTION;

    printf("Alt = %d\n", alt);
    printf("---------\n");
    printf("ADD = %d\n", OP_ADDITION);
    printf("SUB = %d\n", OP_SUBTRACTION);
    printf("MUL = %d\n", OP_MULTIPLICATION);
    printf("DIV = %d\n", OP_DIVISION);
    printf("MOD = %d\n", OP_MODULUS);
    printf("ERR = %d\n", OP_ERROR);

    return 0;
}
```

The value can be negative

```
shell$ ./example
Alt = 1
---------
ADD = 0
SUB = 1
MUL = 10
DIV = 11
MOD = 12
ERR = -1
```

# Unnamed Enumerated Types

```
enum {
    OP_ADDITION,
    OP_SUBTRACTION,
    OP_MULTIPLICATION = 10,
    OP_DIVISION,
    OP_MODULUS = 10,
    OP_ERROR = -1
};

int main(int argc, char *argv[]) {
    int alt;

    alt = OP_SUBTRACTION;

    printf("Alt = %d\n", alt);
    printf("---------\n");
    printf("ADD = %d\n", OP_ADDITION);
    printf("SUB = %d\n", OP_SUBTRACTION);
    printf("MUL = %d\n", OP_MULTIPLICATION);
    printf("DIV = %d\n", OP_DIVISION);
    printf("MOD = %d\n", OP_MODULUS);
    printf("ERR = %d\n", OP_ERROR);

    return 0;
}
```

The value can repeat

```
shell$ ./example
Alt = 1
---------
ADD = 0
SUB = 1
MUL = 10
DIV = 11
MOD = 10
ERR = -1
```

# Enumerated Types

- A named or unnamed collection of constant integer variables sometimes with predefined values.
  - If no value is assigned then one is added to each consecutive variable defined.

```
enum {
    OP_ADDITION,
    OP_SUBTRACTION,
    OP_MULTIPLICATION,
    OP_DIVISION,
    OP_MODULUS,
    OP_ERROR
};
```

**Unnamed**
Can only use the constants defined

```
enum op_type_t {
    OP_ADDITION,
    OP_SUBTRACTION,
    OP_MULTIPLICATION,
    OP_DIVISION,
    OP_MODULUS,
    OP_ERROR
};
typedef enum op_type_t op_type_t;
```

**Named**
Can use the constants defined, and the datatype

# Named Enumerated Types

```
enum op_type_t {
    OP_ADDITION,
    OP_SUBTRACTION,
    OP_MULTIPLICATION,
    OP_DIVISION,
    OP_MODULUS,
    OP_ERROR = -1
};
typedef enum op_type_t op_type_t;

int main(int argc, char *argv[]) {
    op_type_t op;
    int alt;

    op = OP_DIVISION;
    alt = OP_SUBTRACTION;
    printf("Op  = %d\n", op);
    printf("Alt = %d\n", alt);
    printf("---------\n");
    printf("ADD = %d\n", OP_ADDITION);
    printf("SUB = %d\n", OP_SUBTRACTION);
    printf("MUL = %d\n", OP_MULTIPLICATION);
    printf("DIV = %d\n", OP_DIVISION);
    printf("MOD = %d\n", OP_MODULUS);
    printf("ERR = %d\n", OP_ERROR);
    return 0;
}
```

The enum type name can be used instead of 'int'

```
shell$ ./example
Op  = 3
Alt = 1
---------
ADD = 0
SUB = 1
MUL = 2
DIV = 3
MOD = 4
ERR = -1
```

# Named Enumerated Types

```
enum op_type_t {
    OP_ADDITION,
    OP_SUBTRACTION,
    OP_MULTIPLICATION,
    OP_DIVISION,
    OP_MODULUS,
    OP_ERROR = -1
};
typedef enum op_type_t op_type_t;

void display_op(op_type_t op);

int main(int argc, char *argv[]) {
    op_type_t op;
    int alt;

    op = OP_DIVISION;
    alt = OP_SUBTRACTION;

    op = 0;
    display_op(OP_ADDITION);
    display_op(alt);
    display_op(op);
    display_op(-10);

    return 0;
}
```

- Integers and enum values can be mixed freely.
- Named enumerated types provide a clear meaning to the caller of a function regarding the expected range of values for a parameter.
  - `void display_op(op_type_t op);`
  - Compiler will not check this!

```
shell$ ./example
Addition
Subtraction
Addition
Unknown
```

# C Programming: File I/O

Examples: files/*.c

- Interacting with an external file
  - `FILE *fd fopen(char * filename, char * mode)`
    Open the filename according to the 'mode' and return a pointer to the file descriptor.

| Modes | |
|---|---|
| `"r"` | Reading |
| `"w"` | Writing<br>Truncate file, if exists |
| `"r+"`<br>`"w+"` | Reading & Writing |
| `"a"` | Append to a file |

  - `fclose(FILE *fd)`
    Close the file pointed to by the file descriptor 'fd'.
  - `feof(FILE *fd)`
    Return 0 if not at the end of the file pointed to by 'fd'.

# Formatted Output in C

Examples: files/*.c

- To **write** a value to a stream:

```
int fprintf(FILE *stream, char *format, <args>);
```

| Special Characters | |
|---|---|
| \n | newline |
| \t | tab |
| \\ | \ |
| \” | “ |
| \r | Carriage return |
| \f | Form Feed |

| Format | |
|---|---|
| %d | integer |
| %f | float, double |
| %c | char |
| %s | char * (String) |

| Precision ( _ = space) | |
|---|---|
| %3d | ___<br>_12 |
| %-3d | ___<br>12_ |
| %6.2f | ___.__<br>__1.00<br>_13.14 |

- Predefined file streams
  - stdout OUT Standard Output
  - stderr OUT Standard Error
  - stdin IN Standard In

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  double result;
  char hello[256] = "Hello World";

  for(i = 0; i < 10; ++i) {
    fprintf(stderr, "%s %2d %6.2f\n",
            hello, i, result);
  }
  return 0;
}
```

# Interacting with an External File in C

- To **read** a value to a file we can use the following command.

```
char* fgets(char *str, int size,
            FILE *fd)
```

- Get at most "size-1" number of characters from the <stream>.
- Store the result of the string in "str"
- Return a pointer to the string, or NULL if at the End-Of-File

**You can also use:**

```
int fgetc( FILE *stream );

ssize_t getline(char **buf, size_t *bytes,
                FILE *stream );
```

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  FILE *fd = NULL;
  char line[256];
  char *fgets_rtn = NULL;

  fd = fopen("king.txt", "r");
  if( NULL == fd ) {
    fprintf(stderr, "Error: fopen()\n");
    return -1;
  }


  fgets_rtn = fgets(line, 256, fd);
  if( NULL != fgets_rtn ) {
    printf("Line) %s", line);
  }




  fclose(fd);
  return 0;
}
```

# Interacting with an External File in C

- To **read** the entire file we need some mechanism to know when we have reached the end of the file.

```
int feof(FILE *fd)
```

- Returns non-zero if at the end of the file
- Returns zero while not at the end of the file

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  FILE *fd = NULL;
  char line[256];
  char *fgets_rtn = NULL;

  fd = fopen("king.txt", "r");
  if( NULL == fd ) {
    fprintf(stderr, "Error: fopen()\n");
    return -1;
  }

  while( 0 == feof(fd) ) {
    fgets_rtn = fgets(line, 256, fd);
    if( NULL == fgets_rtn ) {
      break;
    }
    printf("Line) %s", line);
  }


  fclose(fd);
  return 0;
}
```

# Checking if a file exists

```
#include <unistd.h>
```

Examples: files/*.c

- To **check** to see if a file exists (or if you have permission to use it):

```
int access(char *path, int amode);
```

| amode Options | |
|---|---|
| F_OK | If file exists |
| R_OK | Read access |
| W_OK | Write access |
| X_OK | Execute/search access |

```
char filename[256] = "input.txt";

if( 0 == access(filename, F_OK) ) {
  printf("The file %s exists\n", filename);
} else {
 printf("The file %s does not exist\n", filename);
}
```

# Flushing a stream

```
#include <unistd.h>
```

Examples: files/*.c

- To force a write of all buffered data for a given stream:

```
int fflush(FILE *stream);
```

```
// Flush output to stdout
fflush( stdout );

// Flush output to all open output streams
fflush( NULL );
```