C PROGRAMMING

Michael D'Argenio – mjdargen@ncsu.edu
Electrical Engineering – SS 2019 – Duke TIP

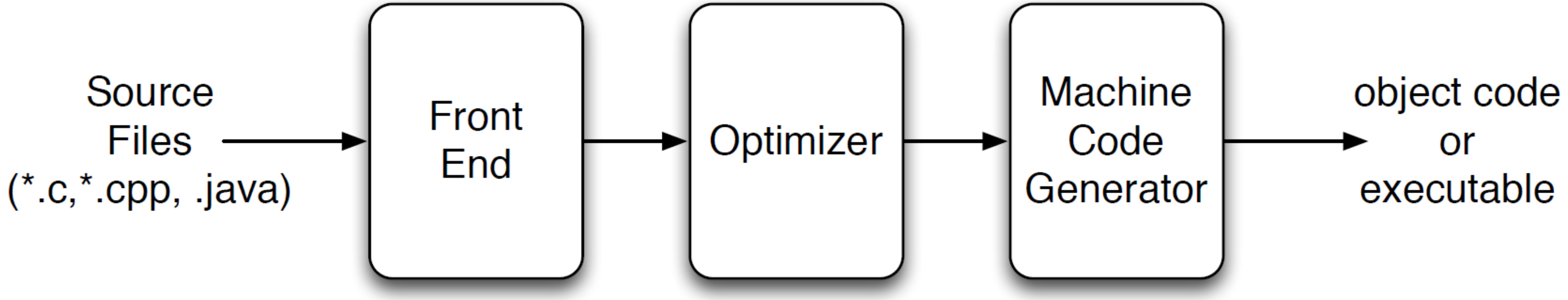
INTRO TO C PROGRAMMING

C: A High-Level Language

- Gives symbolic names to values
  - don’t need to know which register or memory location
- Provides abstraction of underlying hardware
  - operations do not depend on instruction set
  - example: can write “a = b * c”, even though architecture may not have a multiply instruction
- Provides expressiveness
  - use meaningful symbols that convey meaning
  - simple expressions for common control patterns (if-then-else)
- Enhances code readability
- Safeguards against bugs
  - can enforce rules or conditions at compile-time or run-time

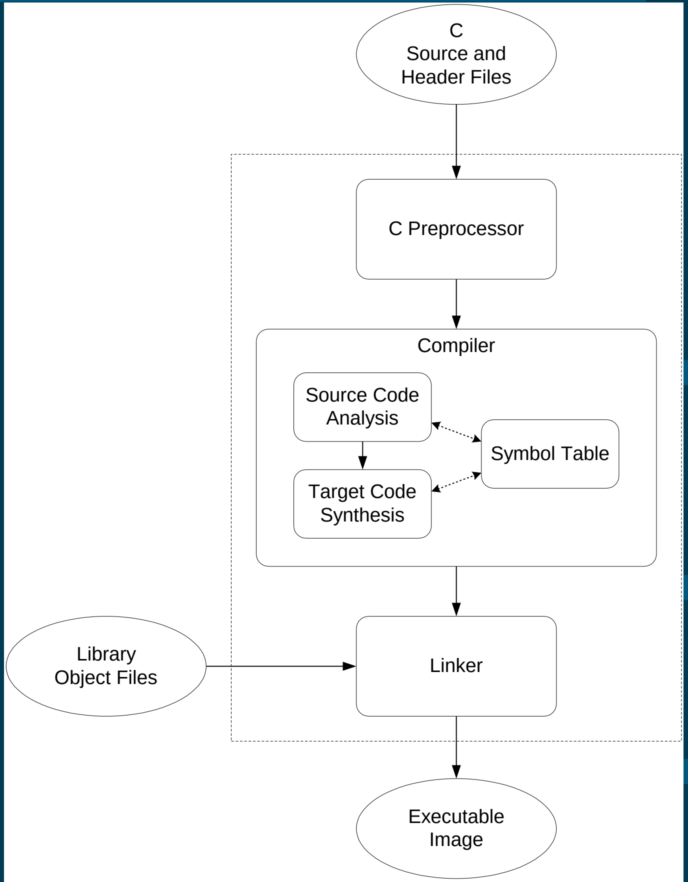
Compilation

- Translates high-level language statements into machine language
  - does not execute, but creates executable program
- Performs optimization over multiple statements.
- Changes to programs requires recompilation.



Compiling a C Program

- Entire mechanism is usually called the "compiler"
- Preprocessor
  - macro substitution
  - conditional compilation
  - "source-level" transformations
    - output is still C
- Compiler
  - generates object file
    - machine instructions
- Linker
  - combine object files (including libraries) into executable image



Compiler

- Source Code Analysis or “front end”
  - parses programs to identify its pieces (variables, expressions, statements, functions, etc.)
  - depends on language (not on target machine)
- Code Generation or “back end”
  - generates machine code from analyzed source
  - may optimize machine code to make it run more efficiently
  - very dependent on target machine
- Symbol Table
  - map between symbolic names and items
  - like assembler, but more kinds of information

A Simple C Program

```
#include <stdio.h>
#define STOP 0

/* Function: main                                        */
/* Description: counts down from user input to STOP */
main()
{
  /* variable declarations */
  int counter;   /* an integer to hold count values */
  int startPoint; /* starting point for countdown */
  /* prompt user for input */
  printf("Enter a positive number: ");
  scanf("%d", &startPoint);   /* read into startPoint */
  /* count down and print count */
  for (counter=startPoint; counter >= STOP; counter--)
    printf("%d\n", counter);
}
```

Preprocessor Directives

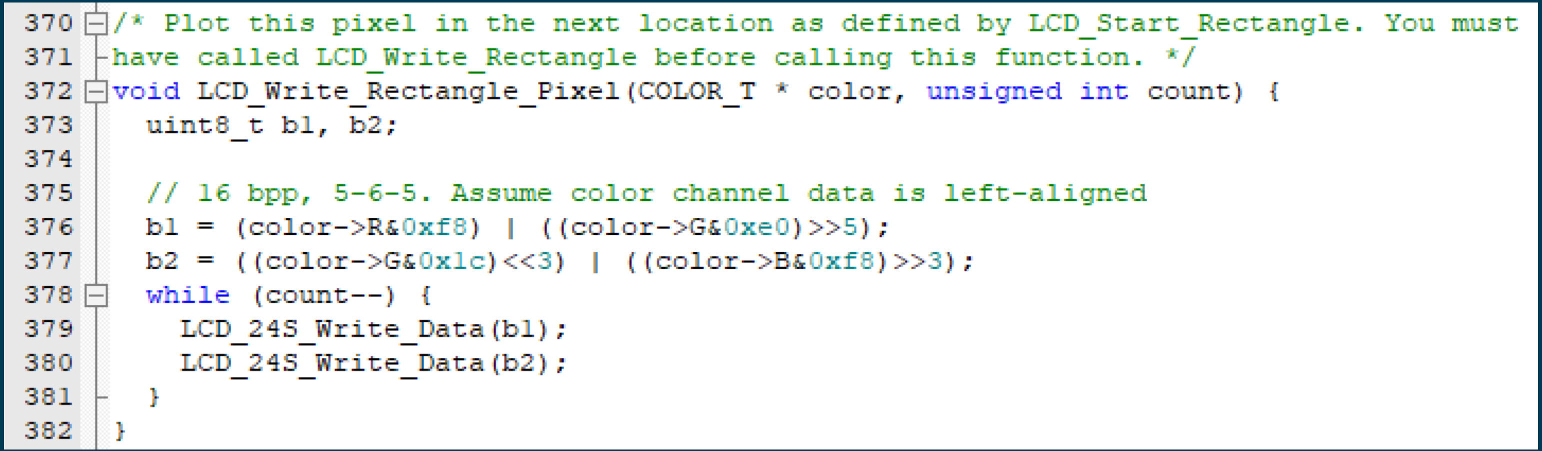
- `#include <stdio.h>`
  - Before compiling, copy contents of header file (stdio.h) into source code.
  - Header files typically contain descriptions of functions and variables needed by the program.
    - no restrictions -- could be any C source code

- `#define STOP 0`
  - Before compiling, replace all instances of the string "STOP" with the string "0"
  - Called a *macro*
  - Used for values that won't change during execution, but might change if the program is reused. (Must recompile.)

Comments

- Comments are used to help reader and yourself!
- Single line or inline comments
  - Begin with //
  - No end marker
- Multi-line or block comments
  - Begins with /* and ends with */
  - Can span multiple lines
  - Cannot have a comment within a comment
  - Comments are not recognized within a string (meaning in “”)
    - example: "my/*don't print this*/string“ would be printed as: my/*don't print this*/string

Comments Example



```c
/* Plot this pixel in the next location as defined by LCD_Start_Rectangle. You must
have called LCD_Write_Rectangle before calling this function. */
void LCD_Write_Rectangle_Pixel(COLOR_T * color, unsigned int count) {
  uint8_t b1, b2;

  // 16 bpp, 5-6-5. Assume color channel data is left-aligned
  b1 = (color->R&0xf8) | ((color->G&0xe0)>>5);
  b2 = ((color->G&0x1c)<<3) | ((color->B&0xf8)>>3);
  while (count--) {
    LCD_24S_Write_Data(b1);
    LCD_24S_Write_Data(b2);
  }
}
```

`main` Function

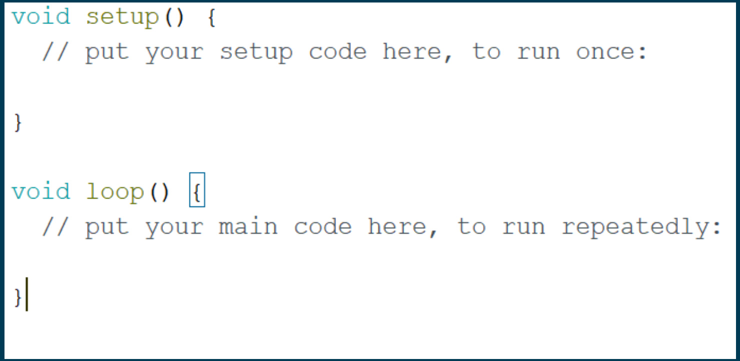
- Every C program must have a function called `main()`.
- This is the code that is executed when the program is run.
- The code for the function lives within brackets:

```
void main()
{
    /* code goes here */
}
```

*Special Note for Arduino

- Every C program must have a function called `main()`.
- However, Arduino has created an abstraction and instead has a `setup()` and `loop()` functions.
- This is what it looks like behind the scenes:

```
void main()
{
  setup();
  while (1)
  {
    loop();
  }
}
```



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Variable Declarations

- Variables are used as names for data items.
- Each variable has a *type*, which tells the compiler how the data is to be interpreted (and how much space it needs, etc.).

```
int counter;
int startPoint;
```

- `int` is a predefined integer type in C.

Input and Output

- Variety of I/O functions in *C Standard Library*.
- Must include `<stdio.h>` to use them.

```
printf("%d\n", counter);
```

- String contains characters to print and formatting directions for variables.
- This call says to print the variable `counter` as a decimal integer, followed by a linefeed (`\n`).

```
scanf("%d", &startPoint);
```

- String contains formatting directions for looking at input.
- This call says to read a decimal integer and assign it to the variable `startPoint`. (Don't worry about the `&` yet.)

More About Output

- Can print arbitrary expressions, not just variables

```
printf("%d\n", startPoint - counter);
```

- Print multiple expressions with a single statement

```
printf("%d %d\n", counter, startPoint - counter);
```

- Different formatting options:
  - `%d` decimal integer
  - `%x` hexadecimal integer
  - `%c` ASCII character
  - `%f` floating-point number

Examples

- This code:

```
printf("%d is a prime number.\n", 43);
printf("43 plus 59 in decimal is %d.\n", 43+59);
printf("43 plus 59 in hex is %x.\n", 43+59);
printf("43 plus 59 as a character is %c.\n", 43+59);
```

- Produces this output:

```
43 is a prime number.
43 + 59 in decimal is 102.
43 + 59 in hex is 66.
43 + 59 as a character is f.
```

Examples of Input

- Many of the same formatting characters are available for user input.
- `scanf("%c", &nextChar);`
  - reads a single character and stores it in nextChar
- `scanf("%f", &radius);`
  - reads a floating point number and stores it in radius
- `scanf("%d %d", &length, &width);`
  - reads two decimal integers (separated by whitespace), stores the first one in length and the second in width
- Must use ampersand (&) for variables being modified.
  * explained later

Compiling and Linking

- Various compilers available
  - cc, gcc
  - includes preprocessor, compiler, and linker

- Lots and lots of options!
  - level of optimization, debugging
  - preprocessor, linker options
  - intermediate files -- object (.o), assembler (.s), preprocessor (.i), etc.

VARIABLES

Basic C Elements

- Variables
  - named, typed data items
- Operators
  - predefined actions performed on data items
  - combined with variables to form expressions, statements

Data Types

- C has three basic data types:

`int`	integer (16 bits)
`double`	floating point (32 bits)
`char`	character (8 bits)

Variable Names

- Any combination of letters, numbers, and underscore (_)
- Case matters
  - "sum" is different than "Sum"
- Cannot begin with a number
  - usually, variables beginning with underscore are used only in special library routines
- 31 character limit to variable names

Examples

- Legal

```
i
wordsPerSecond
words_per_second
_green
aReally_longName_moreThan31chars
aReally_longName_moreThan31characters
```

same identifier

- Illegal

```
10sdigit
ten'sdigit
done?
double
```

reserved keyword

Literals

- Integer
  - `123    /* decimal */`
  - `-123`
  - `0x123 /* hexadecimal */`
- Floating point
  - `6.023`
  - `6.023e23` /* 6.023 x 10^{23} */
  - `5E12` /* 5.0 x 10^{12} */
- Character
  - `‘c’`
  - `'\n'   /* newline */`
  - `'\xA' /* ASCII 10 (0xA) */`

Scope: Global and Local

- Where is the variable accessible?
  - Global: accessed anywhere in program
  - Local: only accessible in a particular region
- Compiler infers scope from where variable is declared
  - programmer doesn't have to explicitly state
- Variable is local to the block in which it is declared
  - block defined by open and closed braces { }
  - can access variable declared in any "containing" block
- Global variable is declared outside all blocks

Example

```
#include <stdio.h>
int itsGlobal = 0;

main(){
  int itsLocal = 1;   /* local to main */
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
  {
    int itsLocal = 2;   /* local to this block */
    itsGlobal = 4;      /* change global variable */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
  }
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
}
```

Output

```
    Global 0 Local 1
    Global 4 Local 2
    Global 4 Local 1
```

OPERATIONS

Operators

- Programmers manipulate variables using the *operators* provided by the high-level language.
- Variables and operators combine to form *expressions* and *statements* which denote the work to be done by the program.
- Each operator may correspond to many machine instructions.
    - Example: The multiply operator (*) typically requires multiple ADD instructions. (multiply is not always supported)

Expression

- Any combination of variables, constants, operators, and function calls
  - every expression has a type, derived from the types of its components (according to C typing rules)

- Examples:
  - `counter >= STOP`
  - `x + sqrt(y)`
  - `x & z + 3 || 9 - w-- % 6`

Statement

- Expresses a complete unit of work
    - executed in sequential order

- Simple statement ends with semicolon
    - `z = x * y; /* assign product to z */`
    - `y = y + 1; /* after multiplication */`
    - `;  /* null statement */`

- Compound statement groups simple statements using braces.
    - syntactically equivalent to a simple statement
    - `{  z = x * y; y = y + 1; }`

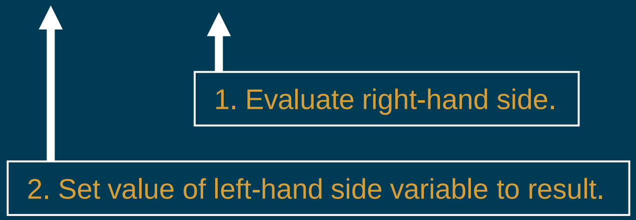
Operators

- Three things to know about each operator
- (1) Function
  - what does it do?
- (2) Precedence
  - in which order are operators combined?
  - Example:
    "a * b + c * d" is the same as "(a * b) + (c * d)"
    because multiply (*) has a higher precedence than addition (+)
- (3) Associativity
  - in which order are operators of the same precedence combined?
  - Example:
    "a - b - c" is the same as "(a - b) - c"
    because add/sub associate left-to-right

Assignment Operator

- Changes the value of a variable.

-
  ```
  x = x + 4;
  ```



1. Evaluate right-hand side.

2. Set value of left-hand side variable to result.

Assignment Operator

- All expressions evaluate to a value, even ones with the assignment operator.
- For assignment, the result is the value assigned.
  - usually (but not always) the value of the right-hand side
    - type conversion might make assigned value different than computed value
- Assignment associates right to left.
- `y = x = 3;`
- y gets the value 3, because (x = 3) evaluates to the value 3.

Arithmetic Operators

Symbol	Operation	Usage	Precedence	Assoc
*	multiply	`x * y`	6	l-to-r
/	divide	`x / y`	6	l-to-r
%	modulo	`x % y`	6	l-to-r
+	addition	`x + y`	7	l-to-r
-	subtraction	`x - y`	7	l-to-r

- All associate left to right.
- `* / %` have higher precedence than `+ -`.

Arithmetic Expressions

- If mixed types, smaller type is "promoted" to larger.

  ```
  x + 4.3
  ```

  if x is int, converted to double and result is double

- Integer division -- fraction is dropped.

  ```
  x / 3
  ```

  if x is int and x=5, result is 1 (not 1.666666...)

- Modulo -- result is remainder.

  ```
  x % 3
  ```

  if x is int and x=5, result is 2.

Bitwise Operators

Symbol	Operation	Usage	Precedence	Assoc
~	bitwise NOT	`~x`	4	r-to-l
<<	left shift	`x << y`	8	l-to-r
>>	right shift	`x >> y`	8	l-to-r
&	bitwise AND	`x & y`	11	l-to-r
^	bitwise XOR	`x ^ y`	12	l-to-r
|	bitwise OR	`x | y`	13	l-to-r

- Operate on variables bit-by-bit.
  - Like LC-3 AND and NOT instructions.
- Shift operations are logical (not arithmetic). Operate on *values* -- neither operand is changed.

Logical Operators

Symbol	Operation	Usage	Precedence	Assoc
!	logical NOT	`!x`	4	r-to-l
&&	logical AND	`x && y`	14	l-to-r
||	logical OR	`x || y`	15	l-to-r

- Treats entire variable (or value) as TRUE (non-zero) or FALSE (zero).
- Result is 1 (TRUE) or 0 (FALSE).

Relational Operators

Symbol	Operation	Usage	Precedence	Assoc
>	greater than	`x > y`	9	l-to-r
>=	greater than or equal	`x >= y`	9	l-to-r
<	less than	`x < y`	9	l-to-r
<=	less than or equal	`x <= y`	9	l-to-r
==	equal	`x == y`	10	l-to-r
!=	not equal	`x != y`	10	l-to-r

- Result is 1 (TRUE) or 0 (FALSE).
- Note: Don't confuse equality (==) with assignment (=).

Special Operators: ++ and --

- Changes value of variable before (or after) its value is used in an expression.

Symbol	Operation	Usage	Precedence	Assoc
++	postincrement	`x++`	2	r-to-l
--	postdecrement	`x--`	2	r-to-l
++	preincrement	`++x`	3	r-to-l
--	predecrement	`--x`	3	r-to-l

- Pre: Increment/decrement variable before using its value.
- Post: Increment/decrement variable after using its value.

Using ++ and --

- `x = 4;`
- `y = x++;`
- Results: **x = 5, y = 4** (because x is incremented after assignment)

- `x = 4;`
- `y = ++x;`
- Results: **x = 5, y = 5** (because x is incremented before assignment)

Practice with Precedence

- Assume a=1, b=2, c=3, d=4.

```
x = a * b + c * d / 2;  /* x = 8 */
```

same as:

```
x = (a * b) + ((c * d) / 2);
```

- For long or confusing expressions, use parentheses, because reader might not have memorized precedence table.
- Note: Assignment operator has lowest precedence, so all the arithmetic operations on the right-hand side are evaluated first.

Symbol Table

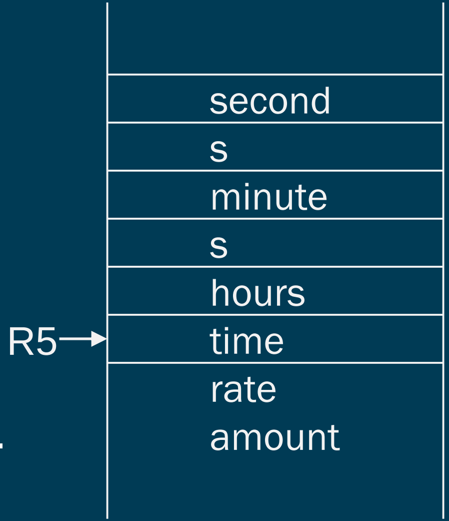
- Like assembler, compiler needs to know information associated with identifiers.
- In assembler, all identifiers were labels and information is address

- Compiler keeps more information
  - Name (identifier)
  - Type
  - Location in memory
  - Scope

Name	Type	Offset	Scope
amount	int	0	main
hours	int	-3	main
minutes	int	-4	main
rate	int	-1	main
seconds	int	-5	main
time	int	-2	main

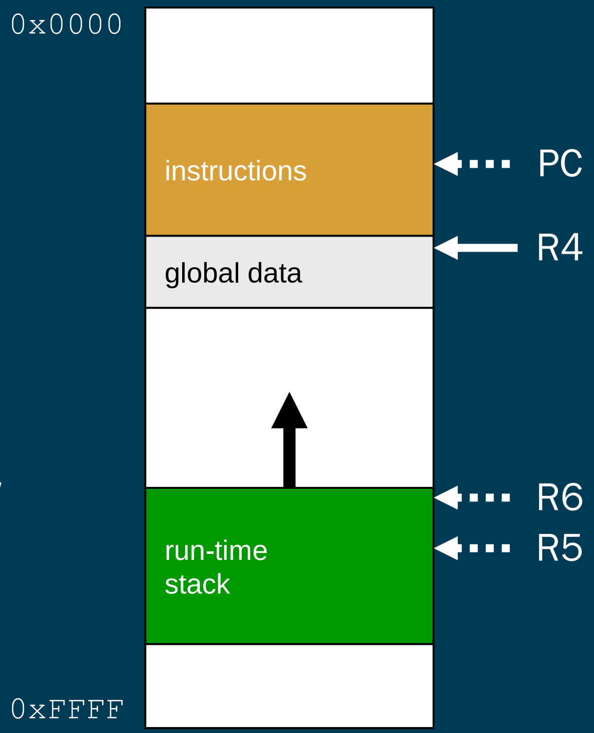
Local Variable Storage

- Local variables are stored in an *activation record*, also known as a *stack frame*.
- Symbol table "offset" gives the distance from the base of the frame.
  - R5 is the frame pointer – holds address of the base of the current frame.
  - A new frame is pushed on the run-time stack each time a block is entered.
  - Because stack grows downward, base is the highest address of the frame, and variable offsets are <= 0.



Allocating Space for Variables

- Global data section
  - All global variables stored here (actually all static variables)
  - R4 points to beginning
- Run-time stack
  - Used for local variables
  - R6 points to top of stack
  - R5 points to top frame on stack
  - New frame for each block (goes away when block exited)
- Offset = distance from beginning of storage area
  - Global: `LDR R1, R4, #4`
  - Local: `LDR R2, R5, #-3`



Variables and Memory Locations

- In our examples, a variable is always stored in memory.
- When assigning to a variable, must store to memory location.
- A real compiler would perform code optimizations that try to keep variables allocated in registers.
- Why?

Example: Compiling to LC-3

```
#include <stdio.h>
int inGlobal;

main()
{
  int inLocal;   /* local to main */
  int outLocalA;
  int outLocalB;

  /* initialize */
  inLocal = 5;
  inGlobal = 3;

  /* perform calculations */
  outLocalA = inLocal++ & ~inGlobal;
  outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

  /* print results */
  printf("The results are: outLocalA = %d, outLocalB = %d\n",
         outLocalA, outLocalB);
}
```

Example: Symbol Table

Name	Type	Offset	Scope
`inGlobal`	`int`	0	`global`
`inLocal`	`int`	0	`main`
`outLocalA`	`int`	-1	`main`
`outLocalB`	`int`	-2	`main`

Example: Code Generation

```
; main
; initialize variables
        AND R0, R0, #0
        ADD R0, R0, #5   ; inLocal = 5
        STR R0, R5, #0   ; (offset = 0)

        AND R0, R0, #0
        ADD R0, R0, #3   ; inGlobal = 3
        STR R0, R4, #0   ; (offset = 0)
```

Example (continued)

```
; first statement:
; outLocalA = inLocal++ & ~inGlobal;
        LDR R0, R5, #0   ; get inLocal
        ADD R1, R0, #1   ; increment
        STR R1, R5, #0   ; store

        LDR R1, R4, #0   ; get inGlobal
        NOT R1, R1       ; ~inGlobal
        AND R2, R0, R1   ; inLocal & ~inGlobal
        STR R2, R5, #-1  ; store in outLocalA
                         ; (offset = -1)
```

Example (continued)

- ; next statement:
- ; outLocalB = (inLocal + inGlobal)
  ;   - (inLocal - inGlobal);

-
  ```
        LDR R0, R5, #0   ; inLocal
        LDR R1, R4, #0   ; inGlobal
        ADD R0, R0, R1   ; R0 is sum
        LDR R2, R5, #0   ; inLocal
        LDR R3, R5, #0   ; inGlobal
        NOT R3, R3
        ADD R3, R3, #1
        ADD R2, R2, R3   ; R2 is difference
        NOT R2, R2       ; negate
        ADD R2, R2, #1
        ADD R0, R0, R2   ; R0 = R0 - R2
        STR R0, R5, #-2 ; outLocalB (offset = -2)
  ```

Special Operators: +=, *=, etc.

- Arithmetic and bitwise operators can be combined with assignment operator.

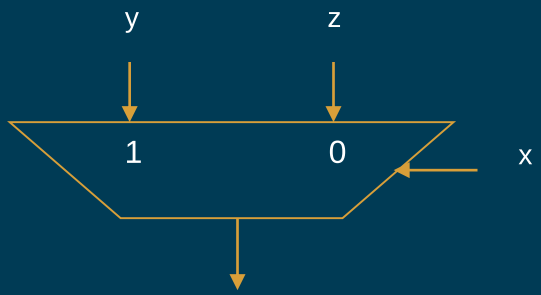
Statement	Equivalent assignment
`x += y;`	`x = x + y;`
`x -= y;`	`x = x - y;`
`x *= y;`	`x = x * y;`
`x /= y;`	`x = x / y;`
`x %= y;`	`x = x % y;`
`x &= y;`	`x = x & y;`
`x |= y;`	`x = x | y;`
`x ^= y;`	`x = x ^ y;`
`x <<= y;`	`x = x << y;`
`x >>= y;`	`x = x >> y;`

All have same precedence and associativity as = and associate right-to-left.

Special Operator: Conditional

Symbol	Operation	Usage	Precedence	Assoc
`? :`	conditional	`x?y:z`	16	l-to-r

- If x is TRUE (non-zero), result is y; else, result is z.
- Like a MUX, with x as the select signal.



CONTROL STRUCTURES

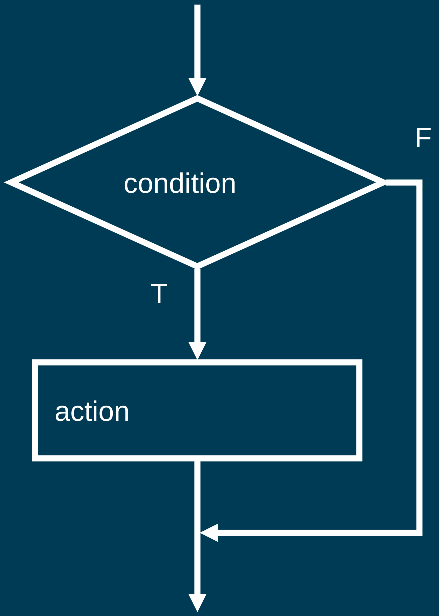
Control Structures

- Conditional - making a decision about which code to execute, based on evaluated expression
  - `if`
  - `if-else`
  - `switch`

- Iteration - executing code multiple times, ending based on evaluated expression
  - `while`
  - `for`
  - `do-while`

If

-
  ```
  if (condition)
    action;
  ```
- Condition is a C expression, which evaluates to TRUE (non-zero) or FALSE (zero).
- Action is a C statement, which may be simple or compound (a block).



Condition is a C expression, which evaluates to TRUE (non-zero) or FALSE (zero).
Action is a C statement, which may be simple or compound (a block).

Example If Statements

```
if (x <= 10)
      y = x * x + 5;
```

```
if (x <= 10) {
      y = x * x + 5;
      z = (2 * y) / 3;
}
```

compound statement;
both executed if x <= 10

```
if (x <= 10)
      y = x * x + 5;
      z = (2 * y) / 3;
```

only first statement is conditional;
second statement is
always executed

More If Examples

-
  ```
  if (0 <= age && age <= 11)
     kids += 1;
  ```

-
  ```
  if (month == 4 || month == 6 ||
          month == 9 || month == 11)
     printf("The month has 30 days.\n");
  ```

-
  ```
  if (x = 2)
     y = 5;
  ```

always true,
so action is *always* executed!

This is a common programming error (= instead of ==), not caught by compiler because it's syntactically correct.

If's Can Be Nested

The following two statements are the same:

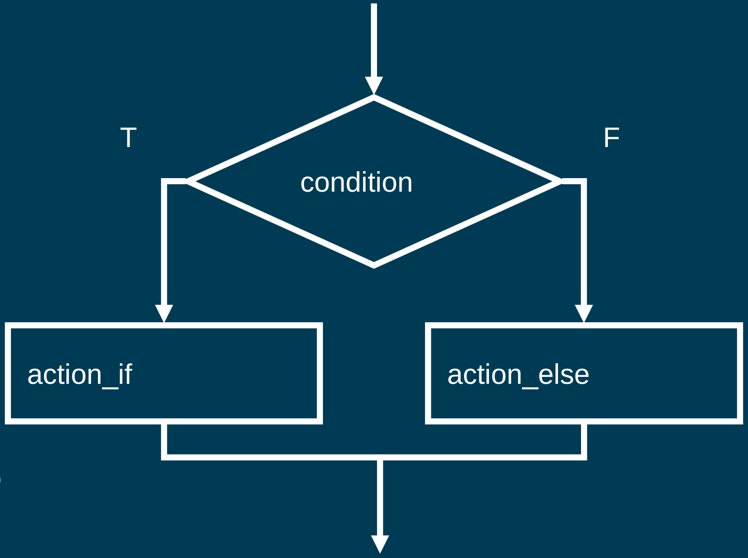
```
if (x == 3) {
   if (y != 6) {
      z = z + 1;
      w = w + 2;
   }
}
```

```
if ((x == 3) && (y != 6)){
   z = z + 1;
   w = w + 2;
}
```

If-else

-
  ```
  if (condition)
     action_if;
  else
     action_else;
  ```

- *Else* allows choice between two mutually exclusive actions without re-testing condition.



Matching Else with If

- Else is always associated with *closest* unassociated if.

```
if (x != 10)
  if (y > 3)
    z = z / 2;
  else
    z = z * 2;
```

is the same as...

```
if (x != 10) {
  if (y > 3)
    z = z / 2;
  else
    z = z * 2;
}
```

is NOT the same as...

```
if (x != 10) {
  if (y > 3)
    z = z / 2;
}
else
    z = z * 2;
```

If, Else If, Else

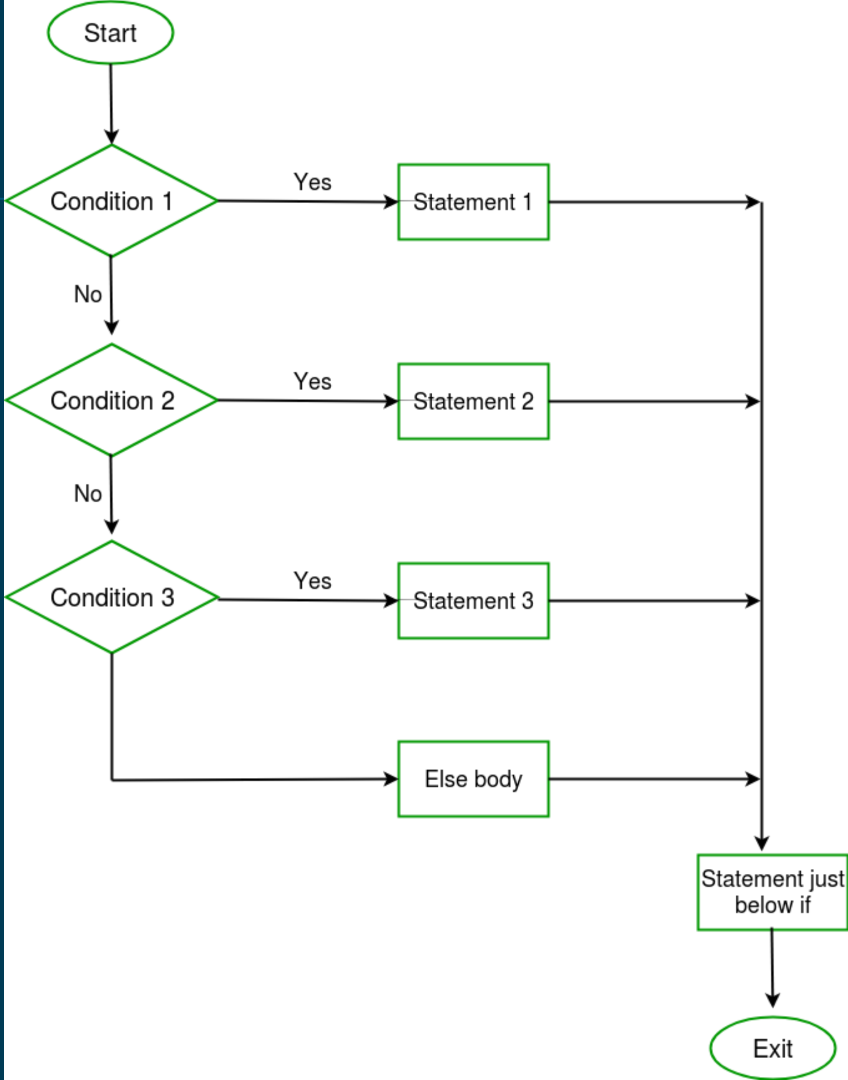
```
if (condition == 0)
  action_if;
else if (condition == 1)
  action_else;
else if (condition == 2)
  action_else;
else
  action_else;
```

if

else if

else if

else



Chaining If's and Else's

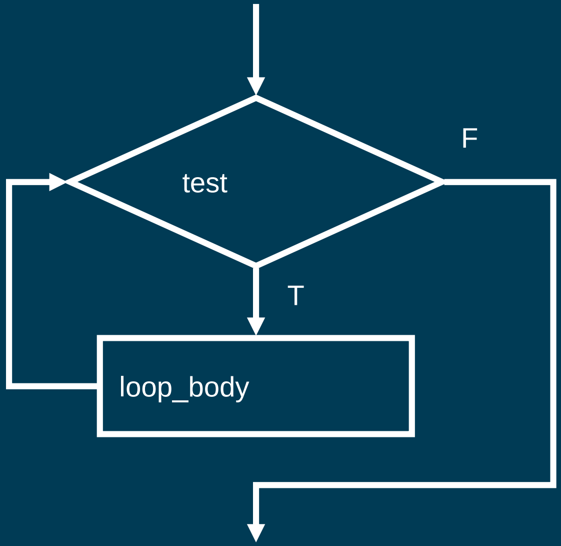
```
if (month == 4 || month == 6 || month == 9 ||
    month == 11)
   printf("Month has 30 days.\n");
else if (month == 1 || month == 3 ||
         month == 5 || month == 7 ||
         month == 8 || month == 10 ||
         month == 12)
   printf("Month has 31 days.\n");
else if (month == 2)
   printf("Month has 28 or 29 days.\n");
else
   printf("Don't know that month.\n");
```

While

-
  ```
  while (test)
     loop_body;
  ```

- Executes loop body as long as test evaluates to TRUE (non-zero).

- Note: Test is evaluated **before** executing loop body.



Infinite Loops

- The following loop will never terminate:

-
  ```
  x = 0;
  while (x < 10)
    printf("%d ", x);
  ```

- Loop body does not change condition, so test never fails.

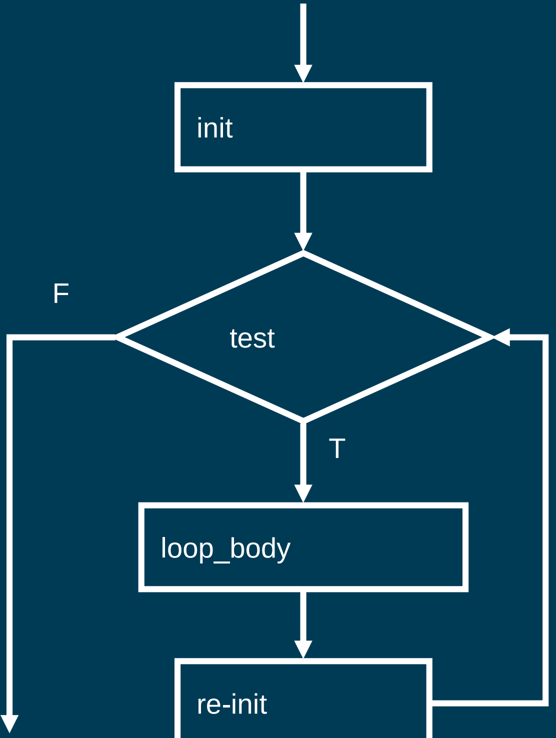
- This is a common programming error that can be difficult to find.

For

-
  ```
  for (init; end-test; re-init)
     statement
  ```

- Executes loop body as long as test evaluates to TRUE (non-zero). Initialization and re-initialization code included in loop statement.

- Note: Test is evaluated **before** executing loop body



Example For Loops

- `/* -- what is the output of this loop? -- */`

-
  ```
  for (i = 0; i <= 10; i ++)
    printf("%d ", i);
  ```

- `/* -- what does this one output? -- */`
- `letter = 'a';`

-
  ```
  for (c = 0; c < 26; c++)
    printf("%c ", letter+c);
  ```

- `/* -- what does this loop do? -- */`

-
  ```
  numberOfOnes = 0;
  for (bitNum = 0; bitNum < 16; bitNum++) {
    if (inputValue & (1 << bitNum))
      numberOfOnes++;
  }
  ```

Nested Loops

- Loop body can (of course) be another loop.

```
/* print a multiplication table */
for (mp1 = 0; mp1 < 10; mp1++) {
  for (mp2 = 0; mp2 < 10; mp2++) {
    printf("%d\t", mp1*mp2);
  }
  printf("\n");
}
```

Braces aren't necessary, for single line loops but they make the code easier to read.

Another Nested Loop

- The test for the inner loop depends on the counter variable of the outer loop.

-
  ```
  for (outer = 1; outer <= input; outer++) {
    for (inner = 0; inner < outer; inner++) {
      sum += inner;
    }
  }
  ```

For vs. While

- In general:

- *For* loop is preferred for *counter-based* loops.
  - Explicit counter variable
  - Easy to see how counter is modified each loop

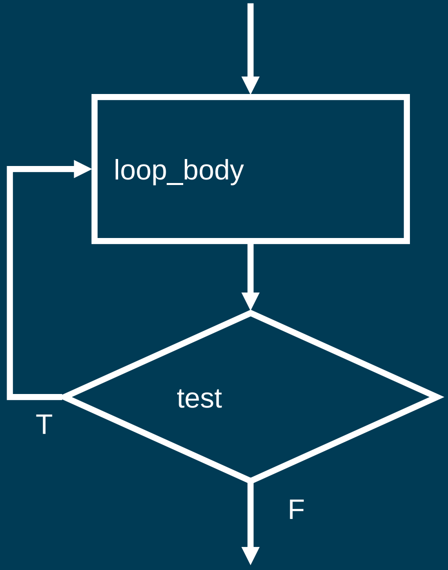
- *While* loop is preferred for *sentinel-based* loops.
  - Test checks for sentinel value.

- Either kind of loop can be expressed as the other, so it’s really a matter of style and readability.

Do-While

```
do
   loop_body;
while (test);
```

- Executes loop body as long as test evaluates to TRUE (non-zero).
- Note: Test is evaluated **after** executing loop body.



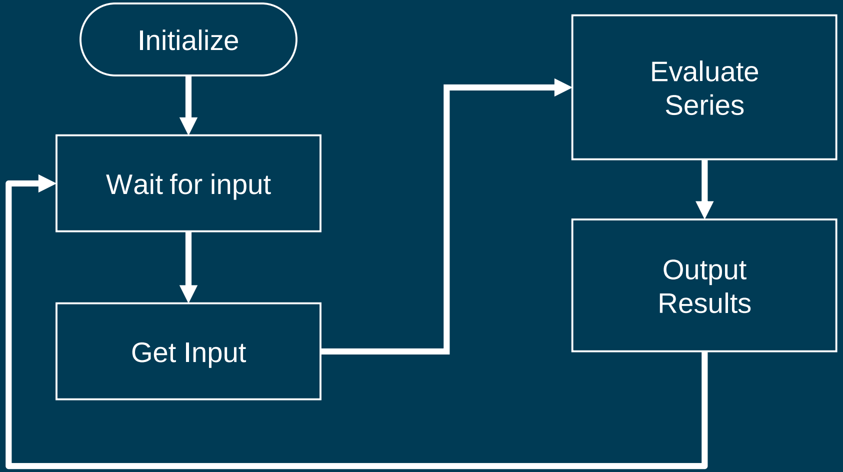
Problem Solving in C

- Same basic constructs
  - Sequential -- C statements
  - Conditional -- if-else, switch
  - Iterative -- while, for, do-while

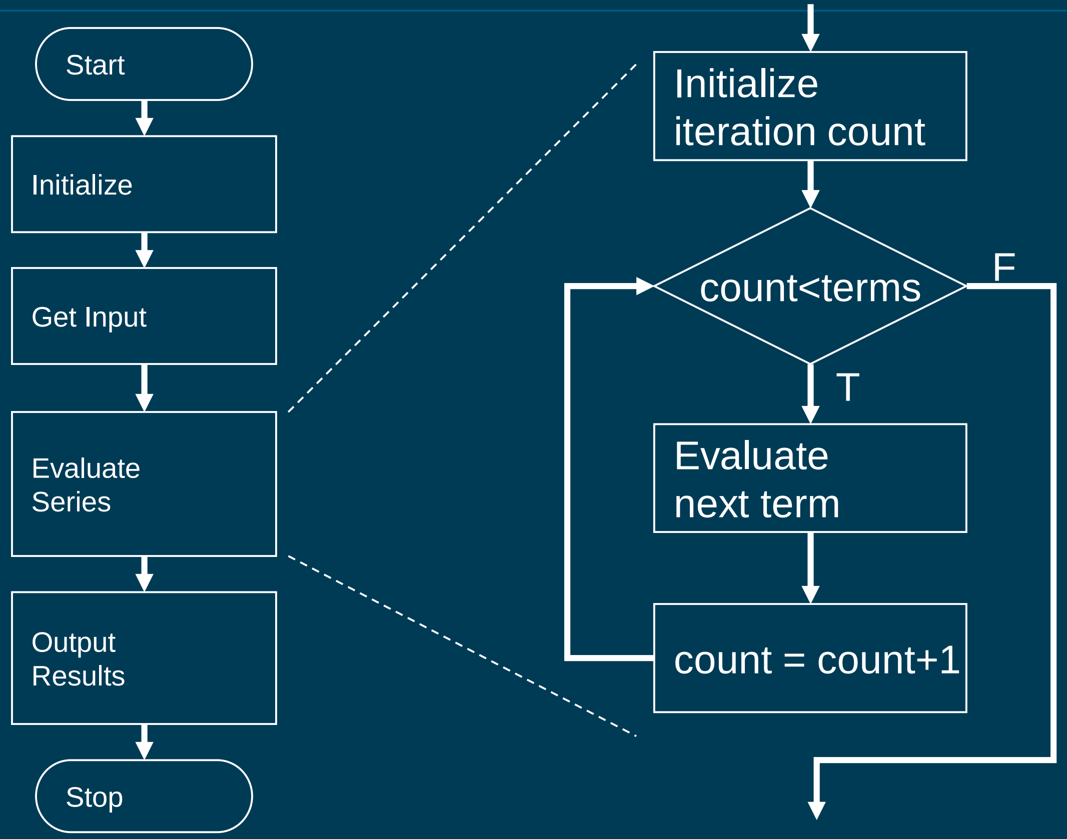
Problem 1: Calculating Pi

- Calculate π using its series expansion. User inputs number of terms.

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots + (-1)^{n-1}\frac{4}{2n+1} + \cdots$$

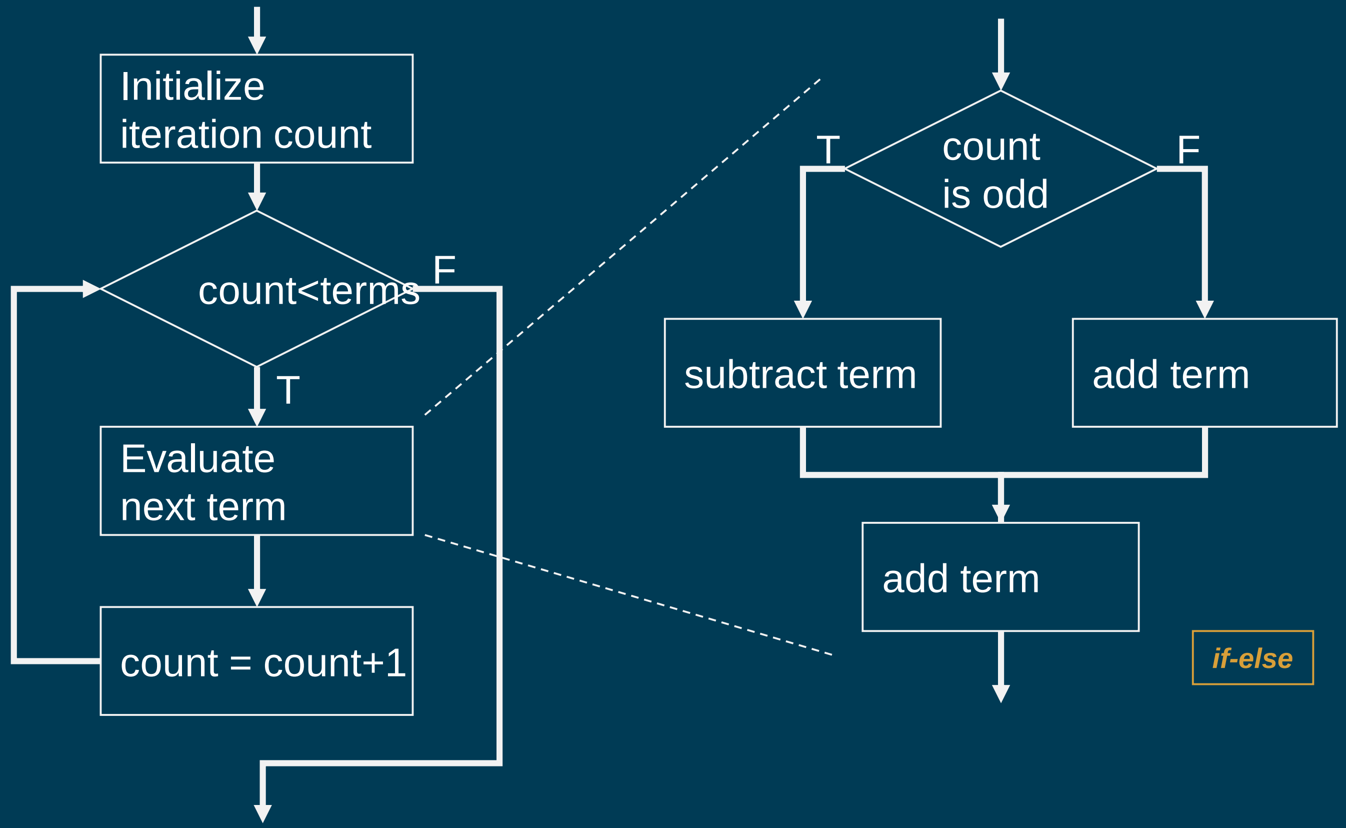


Pi: 1st refinement



for loop

Pi: 2nd refinement

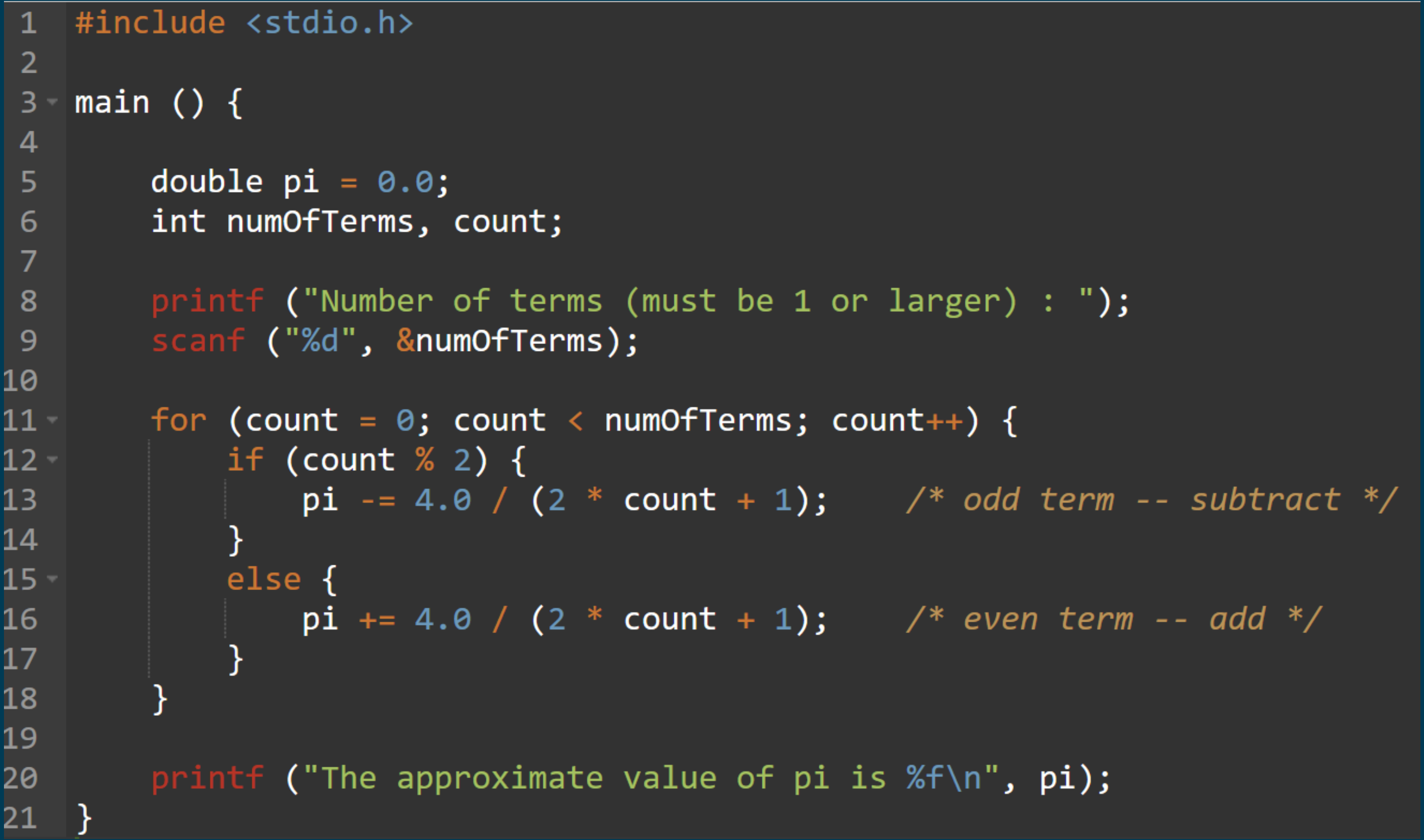


Pi: Code for Evaluate Terms

```
for (count=0; count < numOfTerms; count++) {
  if (count % 2) {
    /* odd term -- subtract */
    pi -= 4.0 / (2 * count + 1);
  }
  else {
    /* even term -- add */
    pi += 4.0 / (2 * count + 1);
  }
}
```

Note: Code in text is slightly different, but this code corresponds to equation.

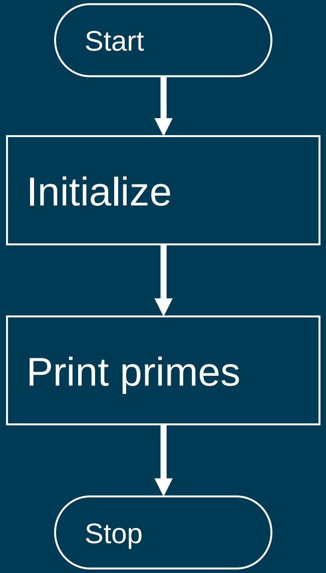
Pi: Complete Code



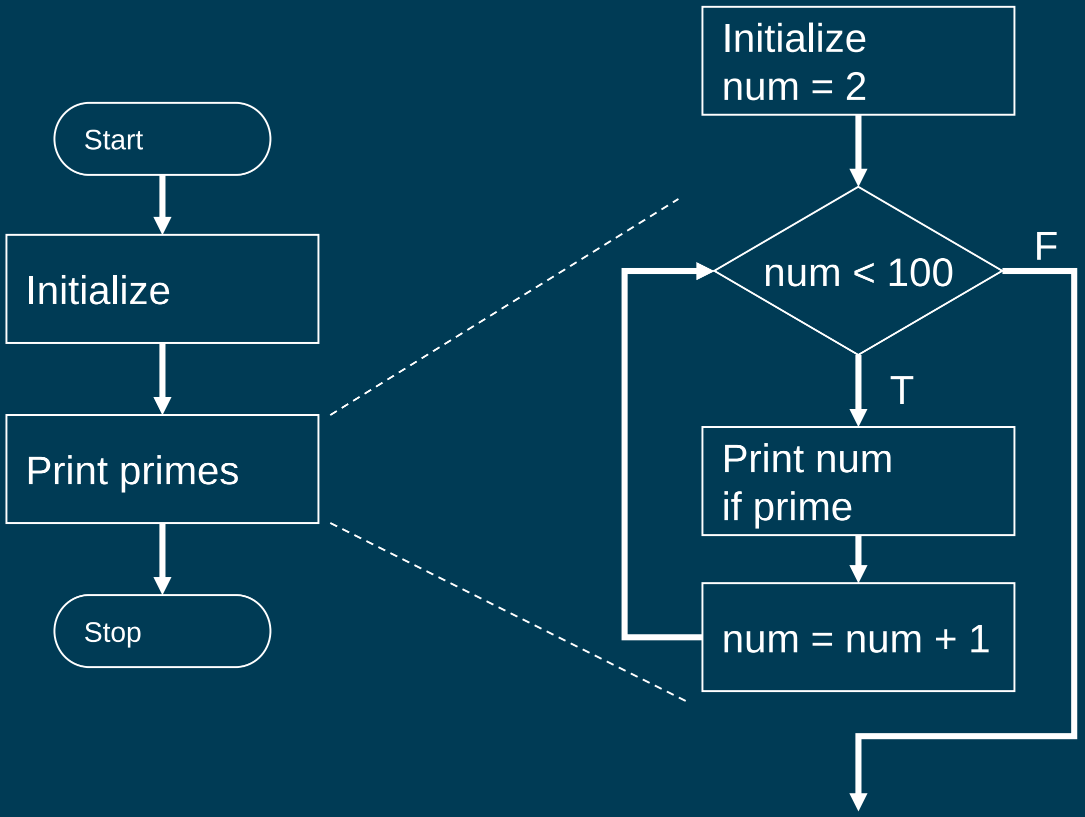
```
#include <stdio.h>

main () {

    double pi = 0.0;
    int numOfTerms, count;

    printf ("Number of terms (must be 1 or larger) : ");
    scanf ("%d", &numOfTerms);

    for (count = 0; count < numOfTerms; count++) {
        if (count % 2) {
            pi -= 4.0 / (2 * count + 1);    /* odd term -- subtract */
        }
        else {
            pi += 4.0 / (2 * count + 1);    /* even term -- add */
        }
    }

    printf ("The approximate value of pi is %f\n", pi);
}
```

Problem 2: Finding Prime Numbers

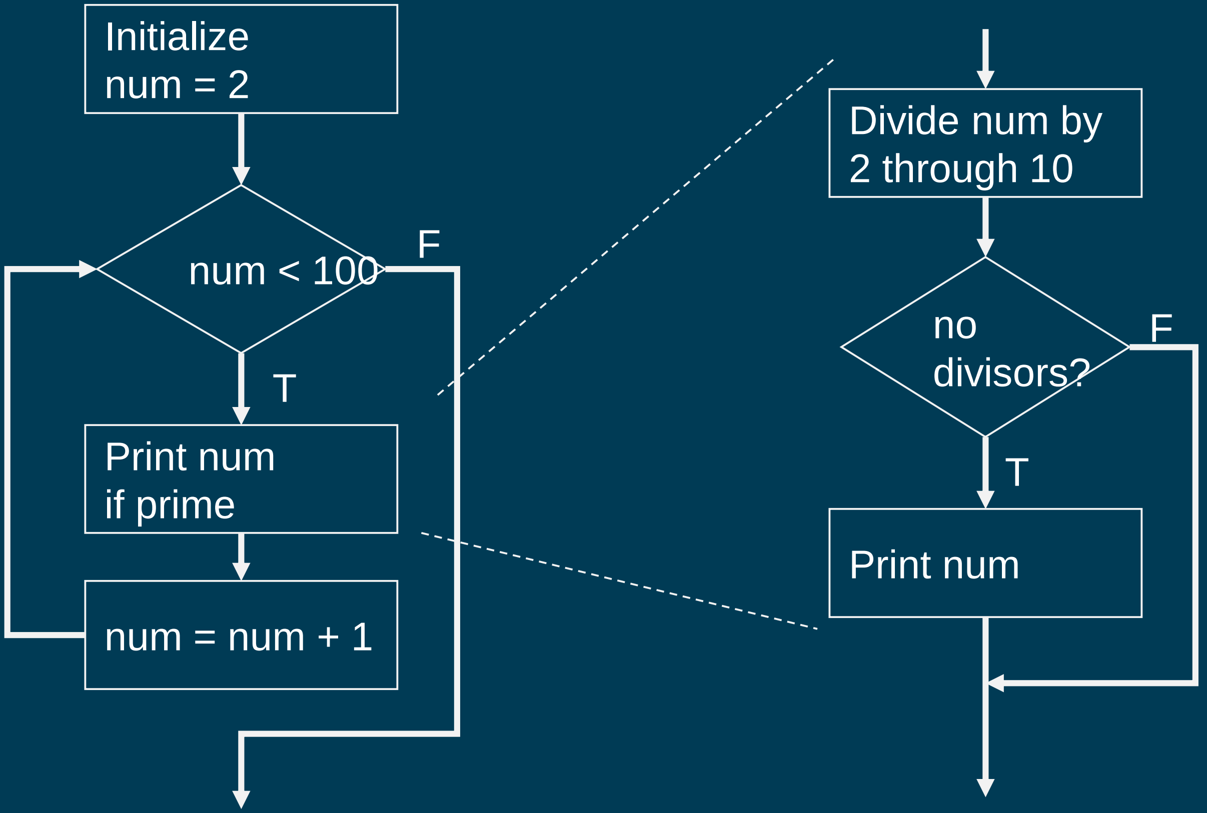
- Print all prime numbers less than 100.
    - A number is prime if its only divisors are 1 and itself.
    - All non-prime numbers less than 100 will have a divisor between 2 and 10.



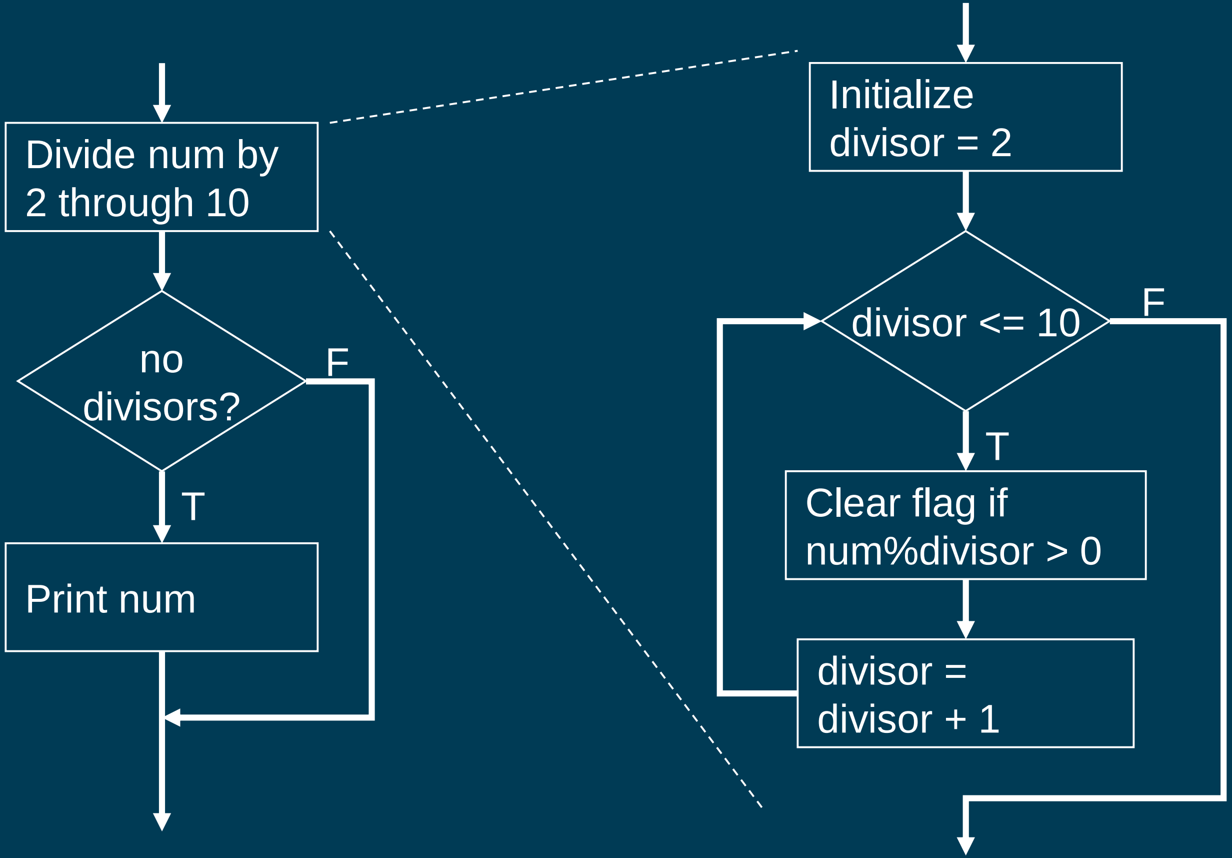
Primes: 1st refinement



Primes: 2nd refinement



Primes: 3rd refinement



Primes: Using a Flag Variable

- To keep track of whether number was divisible, we use a "flag" variable.
  - Set prime = TRUE, assuming that this number is prime.
  - If any divisor divides number evenly, set prime = FALSE.
    - Once it is set to FALSE, it stays FALSE.
  - After all divisors are checked, number is prime if the flag variable is still TRUE.

- Use macros to help readability.

```
#define TRUE  1
#define FALSE 0
```

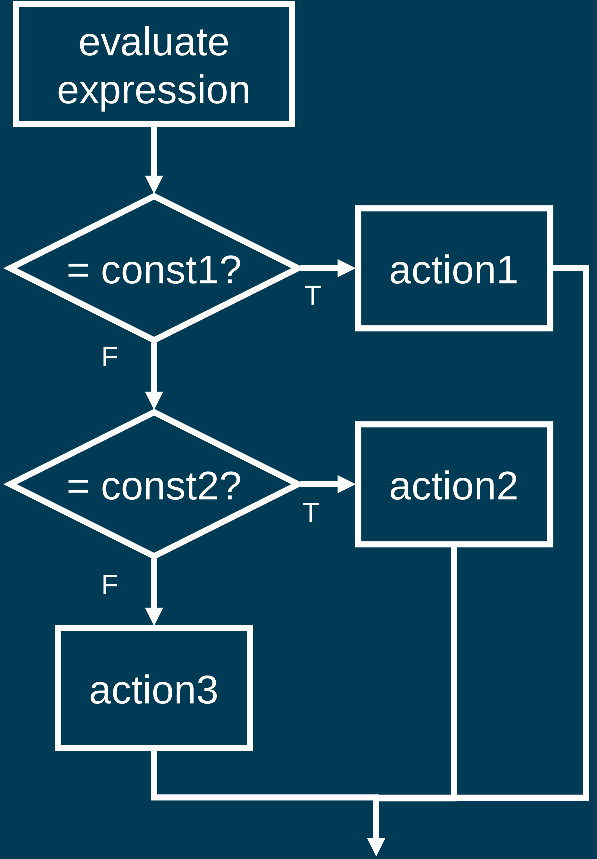
Primes: Complete Code

```
#include <stdio.h>
#define TRUE  1
#define FALSE 0

main () {
  int num, divisor, prime;

  /* start with 2 and go up to 100 */
  for (num = 2; num < 100; num ++ ) {

    prime = TRUE;  /* assume num is prime */
    /* test whether divisible by 2 through 10 */
    for (divisor = 2; divisor <= 10; divisor++)
      if (((num % divisor) == 0) && (num != divisor))
        prime = FALSE;  /* not prime */

    if (prime)  /* if prime, print it */
      printf("The number %d is prime\n", num);
  }
}
```

Switch

```
switch (expression) {
case const1:
  action1; break;
case const2:
  action2; break;
default:
  action3;
}
```

- Alternative to long if-else chain.
- If break is not used, then case "falls through" to the next.



Switch Example

```
/* same as month example for if-else */
switch (month) {
  case 4:
  case 6:
  case 9:
  case 11:
    printf("Month has 30 days.\n");
    break;
  case 1:
  case 3:
  /* some cases omitted for brevity...*/
    printf("Month has 31 days.\n");
    break;
  case 2:
    printf("Month has 28 or 29 days.\n");
    break;
  default:
    printf("Don't know that month.\n");
}
```

More About Switch

- Case expressions must be constant.
-
```
        case i:    /* illegal if i is a variable */
```

- If no break, then next case is also executed.
-
```
        switch (a) {
          case 1:
            printf("A");
          case 2:
            printf("B");
          default:
            printf("C");
        }
```

If a is 1, prints "ABC".
If a is 2, prints "BC".
Otherwise, prints "C".

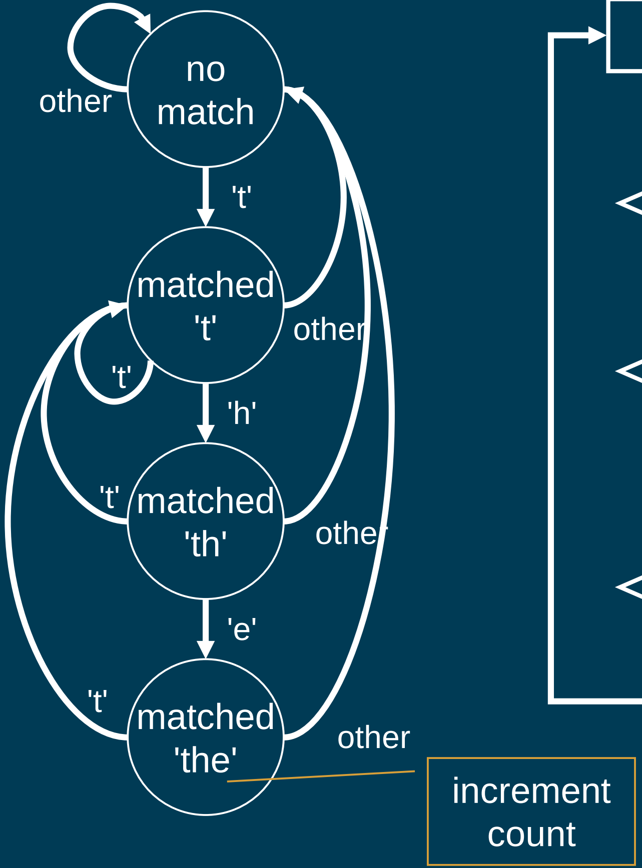
Problem 3: Searching for Substring

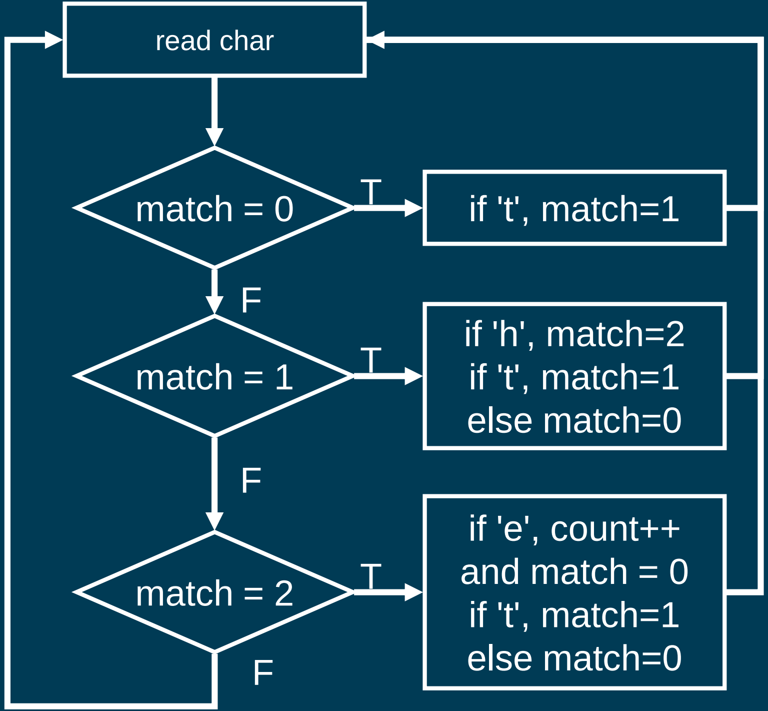
- Have user type in a line of text (ending with linefeed) and print the number of occurrences of "the".

- Reading characters one at a time
  - Use the `getchar()` function -- returns a single character.

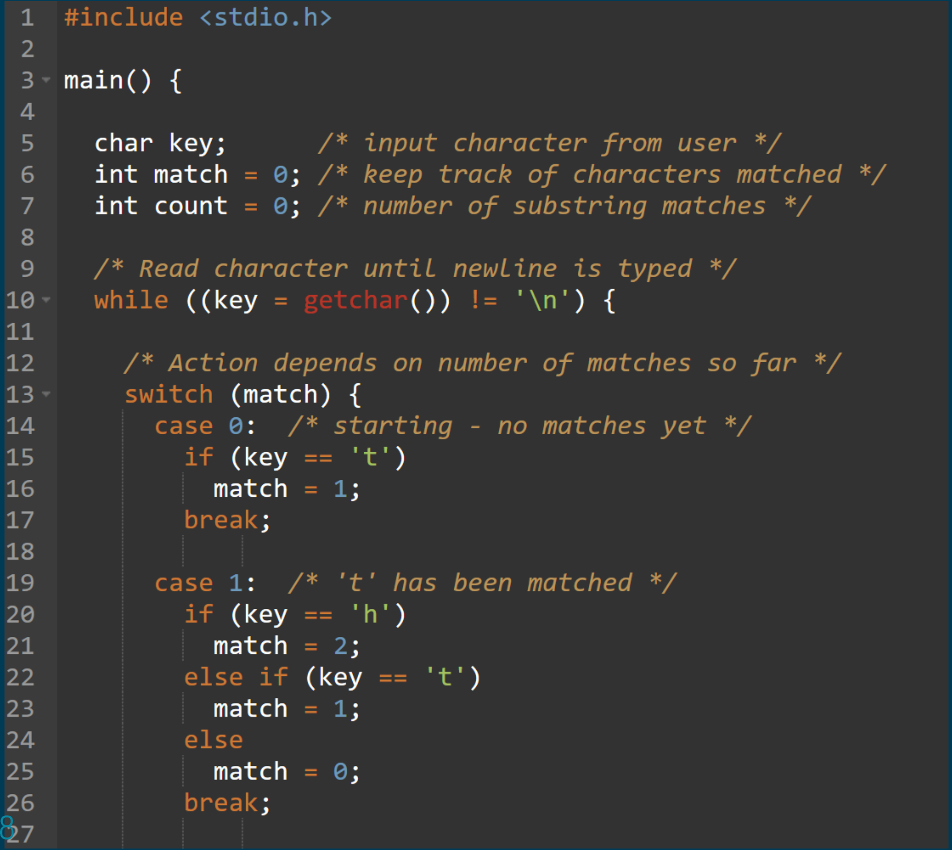
- Don't need to store input string; look for substring as characters are being typed.
  - Similar to state machine: based on characters seen, move toward success state or move back to start state.
  - Switch statement is a good match to state machine.

Substring: State machine to flow chart

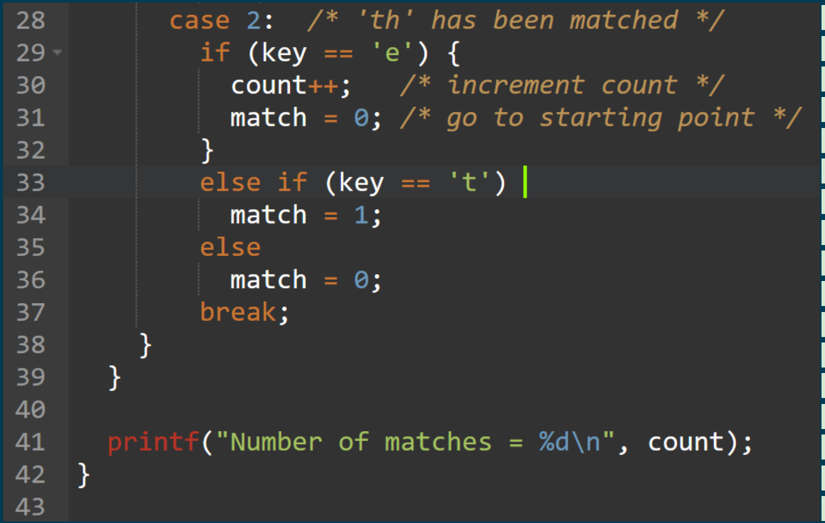




Substring: Code (Part 1)



```c
#include <stdio.h>

main() {

  char key;      /* input character from user */
  int match = 0; /* keep track of characters matched */
  int count = 0; /* number of substring matches */

  /* Read character until newline is typed */
  while ((key = getchar()) != '\n') {

    /* Action depends on number of matches so far */
    switch (match) {
      case 0:  /* starting - no matches yet */
        if (key == 't')
          match = 1;
        break;

      case 1:  /* 't' has been matched */
        if (key == 'h')
          match = 2;
        else if (key == 't')
          match = 1;
        else
          match = 0;
        break;

```



```c
      case 2:  /* 'th' has been matched */
        if (key == 'e') {
          count++;   /* increment count */
          match = 0; /* go to starting point */
        }
        else if (key == 't')
          match = 1;
        else
          match = 0;
        break;
    }
  }

  printf("Number of matches = %d\n", count);
}

```

Break and Continue

- `break;`
  - used *only* in switch statement or iteration statement
  - passes control out of the “smallest” (loop or switch) statement containing it to the statement immediately following
  - usually used to exit a loop before terminating condition occurs (or to exit switch statement when case is done)
- `continue;`
  - used only in iteration statement
  - terminates the execution of the loop body for this iteration
  - loop expression is evaluated to see whether another iteration should be performed
  - if `for` loop, also executes the re-initializer

Example

- What does the following loop do?

```
for (i = 0; i <= 20; i++) {
  if (i%2 == 0) continue;
  printf("%d ", i);
}
```

- What would be an easier way to write this?

- What happens if `break` instead of `continue`?

FUNCTIONS

Function

- Smaller, simpler, subcomponent of program
- Provides abstraction
  - hide low-level details
  - give high-level structure to program, easier to understand overall program flow
  - enables separable, independent development
- C functions
  - zero or multiple arguments passed in
  - single result returned (optional)
  - return value is always a particular type
- In other languages, called procedures, subroutines, ...

Example of High-Level Structure

```
main() {
  SetupBoard();   /* place pieces on board */

  DetermineSides();   /* choose black/white */


  /* Play game */
  do {
    WhitesTurn();
    BlacksTurn();
  } while (NoOutcomeYet());
}
```

Structure of program is evident, even without knowing implementation.

Functions in C

- Declaration (also called prototype)

```
int Factorial(int n);
```

type of return value

name of function

types of argument

- Function call - used in expression

```
a = x + Factorial(f + g);
```

1. evaluate arguments

2, execute function

3. use return value in expression

Function Definition

- State type, name, types of arguments
  - must match function declaration
  - give name to each argument (doesn't have to match declaration)

```
int Factorial(int n) {
  int i;
  int result = 1;
  for (i = 1; i <= n; i++)
    result *= i;
  return result;
}
```

gives control back to calling function and returns value

Why Declaration?

- Since function definition also includes return and argument types, why is declaration needed?

- Use might be seen before definition. Compiler needs to know return and arg types and number of arguments.

- Definition might be in a different file, written by a different programmer.
  - include a "header" file with function declarations only
  - compile separately, link together to make executable

Example

```
double ValueInDollars(double amount, double rate);
```

declaration

function call (invocation)

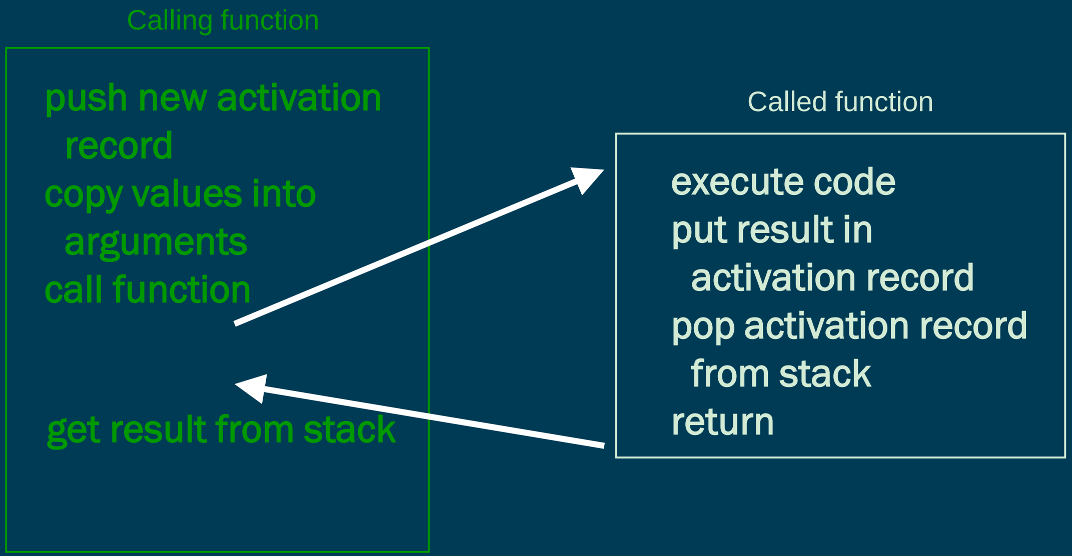
```
main() {
  ...
  dollars = ValueInDollars(francs,DOLLARS_PER_FRANC);
  printf("%f francs equals %f dollars.\n", francs, dollars);
  ...
}
```

definition

```
double ValueInDollars(double amount, double rate) {
  return amount * rate;
}
```

Implementing Functions: Overview

- Activation record
  - information about each function, including arguments and local variables
  - stored on run-time stack


Calling function
push new activation record
copy values into arguments
call function
get result from stack
Called function
execute code
put result in activation record
pop activation record from stack
return

Run-Time Stack

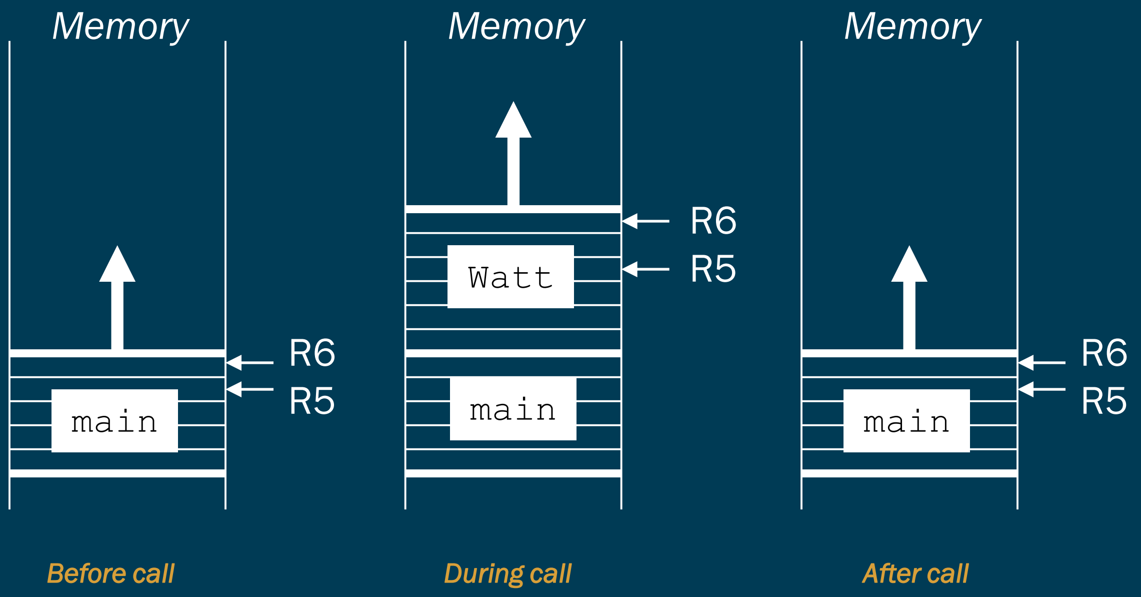
- Recall that local variables are stored on the run-time stack in an *activation record*

- Frame pointer (R5) points to the beginning of a region of activation record that stores local variables for the current function

- When a new function is called, its activation record is pushed on the stack;

- When it returns, its activation record is popped off of the stack.

Run-Time Stack



DATA STRUCTURES:

Pointers

Data Structures

- A data structure is a particular organization of data in memory.
  - We want to group related items together.
  - We want to organize these data bundles in a way that is convenient to program and efficient to execute.
- We will talk about 4 data structures
  - Arrays
  - Pointers
  - struct – directly supported by C
  - linked list – built from struct and dynamic allocation

Pointers and Arrays

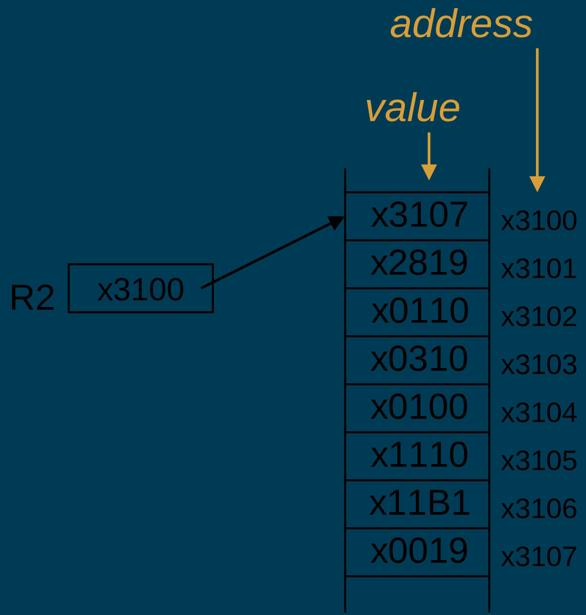
- Pointer
  - Address of a variable in memory
  - Allows us to indirectly access variables
    - in other words, we can talk about its *address* rather than its *value*
- Array
  - A list of values arranged sequentially in memory
  - Example: a list of telephone numbers
  - Expression `a[4]` refers to the 5th element of the array `a`

Address vs. Value

- Sometimes we want to deal with the address of a memory location, rather than the value it contains.

- Recall example from Chapter 6: adding a column of numbers.
  - R2 contains address of first location.
  - Read value, add to sum, and increment R2 until all numbers have been processed.

- R2 is a pointer -- it contains the address of data we're interested in.

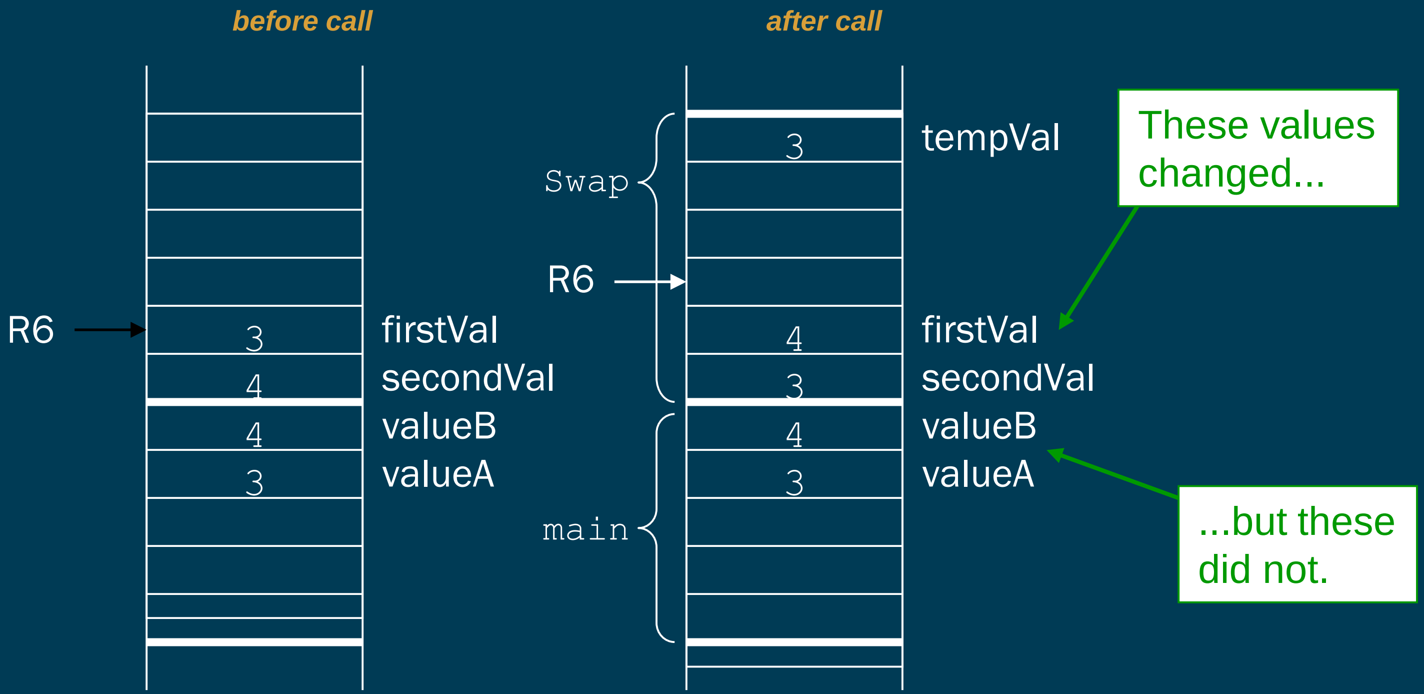


Another Need for Addresses

- Consider the following function that's supposed to swap the values of its arguments.

```
void Swap(int firstVal, int secondVal) {
  int tempVal = firstVal;
  firstVal = secondVal;
  secondVal = tempVal;
}
```

Executing the Swap Function



Swap needs <u>addresses</u> of variables outside its own activation record.

Pointers in C

- Declaration

```
int *p;   /* p is a pointer to an int */
```

- A pointer in C is always a pointer to a particular data type: `int*`, `double*`, `char*`, etc.

- Operators

```
*p
```
-- returns the value pointed to by p

```
&z
```
-- returns the address of variable z

Example

- `int i;`
- `int *ptr;`
- `i = 4;`
- `ptr = &i;`
- `*ptr = *ptr + 1;`

store the value 4 into the memory location associated with i

store the address of i into the memory location associated with ptr

read the contents of memory at the address stored in ptr

store the result into memory at the address stored in ptr

Pointers as Arguments

- Passing a pointer into a function allows the function to read/change memory outside its activation record.

-
```
void NewSwap(int *firstVal, int *secondVal)
{
  int tempVal = *firstVal;
  *firstVal = *secondVal;
  *secondVal = tempVal;
}
```

Arguments are integer pointers. Caller passes addresses of variables that it wants function to change.

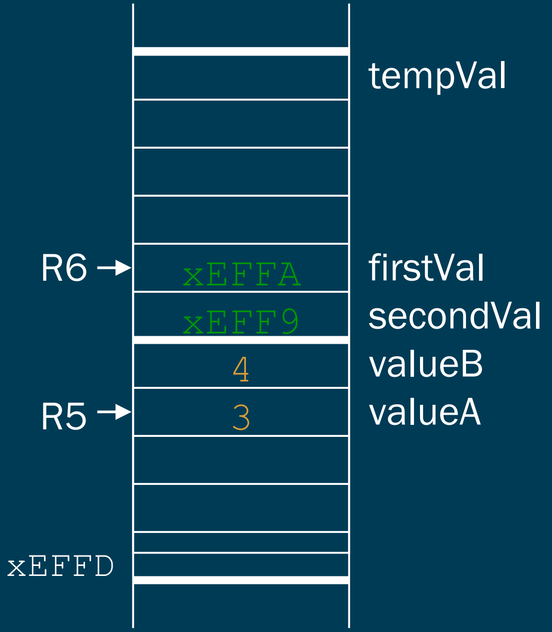
Passing Pointers to a Function

- main() wants to swap the values of valueA and valueB
- passes the addresses to NewSwap:

```
NewSwap(&valueA, &valueB);
```

- Code for passing arguments:

```
ADD R0, R5, #-1 ; addr of valueB
ADD R6, R6, #-1 ; push
STR R0, R6, #0
ADD R0, R5, #0  ; addr of valueA
ADD R6, R6, #-1 ; push
STR R0, R6, #0
```

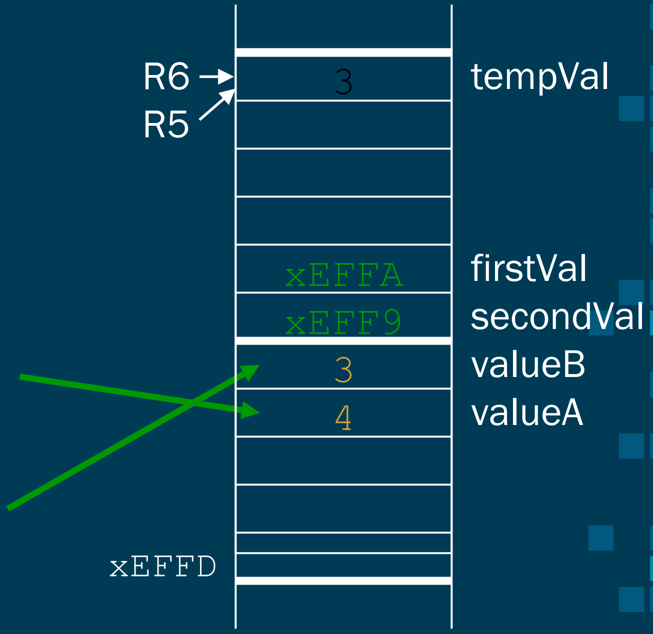


Code Using Pointers

- Inside the NewSwap routine

-
  ```
      ; int tempVal = *firstVal;
      LDR  R0, R5, #4 ; R0=xEFFA
      LDR  R1, R0, #0 ; R1=M[xEFFA]=3
      STR  R1, R5, #4 ; tempVal=3
      ; *firstVal = *secondVal;
      LDR  R1, R5, #5 ; R1=xEFF9
      LDR  R2, R1, #0 ; R1=M[xEFF9]=4
      STR  R2, R0, #0 ; M[xEFFA]=4
      ; *secondVal = tempVal;
  ```

-
  ```
      LDR  R2, R5, #0 ; R2=3
      STR  R2, R1, #0 ; M[xEFF9]=3
  ```



Null Pointer

- Sometimes we want a pointer that points to nothing.
- In other words, we declare a pointer, but we're not ready to actually point to something yet.

```
int *p;
p = NULL;   /* p is a null pointer */
```

- `NULL` is a predefined macro that contains a value that a non-null pointer should never hold.
  - Often, NULL = 0, because Address 0 is not a legal address for most programs on most platforms.

Using Arguments for Results

- Pass address of variable where you want result stored
  - useful for multiple results

    Example:

    return value via pointer

    return status code as function result

- This solves the mystery of why '&' with argument to scanf:
-
```
scanf("%d ", &dataIn);
```

read a decimal integer and store in `dataIn`

Syntax for Pointer Operators

- Declaring a pointer - Either of these work - whitespace doesn't matter. Type of variable is `int*` (integer pointer) `type *var;`
  ```
  type* var;
  ```
- Creating a pointer - Must be applied to a memory object, such as a variable. In other words, `&3` is not allowed.
  ```
  &var
  ```
- Dereferencing - Can be applied to any expression. All of these are legal:

`*var`	contents of mem loc pointed to by var
`**var`	contents of mem loc pointed to by mem loc pointed to by var
`*3`	contents of memory location 3

Example using Pointers

- IntDivide performs both integer division and remainder, returning results via pointers. (Returns –1 if divide by zero.)

```
int IntDivide(int x, int y, int *quoPtr, int *remPtr);

main()
{
   int dividend, divisor;  /* numbers for divide op */
   int quotient, remainer; /* results */
   int error;
   /* ...code for dividend, divisor input removed... */
   error = IntDivide(dividend, divisor,
                     &quotient, &remainder);
   /* ...remaining code removed... */
}
```

C Code for IntDivide

```
int IntDivide(int x, int y, int *quoPtr, int *remPtr)
{
    if (y != 0) {
        *quoPtr = x / y;   /* quotient in *quoPtr */
        *remPtr = x % y;   /* remainder in *remPtr */
        return 0;
    }
    else
        return -1;
}
```

DATA STRUCTURES:

Arrays

Arrays

- How do we allocate a group of memory locations?
  - character string
  - table of numbers
- How about this?
- Not too bad, but…
  - what if there are 100 numbers?
  - how do we write a loop to process each number?
- Fortunately, C gives us a better way -- the *array*.

  `int num[4];`
- Declares a sequence of four integers, referenced by:

  `num[0]`, `num[1]`, `num[2]`, `num[3]`.

```
int num0;
int num1;
int num2;
int num3;
```

Array Syntax

- Declaration

```
type   variable[num_elements];
```

all array elements are of the same type

number of elements must be known at compile-time

- Array Reference

```
variable[index];
```

i-th element of array (starting with zero);
<u>no limit checking</u> at compile-time or run-time

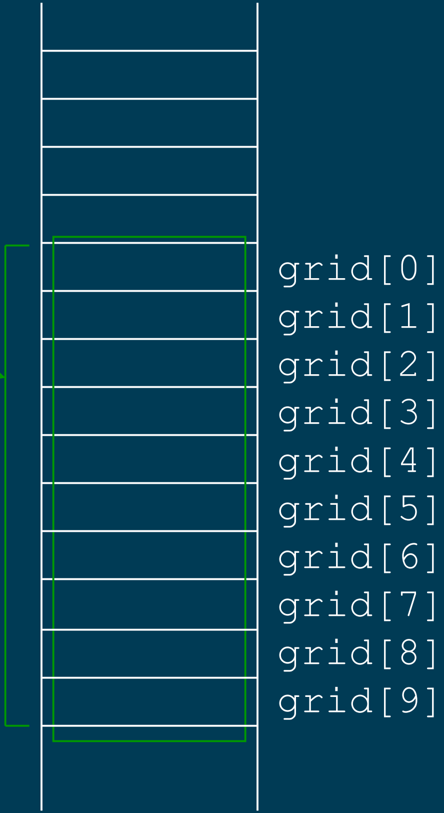
Array as a Local Variable

- Array elements are allocated as part of the activation record.

```
int grid[10];
```

- First element (`grid[0]`) is at lowest address of allocated space.

  If `grid` is first variable allocated, then R5 will point to `grid[9]`.



Passing Arrays as Arguments

- C passes arrays by reference
  - the address of the array (i.e., of the first element) is written to the function's activation record
  - otherwise, would have to copy each element

```
main() {
  int numbers[MAX_NUMS];
  …
  mean = Average(numbers);
  …
}
int Average(int inputValues[MAX_NUMS]) {
  …
  for (index = 0; index < MAX_NUMS; index++)
         sum = sum + indexValues[index];
  return (sum / MAX_NUMS);
}
```

This must be a constant, e.g., **`#define MAX_NUMS 10`**

A String is an Array of Characters

- Allocate space for a string just like any other array:

```
char outputString[16];
```

- Space for string must contain room for terminating zero.
- Special syntax for initializing a string:

```
char outputString[16] = "Result = ";
```

- …which is the same as:

```
outputString[0] = 'R';
outputString[1] = 'e';
outputString[2] = 's';
...
```

I/O with Strings

- Printf and scanf use "%s" format character for string

- Printf -- print characters up to terminating zero

```
    printf("%s", outputString);
```

- Scanf -- read characters until whitespace, store result in string, and terminate with zero

```
    scanf("%s", inputString);
```

Relationship between Arrays & Pointers

- An array name is essentially a pointer to the first element in the array

```
char word[10];
char *cptr;
cptr = word;   /* points to word[0] */
```

- *Difference:* Can change the contents of cptr, as in

```
cptr = cptr + 1;
```

- (The identifier "word" is not a variable.)

Correspondence: Pointers & Arrays

- Given the declarations on the previous page, each line below gives three equivalent expressions:

`cptr`	`word`	`&word[0]`
`(cptr + n)`	`word + n`	`&word[n]`
`*cptr`	`*word`	`word[0]`
`*(cptr + n)`	`*(word + n)`	`word[n]`

Common Pitfalls with Arrays in C

- Overrun array limits
  - There is no checking at run-time or compile-time to see whether reference is within array bounds.

```
int array[10];
int i;
for (i = 0; i <= 10; i++) array[i] = 0;
```

- Declaration with variable size
  - Size of array must be known at compile time.

```
void SomeFunction(int num_elements) {
  int temp[num_elements];
  …
}
```

Pointer Arithmetic

- Address calculations depend on size of elements
  - In our LC-3 code, we've been assuming one word per element.
    - e.g., to find 4th element, we add 4 to base address
  - It's ok, because we've only shown code for int and char, both of which take up one word.
  - If double, we'd have to add 8 to find address of 4th element.
- C does size calculations under the covers, depending on size of item being pointed to:

```
double x[10];
```

allocates 20 words (2 per element)

```
double *y = x;
*(y + 3) = 13;
```

same as x[3] -- base address plus 6

DATA STRUCTURES:

Structs

Structures in C

- A `struct` is a mechanism for grouping together related data items of different types.
  - Recall that an array groups items of a single type.
- Example: We want to represent an airborne aircraft:

```
char flightNum[7];
int altitude;
int longitude;
int latitude;
int heading;
double airSpeed;
```

- We can use a `struct` to group these data together for each plane.

Defining a Struct

- We first need to define a new type for the compiler and tell it what our struct looks like.

```
struct flightType {
  char flightNum[7];  /* max 6 characters */
  int altitude;       /* in meters */
  int longitude;      /* in tenths of degrees */
  int latitude;       /* in tenths of degrees */
  int heading;        /* in tenths of degrees */
   double airSpeed;    /* in km/hr */
};
```

- This tells the compiler how big our struct is and how the different data items ("members") are laid out in memory.
- But it does not allocate any memory.

Declaring and Using a Struct

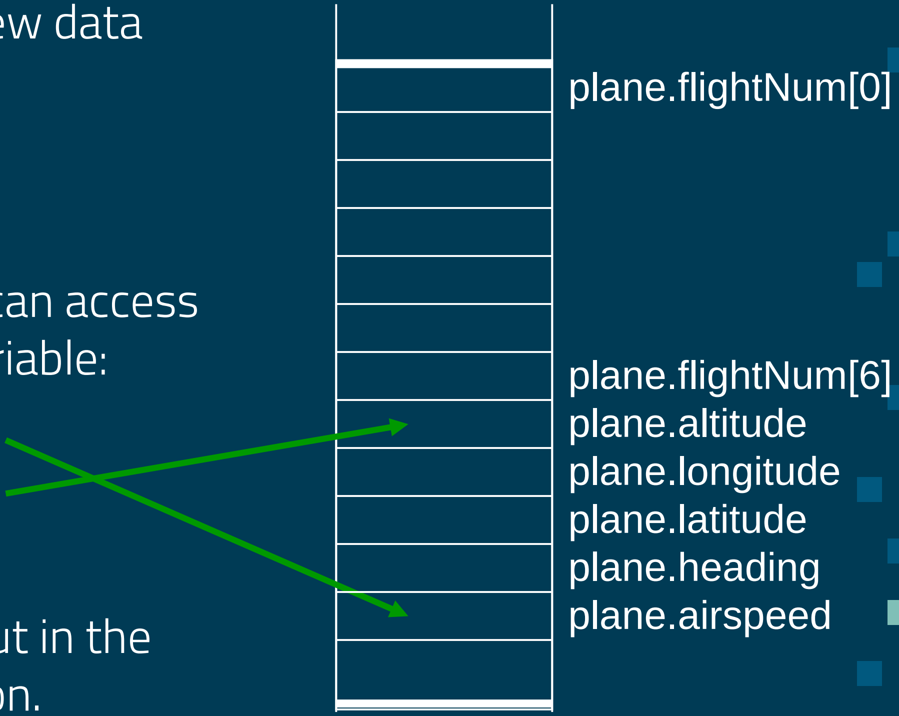
- To allocate memory for a struct, we declare a variable using our new data type.

```
struct flightType plane;
```

- Memory is allocated, and we can access individual members of this variable:

```
plane.airSpeed = 800.0;
plane.altitude = 10000;
```

- A struct's members are laid out in the order specified by the definition.



Defining and Declaring at Once

- You can both define and declare a struct at the same time.

```
struct flightType {
  char flightNum[7];   /* max 6 characters */
  int altitude;        /* in meters */
  int longitude;       /* in tenths of degrees */
  int latitude;        /* in tenths of degrees */
  int heading;         /* in tenths of degrees */
  double airSpeed;     /* in km/hr */
} maverick;
```

- And you can use the flightType name to declare other structs.

```
struct flightType iceMan;
```

typedef

- C provides a way to define a data type by giving a new name to a predefined type.
- Syntax:

```
typedef <type> <name>;
```

- Examples:

```
typedef int Color;
typedef struct flightType WeatherData;
typedef struct ab_type {
  int a;
  double b;
} ABGroup;
```

Using typedef

- This gives us a way to make code more readable by giving application-specific names to types.

```
Color pixels[500];

Flight plane1, plane2;
```

- Typical practice: Put typedef's into a header file, and use type names in main program. If the definition of Color/Flight changes, you might not need to change the code in your main program file.

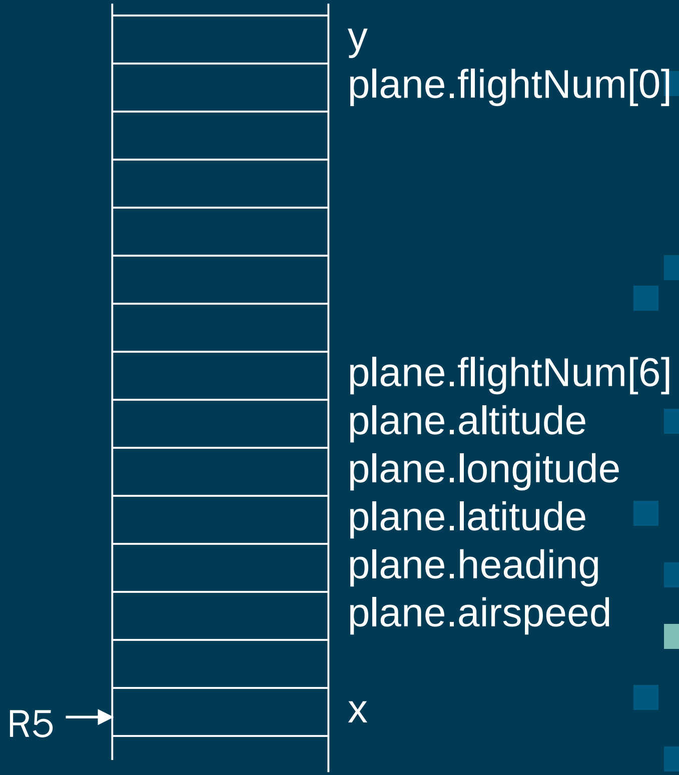
Generating Code for Structs

- Suppose our program starts out like this:

```
int x;
Flight plane;
int y;

plane.altitude = 0;
...
```

- LC-3 code for this assignment:

```
AND  R1, R1, #0
ADD  R0, R5, #-13 ; R0=plane
STR  R1, R0, #7   ; 8th word
```



Array of Structs

- Can declare an array of structs:

```
Flight planes[100];
```

- Each array element is a struct (7 words, in this case).
- To access member of a particular element:

```
planes[34].altitude = 10000;
```

- Because the `[]` and `.` operators are at the same precedence, and both associate left-to-right, this is the same as:

```
(planes[34]).altitude = 10000;
```

Pointer to Struct

- We can declare and create a pointer to a struct:

```
Flight *planePtr;
planePtr = &planes[34];
```

- To access a member of the struct addressed by dayPtr:

```
(*planePtr).altitude = 10000;
```

- Because the . operator has higher precedence than *, this is NOT the same as:

```
*planePtr.altitude = 10000;
```

- C provides special syntax for accessing a struct member through a pointer:

```
planePtr->altitude = 10000;
```

Passing Structs as Arguments

- Unlike an array, a struct is always passed by value into a function.
  - This means the struct members are copied to the function’s activation record, and changes inside the function are not reflected in the calling routine’s copy.
- Most of the time, you’ll want to pass a pointer to a struct.

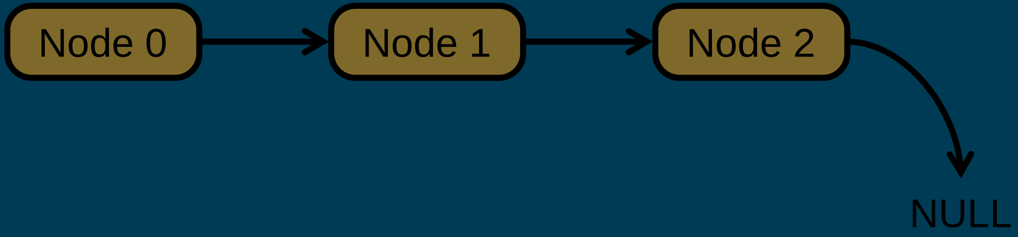
```
int Collide(Flight *planeA, Flight *planeB) {
  if (planeA->altitude == planeB->altitude) {
      ...
  }
  else
    return 0;
}
```

DATA STRUCTURES:

Linked Lists

The Linked List Data Structure

- A linked list is an ordered collection of nodes each of which contains some data, connected using pointers.
    - Each node points to the next node in the list.
    - The first node in the list is called the head.
    - The last node in the list is called the tail.



Linked List vs. Array

- A linked list can only be accessed sequentially.
- To find the 5th element, for instance, you must start from the head and follow the links through four other nodes.
- Advantages of linked list:
  - Dynamic size
  - Easy to add additional nodes as needed
  - Easy to add or remove nodes from the middle of the list (just add or redirect links)
- Advantage of array:
  - Can easily and quickly access arbitrary elements

Example: Car Lot

- Create an inventory database for a used car lot. Support the following actions:
  - Search the database for a particular vehicle.
  - Add a new car to the database.
  - Delete a car from the database.

- The database must remain sorted by vehicle ID.
- Since we don't know how many cars might be on the lot at one time, we choose a linked list representation.

Car data structure

- Each car has the following characteristics: vehicle ID, make, model, year, mileage, cost. Because it’s a linked list, we also need a pointer to the next node in the list:

```
typedef struct carType Car;

struct carType {
  int vehicleID;
  char make[20];
  char model[20];
  int year;
  int mileage;
  double cost;
  Car *next; /* ptr to next car in list */
}
```

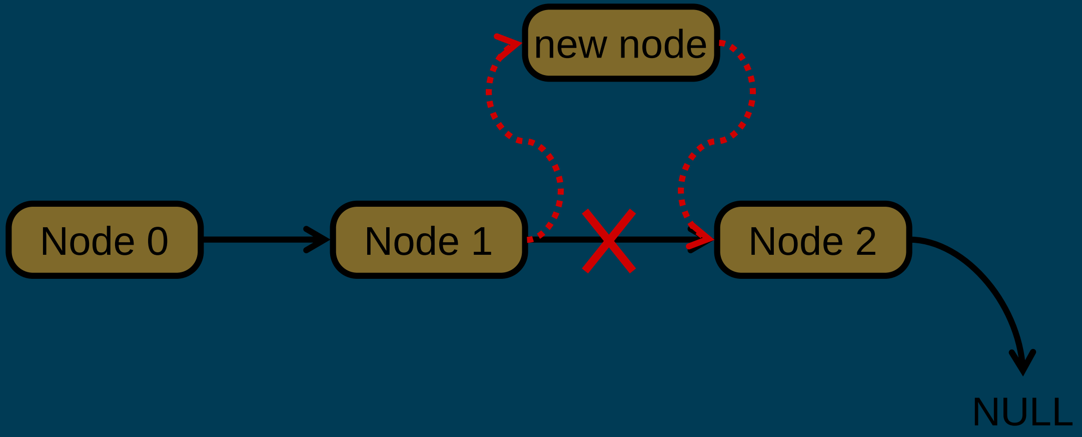
Scanning the List

- Searching, adding, and deleting all require us to find a particular node in the list. We scan the list until we find a node whose ID is >= the one we're looking for.

```
Car *ScanList(Car *head, int searchID) {
  Car *previous, *current;
  previous = head;
  current = head->next;
  /* Traverse until ID >= searchID */
  while((current!=NULL) && (current->vehicleID < searchID)) {
    previous = current;
    current = current->next;
  }
  return previous;
}
```

Adding a Node

- Create a new node with the proper info. Find the node (if any) with a greater vehicleID. “Splice” the new node into the list:

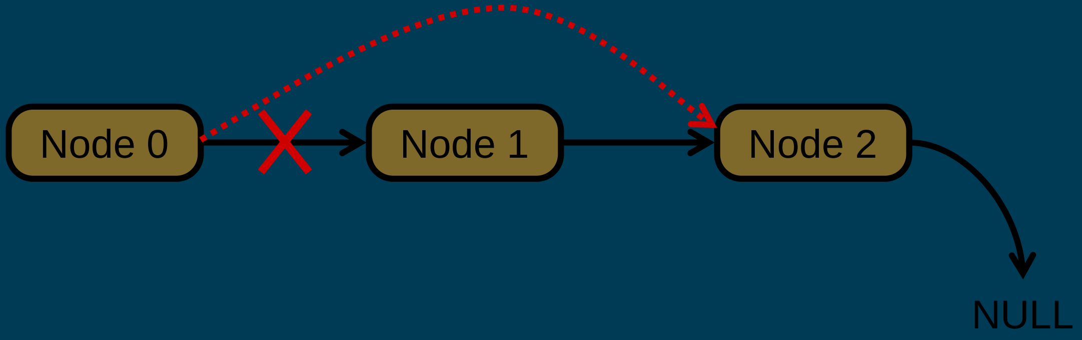


Excerpts from Code to Add a Node

```
newNode = (Car*) malloc(sizeof(Car));
/* initialize node with new car info */
...
prevNode = ScanList(head, newNode->vehicleID);
nextNode = prevNode->next;

if ((nextNode == NULL)
    || (nextNode->vehicleID != newNode->vehicleID)) {
  prevNode->next = newNode;
  newNode->next = nextNode;
}
else {
  printf("Car already exists in database.");
  free(newNode);
}
```

Deleting a Node

- Find the node that points to the desired node.
- Redirect that node's pointer to the next node (or NULL).
- Free the deleted node's memory.



Excerpts from Code to Delete a Node

```
printf(“Enter vehicle ID of car to delete:\n”);
scanf(“%d”, vehicleID);

prevNode = ScanList(head, vehicleID);
delNode = prevNode->next;

if ((delNode != NULL)
     && (delNode->vehicleID == vehicleID))
  prevNode->next = delNode->next;
  free(delNode);
}
else {
  printf(“Vehicle not found in database.\n”);
}
```

Building on Linked Lists

- The linked list is a fundamental data structure.
  - Dynamic
  - Easy to add and delete nodes

- The concepts described here will be helpful when learning about more elaborate data structures:
  - Trees
  - Hash Tables
  - Directed Acyclic Graphs
  - …

DATA STRUCTURES:

Dynamic Allocation

Dynamic Allocation

- Suppose we want our weather program to handle a variable number of planes – as many as the user wants to enter.
  - We can't allocate an array, because we don't know the maximum number of planes that might be required.
  - Even if we do know the maximum number, it might be wasteful to allocate that much memory because most of the time only a few planes' worth of data is needed.

- Solution:
  Allocate storage for data dynamically, as needed.

malloc

- The Standard C Library provides a function for allocating memory at run-time: malloc.

```
void *malloc(int numBytes);
```

- It returns a generic pointer (`void*`) to a contiguous region of memory of the requested size (in bytes).
- The bytes are allocated from a region in memory called the heap.
  - The run-time system keeps track of chunks of memory from the heap that have been allocated.

Using malloc

- To use malloc, we need to know how many bytes to allocate. The `sizeof` operator asks the compiler to calculate the size of a particular type.

```
planes = malloc(n * sizeof(Flight));
```

- We also need to change the type of the return value to the proper kind of pointer – this is called "casting."

```
planes =
   (Flight*) malloc(n* sizeof(Flight));
```

Example

```
int airbornePlanes;
Flight *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

planes = (Flight*) malloc(sizeof(Flight) * airbornePlanes);
if (planes == NULL) {
  printf("Error in allocating the data array.\n");
  ...
}
planes[0].altitude = ...
```

If allocation fails, malloc returns NULL.

Note: Can use array notation or pointer notation.

free

- Once the data is no longer needed, it should be released back into the heap for later use.

- This is done using the free function, passing it the same address that was returned by malloc.

```
void free(void*);
```

- If allocated data is not freed, the program might run out of heap memory and be unable to continue.

RECURSIVE FUNCTIONS

What is Recursion?

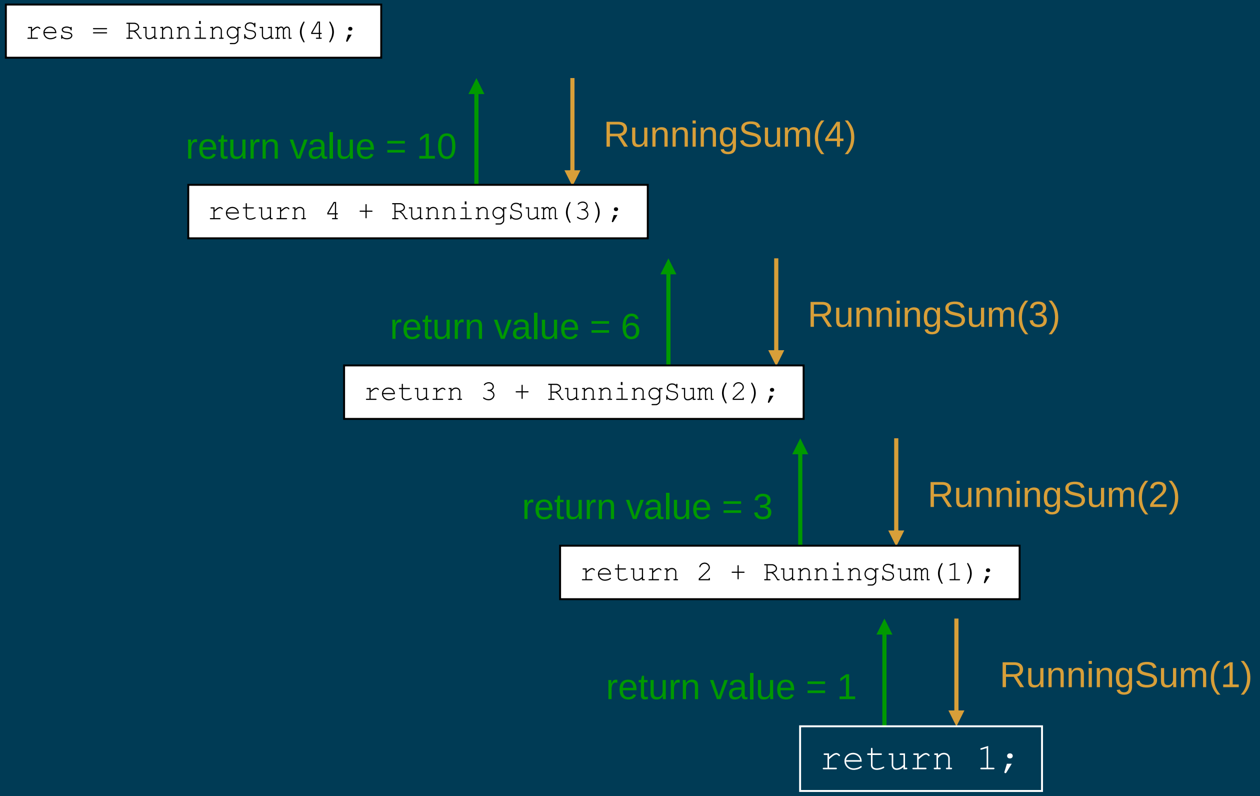
- A recursive function is one that solves its task by calling itself on smaller pieces of data.
  - Similar to recurrence function in mathematics.
  - Like iteration -- can be used interchangeably; sometimes recursion results in a simpler solution.
- Example: Running sum ($\sum_{1}^{n} i$)

Mathematical Definition:
RunningSum(1) = 1
RunningSum(n) =
n + RunningSum(n-1)

Recursive Function:

```
int RunningSum(int n) {
  if (n == 1)
    return 1;
  else
    return n + RunningSum(n-1);
}
```

Executing RunningSum



High-Level Example: Binary Search

- Given a sorted set of exams, in alphabetical order, find the exam for a particular student.

1. Look at the exam halfway through the pile.
2. If it matches the name, we're done; if it does not match, then...
3. If the name is greater (alphabetically), then search the upper half of the stack.
4. If the name is less than the halfway point, then search the lower half of the stack.

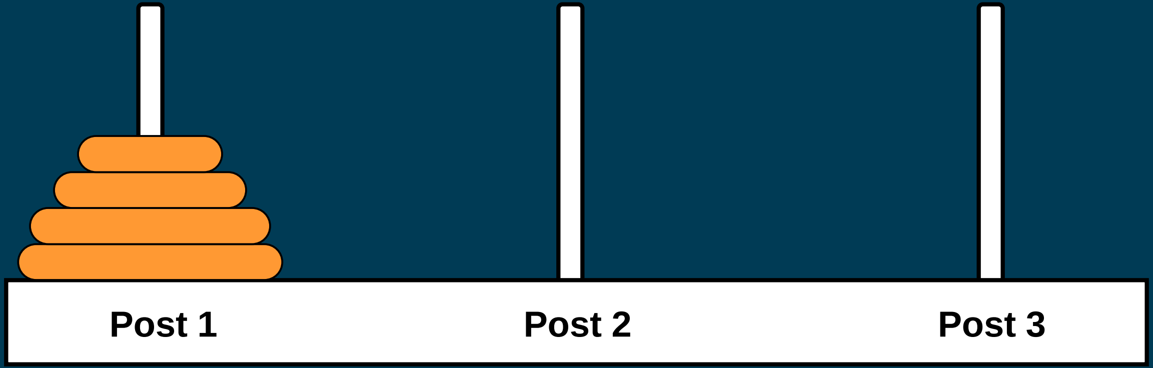
Binary Search: Pseudocode

- Pseudocode is a way to describe algorithms without completely coding them in C.

```
FindExam(studentName, start, end){
  halfwayPoint = (end + start)/2;
  if (end < start)
    ExamNotFound();  /* exam not in stack */
  else if (studentName == NameOfExam(halfwayPoint))
    ExamFound(halfwayPoint); /* found exam! */
  else if (studentName < NameOfExam(halfwayPoint))
    /* search lower half */
    FindExam(studentName, start, halfwayPoint - 1);
  else /* search upper half */
    FindExam(studentName, halfwayPoint + 1, end);
}
```

High-Level Example: Towers of Hanoi

- Task: Move all disks from current post to another post.



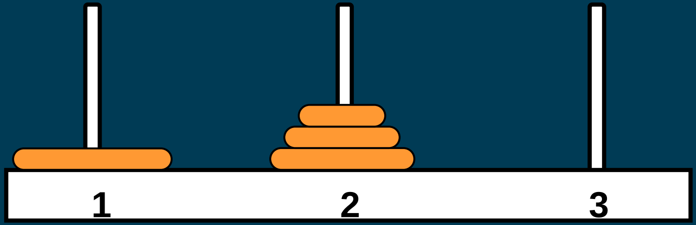
Rules:

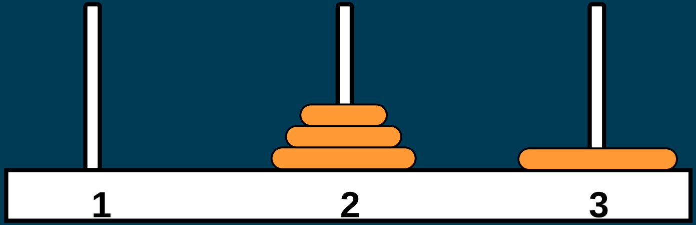
1. Can only move one disk at a time.
2. A larger disk can never be placed on top of a smaller disk.
3. May use third post for temporary storage.

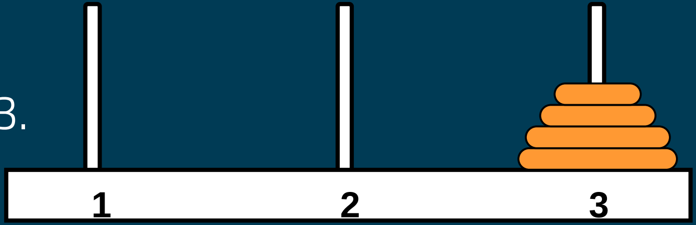
Task Decomposition

- Suppose disks start on Post 1, and target is Post 3.

1. Move top n-1 disks to Post 2.
2. Move largest disk to Post 3.
3. Move n-1 disks from Post 2 to Post 3.







Task Decomposition (cont.)

- Task 1 is really the same problem, with fewer disks and a different target post.
  - "Move n-1 disks from Post 1 to Post 2."
- And Task 3 is also the same problem, with fewer disks and different starting and target posts.
  - "Move n-1 disks from Post 2 to Post 3."
- So this is a recursive algorithm.
  - The terminal case is moving the smallest disk -- can move directly without using third post.
  - Number disks from 1 (smallest) to n (largest).

Towers of Hanoi: Pseudocode

```
MoveDisk(diskNumber, startPost, endPost, midPost) {
  if (diskNumber > 1) {
    /* Move top n-1 disks to mid post */
    MoveDisk(diskNumber-1, startPost, midPost, endPost);

    printf("Move disk number %d from %d to %d.\n",
           diskNumber, startPost, endPost);

    /* Move n-1 disks from mid post to end post */
    MoveDisk(diskNumber-1, midPost, endPost, startPost);
  }
  else
    printf("Move disk number 1 from %d to %d.\n",
           startPost, endPost);
}
```

Detailed Example: Fibonacci Numbers

- Mathematical Definition:

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = 1$$
$$f(0) = 1$$

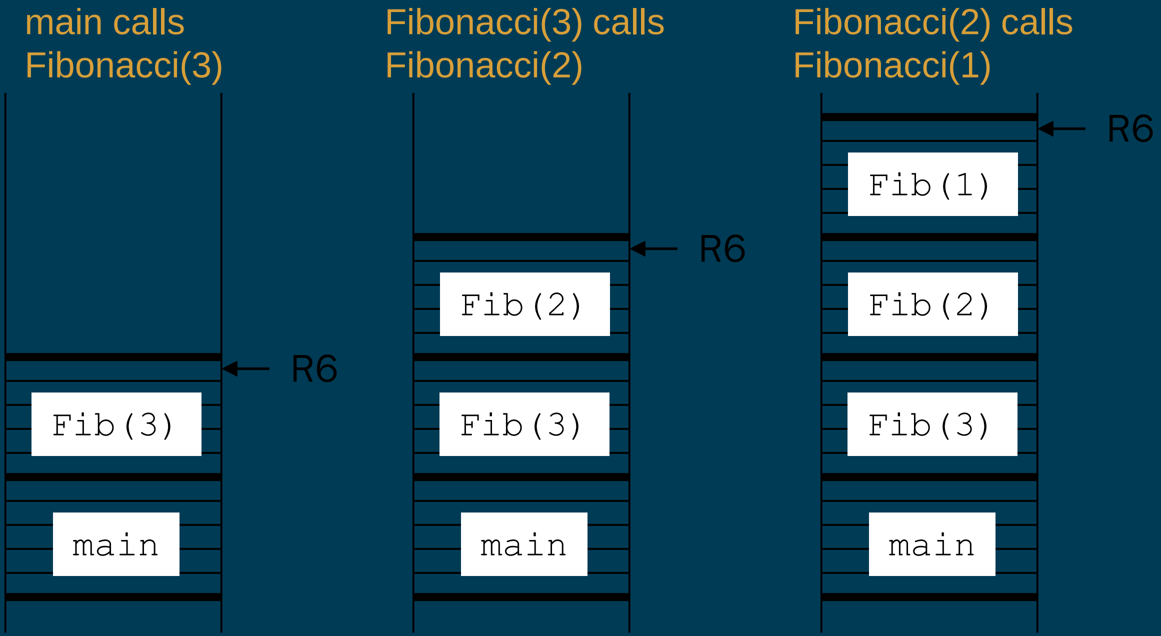
- In other words, the n-th Fibonacci number is the sum of the previous two Fibonacci numbers.

Fibonacci: C Code

```
int Fibonacci(int n)
{
  if ((n == 0) || (n == 1))
    return 1;
  else
    return Fibonacci(n-1) + Fibonacci(n-2);
}
```

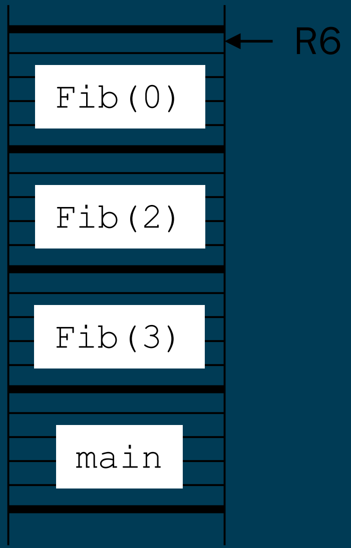
Activation Records

- Whenever Fibonacci is invoked, a new activation record is pushed onto the stack.

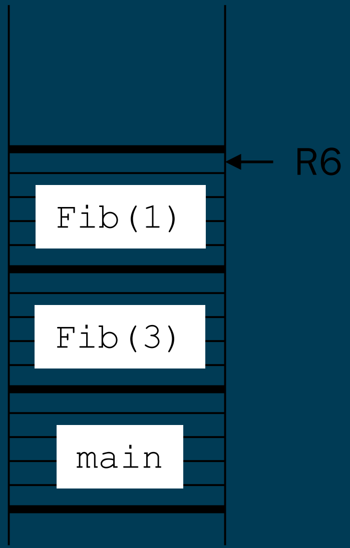


Activation Records (cont.)

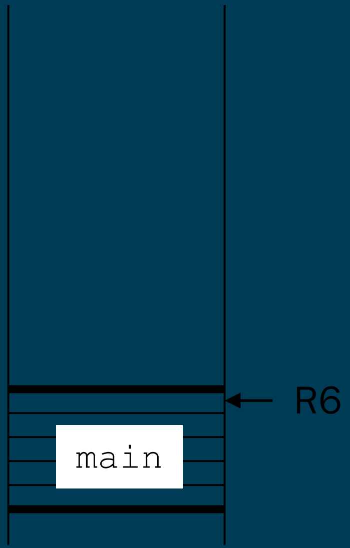
Fibonacci(1) returns, Fibonacci(2) calls Fibonacci(0)



Fibonacci(2) returns, Fibonacci(3) calls Fibonacci(1)



Fibonacci(3) returns



Tracing the Function Calls

- If we are debugging this program, we might want to trace all the calls of Fibonacci.
  - Note: A trace will also contain the arguments passed into the function.
- For Fibonacci(3), a trace looks like:

```
Fibonacci(3)
 Fibonacci(2)
  Fibonacci(1)
  Fibonacci(0)
 Fibonacci(1)
```

- What would trace of Fibonacci(4) look like?

A Final C Example: Printing an Integer

- Recursively converts an integer as a string of ASCII characters.
  - If integer <10, convert to char and print.
  - Else, call self on first (n-1) digits and then print last digit.

```
void IntToAscii(int num) {
  int prefix, currDigit;
  if (num < 10)
    putchar(num + '0');  /* prints single char */
  else {
    prefix = num / 10;   /* shift right one digit */
    IntToAscii(prefix);  /* print shifted num */
    /* then print shifted digit */
    currDigit = num % 10;
    putchar(currDigit + '0');
  }
}
```

Trace of IntToAscii

- Calling IntToAscii with parameter 12345:

```
IntToAscii(12345)
 IntToAscii(1234)
  IntToAscii(123)
   IntToAscii(12)
    IntToAscii(1)
    putchar('1')
   putchar('2')
  putchar('3')
 putchar('4')
putchar('5')
```

I/O IN C

Standard C Library

- I/O commands are not included as part of the C language.
- Instead, they are part of the Standard C Library.
  - A collection of functions and macros that must be implemented by any ANSI standard implementation.
  - Automatically linked with every executable.
  - Implementation depends on processor, operating system, etc., but interface is standard.
- Since they are not part of the language, compiler must be told about function interfaces.
- Standard header files are provided, which contain declarations of functions, variables, etc.

Basic I/O Functions

- The standard I/O functions are declared in the <stdio.h> header file.

Function	*Description*
`putchar`	Displays an ASCII character to the screen.
`getchar`	Reads an ASCII character from the keyboard.
`printf`	Displays a formatted string,
`scanf`	Reads a formatted string.
`fopen`	Open/create a file for I/O.
`fprintf`	Writes a formatted string to a file.
`fscanf`	Reads a formatted string from a file.

Text Streams

- All character-based I/O in C is performed on text streams.
- A stream is a sequence of ASCII characters, such as:
  - the sequence of ASCII characters printed to the monitor by a single program
  - the sequence of ASCII characters entered by the user during a single program
  - the sequence of ASCII characters in a single file
- Characters are processed in the order in which they were added to the stream.
  - E.g., a program sees input characters in the same order as the user typed them.
- Standard input stream (keyboard) is called stdin.
- Standard output stream (monitor) is called stdout.

Character I/O

- `putchar(c)` Adds one ASCII character (c) to stdout.
- `getchar()` Reads one ASCII character from stdin.

- These functions deal with "raw" ASCII characters; no type conversion is performed.

```
char c = 'h';
...
putchar(c);
putchar('h');
putchar(104);
```

Each of these calls prints 'h' to the screen.

Buffered I/O

- In many systems, characters are buffered in memory during an I/O operation.
  - Conceptually, each I/O stream has its own buffer.

- Keyboard input stream
  - Characters are added to the buffer only when the newline character (i.e., the "Enter" key) is pressed.
  - This allows user to correct input before confirming with Enter.

- Output stream
  - Characters are not flushed to the output device until the newline character is added.

Input Buffering

```
printf("Input character 1:\n");
inChar1 = getchar();

printf("Input character 2:\n");
inChar2 = getchar();
```

- After seeing the first prompt and typing a single character, nothing happens.
- Expect to see the second prompt, but character not added to stdin until Enter is pressed.
- When Enter is pressed, newline is added to stream and is consumed by second getchar(), so `inChar2` is set to `'\n'`.

Output Buffering

```
putchar('a');
/* generate some delay */
for (i=0; i<DELAY; i++) sum += i;

putchar('b');
putchar('\n');
```

- User doesn't see any character output until after the delay.
- `'a'` is added to the stream before the delay, but the stream is not flushed (displayed) until `'\n'` is added.

Formatted I/O

- Printf and scanf allow conversion between ASCII representations and internal data types.
- Format string contains text to be read/written, and formatting characters that describe how data is to be read/written.

`%d`	signed decimal integer
`%f`	signed decimal floating-point number
`%x`	unsigned hexadecimal number
`%b`	unsigned binary number
`%c`	ASCII character
`%s`	ASCII string

Special Character Literals

- Certain characters cannot be easily represented by a single keystroke, because they
  - correspond to whitespace (newline, tab, backspace, ...)
  - are used as delimiters for other literals (quote, double quote, ...)
- These are represented by the following sequences:

`\n`	newline
`\t`	tab
`\b`	backspace
`\\`	backslash
`\'`	single quote
`\"`	double quote
`\0`*nnn*	ASCII code *nnn* (in octal)
`\x`*nnn*	ASCII code *nnn* (in hex)

printf

- Prints its first argument (format string) to stdout with all formatting characters replaced by the ASCII representation of the corresponding data argument.

```
int a = 100;
int b = 65;
char c = 'z';
char banner[10] = "Hola!";
double pi = 3.14159;

printf("The variable 'a' decimal: %d\n", a);
printf("The variable 'a' hex: %x\n", a);
printf("The variable 'a' binary: %b\n", a);
printf("'a' plus 'b' as character: %c\n", a+b);
printf("A char %c.\t A string %s\n A float %f\n",
       c, banner, pi);
```

Missing Data Arguments

- What happens when you don't provide a data argument for every formatting character?

```
printf("The value of nothing is %d\n");
```

- `%d` will convert and print whatever is on the stack in the position where it expects the first argument.
- In other words, <u>something</u> will be printed, but it will be a garbage value as far as our program is concerned.

scanf

- Reads ASCII characters from stdin matching characters to its first argument (format string), converting character sequences according to any formatting characters, and storing the converted values to the addresses specified by its data pointer arguments.

```
char name[100];
int bMonth, bDay, bYear;
double gpa;

scanf("%s %d/%d/%d %lf",
      name, &bMonth, &bDay, &bYear, &gpa);
```

scanf Conversion

- For each data conversion, scanf will skip whitespace characters and then read ASCII characters until it encounters the first character that should NOT be included in the converted value.

`%d`	Reads until first non-digit.
`%x`	Reads until first non-digit (in hex).
`%s`	Reads until first whitespace character.

- Literals in format string must match literals in the input stream.
- Data arguments must be pointers, because scanf stores the converted value to that memory address.

scanf Return Value

- The scanf function returns an integer, which indicates the number of successful conversions performed.
  - This lets the program check whether the input stream was in the proper format.
- Example:

```
scanf("%s %d/%d/%d %lf",
      name, &bMonth, &bDay, &bYear, &gpa);
```

Input Stream *Return Value*

```
Mudd 02/16/69 3.02        5
Muss 02 16 69 3.02        2
```

↑

Doesn't match literal '/', so scanf quits after second conversion.

Bad scanf Arguments

- Two problems with scanf data arguments
  - 1. Not a pointer

    ```
    int n = 0;
    scanf("%d", n);
    ```

    - Will use the value of the argument as an address.
  - 2. Missing data argument

    ```
    scanf("%d");
    ```

    - Will get address from stack, where it expects to find first data argument.
- If you're lucky, program will crash because of trying to modify a restricted memory location (e.g., location 0).
- Otherwise, your program will just modify an arbitrary memory location, which can cause very unpredictable behavior.

Variable Argument Lists

- The number of arguments in a printf or scanf call depends on the number of data items being read or written.
- Declaration of printf (from stdio.h):

```
int printf(const char*, ...);
```

- Parameters pushed onto stack from right to left.
- This stack-based calling convention allows for a variable number of arguments, and fixed arguments (which are named first) are always the same offset from the frame ptr.

File I/O

- For our purposes, a file is a sequence of ASCII characters stored on some device.
  - Allows us to process large amounts of data without having to type it in each time or read it all on the screen as it scrolls by.
- Each file is associated with a stream.
  - May be input stream or output stream (or both!).
- The type of a stream is a "file pointer", declared as:

```
FILE *infile;
```

- The `FILE` type is defined in <stdio.h>.

fopen

- The fopen function associates a physical file with a stream.

```
FILE *fopen(char* name, char* mode);
```

- First argument: `name`
  - The name of the physical file, or how to locate it on the storage device. This may be dependent on the underlying operating system.
- Second argument: `mode`
  - How the file will be used:
    `"r"` -- read from the file
    `"w"` -- write, starting at the beginning of the file
    `"a"` -- write, starting at the end of the file (append)

fprintf and fscanf

- Once a file is opened, it can be read or written using `fscanf()` and `fprintf()`, respectively.

- These are just like `scanf()` and `printf()`, except an additional argument specifies a file pointer.

```
fprintf(outfile, "The answer is %d\n", x);

fscanf(infile, "%s %d/%d/%d %lf",
      name, &bMonth, &bDay, &bYear, &gpa);
```