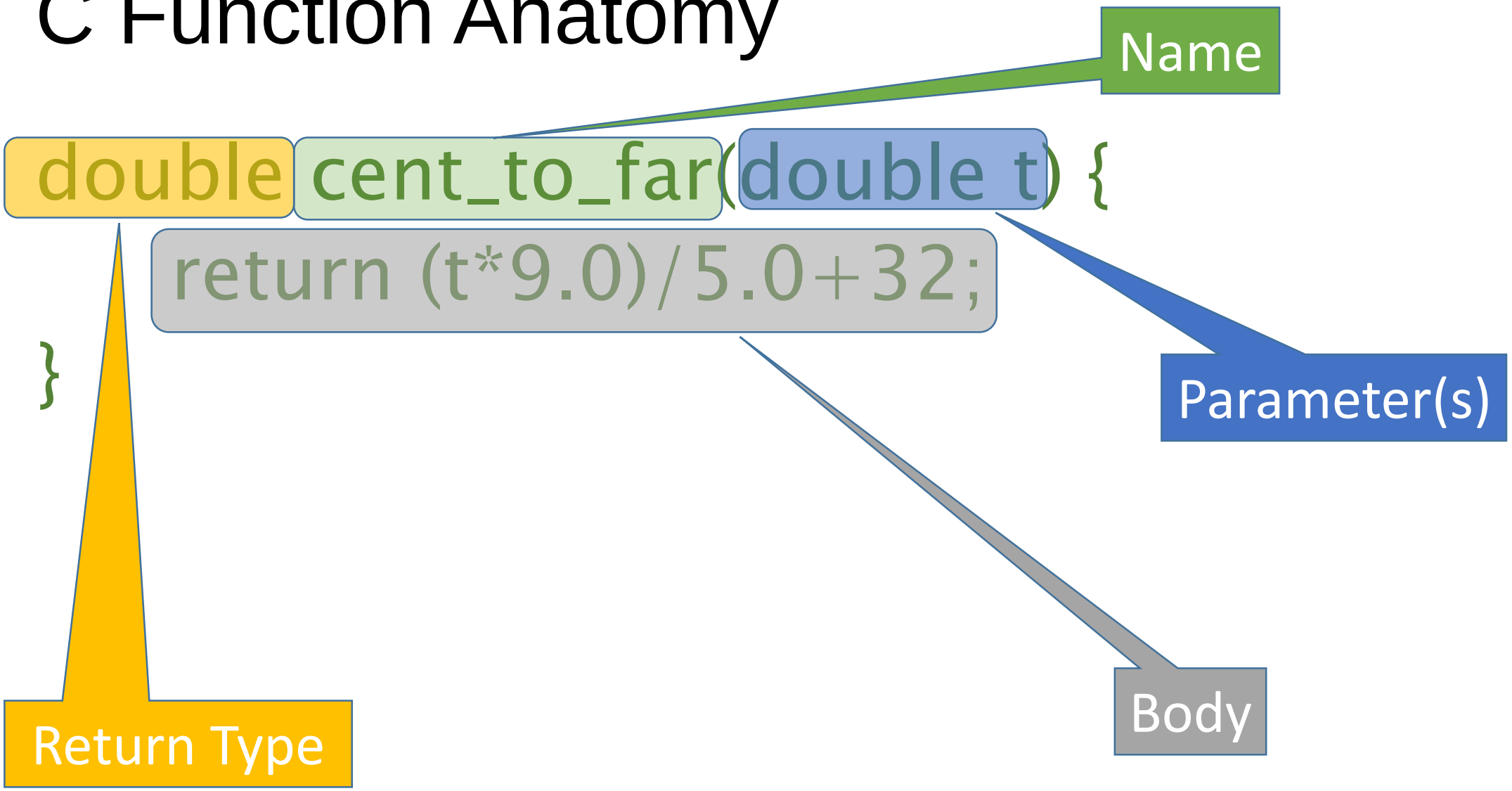
Functions

The Basis of C

C Function Anatomy



Function Invocation

- Invoke a function in a C expression by specifying:

function_name (*argument_expression_list*)

- *function_name* : name of a previously declared function
- *argument_expression_list* : comma separated list of expressions to be used as arguments
- When a function is invoked, C will….
    1. evaluate argument expressions,
    2. Copy argument values into parameters
    3. invoke the specified function (run it's body)
    4. conceptually replace the function invocation with the returned value

```
float warmer=cent_to_far(ctemp*1.10);
```

Functions for Abstraction

- Function Prototype: float cent_to_far(float t)
- Function Behavior: Convert Centigrade to Fahrenheit
- Function Embodiment: WHO CARES?
  - As long as it works, we can ignore the embodiment!
  - Code it once, test it, and then use it lots of times!

The “main” function

- Every C program must have a “main” function
- When C program is run, the OS invokes the main function
- When the main function returns, program ends
- Return value “int”
  - Return value 0 indicates program worked OK
  - Return value other than 0 indicates program failed
- main function arguments – stay tuned
- main function may invoke lower level functions
- For now: int main() { … ; return 0; }

Arguments to main

- When the operating system calls the “main” function, it:
- parses the command line, splitting at “white space” (blanks, tabs)
  >./convertTemp 21.3 F C
- Counts the number of blank delimited words: argc=4
- Creates an array of “words” or strings (really an array of arrays)

argv[0]	“./convertTemp”
argv[1]	“21.3”
argv[2]	“F”
argv[3]	“C”

Size=4

Parameters of main

- Typically use argc and argv as the names of the parameters to main

```
int main(int argc, char **argv) { …
```

- argc argument count – the size of the argv array
- argv argument value (or vector) - a list of strings
  - Each string is an array of characters
  - So argv is an array of arrays of characters
  - Almost (char[])[] argv – argv is an array of (array of characters)
    - (note: not char[][] argv – not a two dimensional array of characters!)
  - Can reference each string as argv[i]
  - Can reference individual letters as argv[i][j] really (argv[i])[j]

Function Prototype Declare

- Function Prototype: float freezingPoint(char scale);
- Prototype consists of return type, function name, & parameter list
- We can declare a function by specifying the prototype;
  - Followed by a semi-colon
- Still need full function definition somewhere else
- Enables “Right Side Up” coding
  - Function Prototypes at the top of the file
  - Function definition for top level function (main) next
  - Then function definitions for lower level functions

Function Internals (abstractly)

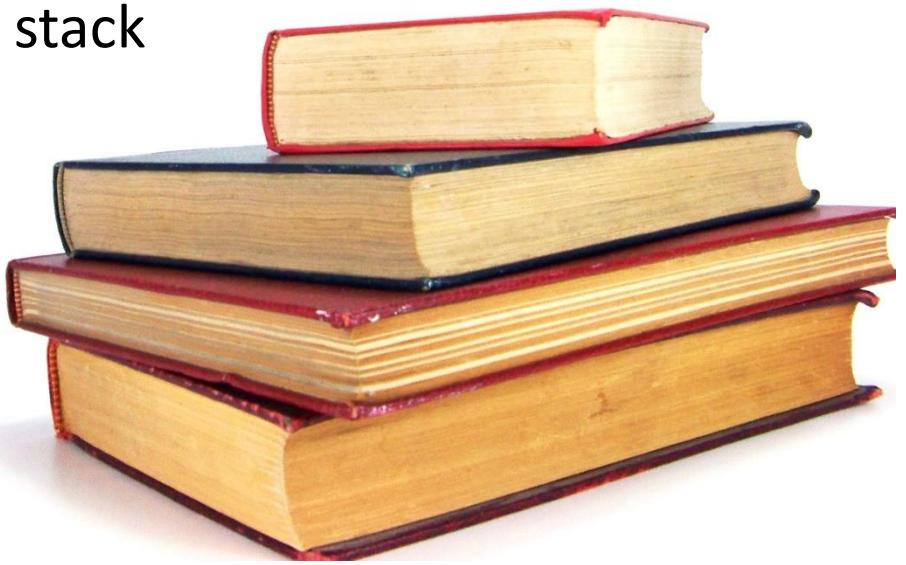
Activation Record

- When a function is called, an “activation record” is created
- Activation records hold:
  - Location in C code where function is invoked and will return
  - Function being invoked
  - Copies of the argument values for this invocation
  - Space for function variable values
  - Space for a return value
- After the function returns, the activation record is deleted

Inv	Fn	args	vars	Ret
main.7	myfn	x=7	a=3,b=4	?

Activation Records are kept in a stack

- "Current" function on top of stack
- When a function is invoked,
  - add its activation record to the top of the stack
- After a function returns,
  - remove it's activation record from the top of the stack



Example Call Stack

```
1. int addem(int x, int y);
2. int main() {
3.     int a=addem(3,4);
4.     a=addem(a,4);
5.     return 0;
6. }
7. int addem(int x, int y) { return x+y;}
```

Inv	Fn	args	vars	Ret
OS	main		a=	

Example Call Stack

```
1. int addem(int x, int y);
2. int main() {
3.      int a=addem(3,4);
4.      a=addem(a,4);
5.      return 0;
6. }
7. int addem(int x, int y) { return x+y;}
```

Inv	Fn	args	vars	Ret
3.10	addem	x=3,y=4		

Inv	Fn	args	vars	Ret
OS	main		a=	

Example Call Stack

```
1. int addem(int x, int y);
2. int main() {
3.      int a=addem(3,4);
4.      a=addem(a,4);
5.      return 0;
6. }
7. int addem(int x, int y) { return x+y;}
```

Inv	Fn	args	vars	Ret
3.10	addem	x=3,y=4		7

Inv	Fn	args	vars	Ret
OS	main		a=	

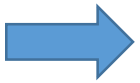
Example Call Stack

```
1. int addem(int x, int y);
2. int main() {
3.      int a=addem(3,4);
4.      a=addem(a,4);
5.      return 0;
6. }
7. int addem(int x, int y) { return x+y;}
```

Inv	Fn	args	vars	Ret
OS	main		a=7	

Example Call Stack

```
1. int addem(int x, int y);
2. int main() {
3.      int a=addem(3,4);
4.      a=addem(a,4);
5.      return 0;
6. }
7. int addem(int x, int y) { return x+y;}
```



Inv	Fn	args	vars	Ret
4.7	addem	x=7,y=4		

Inv	Fn	args	vars	Ret
OS	main		a=7	

Example Call Stack

```
1. int addem(int x, int y);
2. int main() {
3.      int a=addem(3,4);
4.      a=addem(a,4);
5.      return 0;
6. }
7. int addem(int x, int y) { return x+y;}
```

Inv	Fn	args	vars	Ret
4.7	addem	x=7,y=4		11

Inv	Fn	args	vars	Ret
OS	main		a=7	

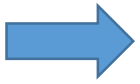
Example Call Stack

```
1. int addem(int x, int y);
2. int main() {
3.      int a=addem(3,4);
4.      a=addem(a,4);
5.      return 0;
6. }
7. int addem(int x, int y) { return x+y;}
```

Inv	Fn	args	vars	Ret
OS	main		a=11	

Example Call Stack

```
1. int addem(int x, int y);
2. int main() {
3.      int a=addem(3,4);
4.      a=addem(a,4);
5.      return 0;
6. }
7. int addem(int x, int y) { return x+y;}
```



Inv	Fn	args	vars	Ret
OS	main		a=11	0

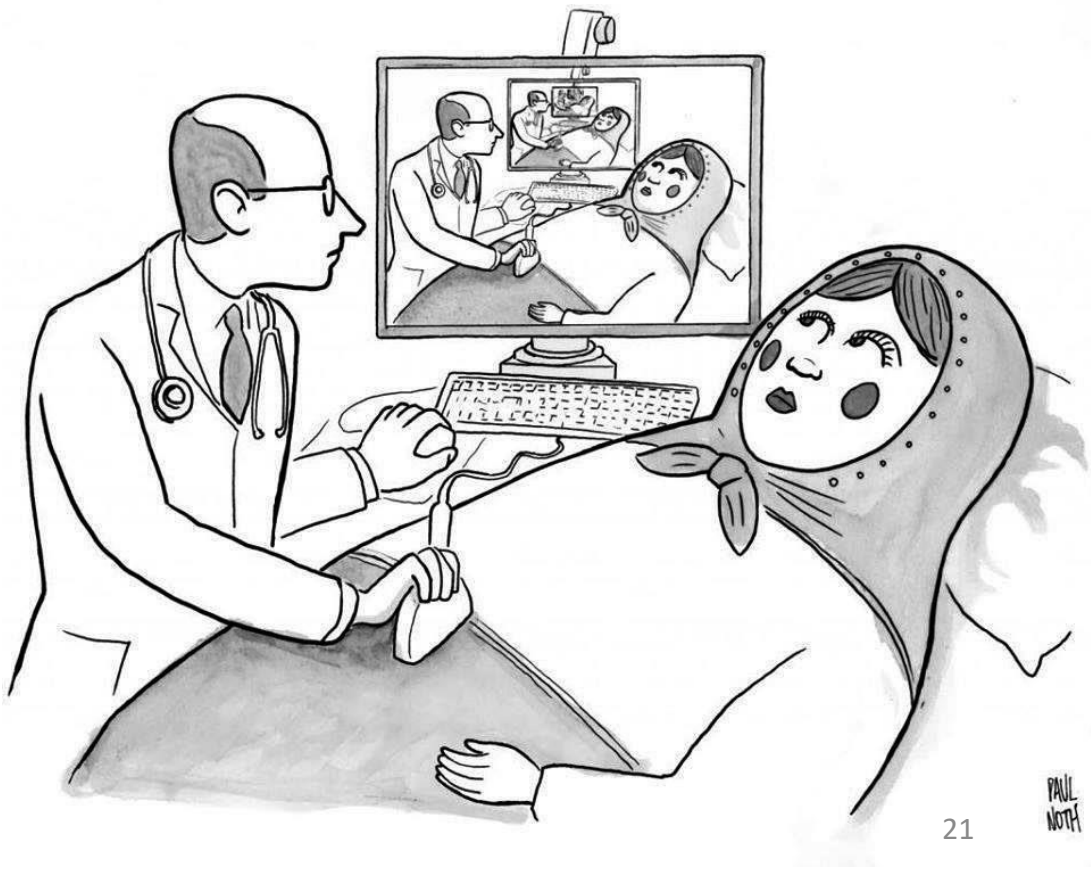
Notes on Call Stacks

- Arguments are passed by value
  - Can’t update callers value!
- GDB “where” command prints call stack
- Function variable management is useful
  - Don’t need to worry about caller’s environment!
  - Don’t need to worry about previous invocations
  - Enables “recursion”

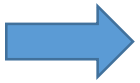
Recursive Function

- A “recursive” function is a function which calls itself
- For example, “factorial”
  - fact(1)=1
  - fact(n)=n * fact(n-1) for n>1

- For example:
  - fact(2) = 2 x fact(1) = 2 * 1 = 2
  - fact(3) = 3 x fact(2) = 3 * 2 = 6
  - fact(4) = 4 x fact(3) = 4 * 6 = 24
  - …



Factorials in C



```
1. int fact(int x);
2. int main() {
3.     int a = fact(4);
4.     return 0;
5. }
6. int fact(int x) {
7.     if (x==1) return 1;
8.     return x*fact(x-1);
9. }
```

Inv	Fn	args	vars	Ret
OS	main		a	

Factorials in C

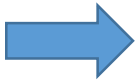
```
1. int fact(int x);
2. int main() {
3.     int a = fact(4);
4.     return 0;
5. }
6. int fact(int x) {
7.     if (x==1) return 1;
8.     return x*fact(x-1);
9. }
```

Inv	Fn	args	vars	Ret
3.12	fact	x=4		

Inv	Fn	args	vars	Ret
OS	main		a	

Factorials in C

```
1. int fact(int x);
2. int main() {
3.     int a = fact(4);
4.     return 0;
5. }
6. int fact(int x) {
7.     if (x==1) return 1;
8.     return x*fact(x-1);
9. }
```



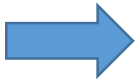
Inv	Fn	args	vars	Ret
8.13	fact	x=3		

Inv	Fn	args	vars	Ret
3.12	fact	x=4		

Inv	Fn	args	vars	Ret
OS	main		a	

Factorials in C

```
1. int fact(int x);
2. int main() {
3.     int a = fact(4);
4.     return 0;
5. }
6. int fact(int x) {
7.     if (x==1) return 1;
8.     return x*fact(x-1);
9. }
```



Inv	Fn	args	vars	Ret
8.13	fact	x=2		

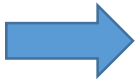
Inv	Fn	args	vars	Ret
8.13	fact	x=3		

Inv	Fn	args	vars	Ret
3.12	fact	x=4		

Inv	Fn	args	vars	Ret
OS	main		a	

Factorials in C

```
1. int fact(int x);
2. int main() {
3.     int a = fact(4);
4.     return 0;
5. }
6. int fact(int x) {
7.     if (x==1) return 1;
8.     return x*fact(x-1);
9. }
```



Inv	Fn	args	vars	Ret
8.13	fact	x=1		

Inv	Fn	args	vars	Ret
8.13	fact	x=2		

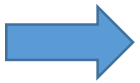
Inv	Fn	args	vars	Ret
8.13	fact	x=3		

Inv	Fn	args	vars	Ret
3.12	fact	x=4		

Inv	Fn	args	vars	Ret
OS	main		a	

Factorials in C

```
1.  int fact(int x);
2.  int main() {
3.      int a = fact(4);
4.      return 0;
5.  }
6.  int fact(int x) {
7.      if (x==1) return 1;
8.      return x*fact(x-1);
9.  }
```



Inv	Fn	args	vars	Ret
8.13	fact	x=1		1

Inv	Fn	args	vars	Ret
8.13	fact	x=2		

Inv	Fn	args	vars	Ret
8.13	fact	x=3		

Inv	Fn	args	vars	Ret
3.12	fact	x=4		

Inv	Fn	args	vars	Ret
OS	main		a	

Factorials in C

```
1. int fact(int x);
2. int main() {
3.     int a = fact(4);
4.     return 0;
5. }
6. int fact(int x) {
7.     if (x==1) return 1;
8.     return x*fact(x-1);
9. }
```

1

Inv	Fn	args	vars	Ret
8.13	fact	x=2		2

Inv	Fn	args	vars	Ret
8.13	fact	x=3		

Inv	Fn	args	vars	Ret
3.12	fact	x=4		

Inv	Fn	args	vars	Ret
OS	main		a	

Factorials in C

```
1. int fact(int x);
2. int main() {
3.     int a = fact(4);
4.     return 0;
5. }
6. int fact(int x) {
7.     if (x==1) return 1;
8.     return x*fact(x-1);
9. }
```

2

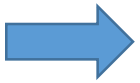
Inv	Fn	args	vars	Ret
8.13	fact	x=3		6

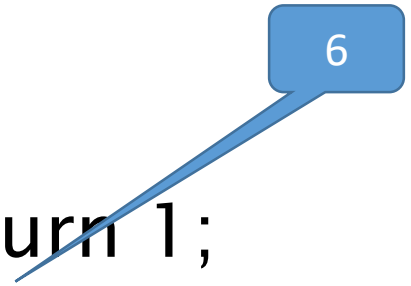
Inv	Fn	args	vars	Ret
3.12	fact	x=4		

Inv	Fn	args	vars	Ret
OS	main		a	

Factorials in C

```
1. int fact(int x);
2. int main() {
3.     int a = fact(4);
4.     return 0;
5. }
6. int fact(int x) {
7.     if (x==1) return 1;
8.     return x*fact(x-1);
9. }
```



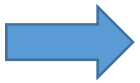


Inv	Fn	args	vars	Ret
3.8	fact	x=4		24

Inv	Fn	args	vars	Ret
OS	main		a	

Factorials in C

```
1. int fact(int x);
2. int main() {
3.     int a = fact(4);
4.     return 0;
5. }
6. int fact(int x) {
7.     if (x==1) return 1;
8.     return x*fact(x-1);
9. }
```



24

Inv	Fn	args	vars	Ret
OS	main		a=24	

Notes on Recursion

- Recursive functions always recurse to a simpler case
  - In “fact” – we always decrease argument by 1
- Recursive functions always have a “base” – a simplest case
  - Must be non-recursive to stop the recursion!
  - A condition that stops the recursion
  - In “fact” – base is the “1” case
- Recursive functions are very similar to inductive proofs in math
  - If the recursion works for 1, and if you assume the recursion works for *n-1* and show it works for *n*, then the recursion works for any integer
  - fact(1)=1, and If fact(3)=3 x 2 x 1, then fact(4) = 4 x fact(3) = 4 x 3 x 2 x 1

Recursion can be replaced by loops

```
1. int fact(int x);
2. int main() {
3.     int a = fact(4);
4.     return 0;
5. }
6. int fact(int x) {
7.     int f=1; int j; for(j=2;j<x;j++) f*=j;
8.     return f;
9. }
```

Recursion vs. Loops

- Recursion is often conceptually very simple, loops can be more complicated to understand
- Loops often seem to be significantly more efficient
  - Don’t have the overhead of function invocation, creating call stacks, etc.
- However, recursion is so common, the compiler optimizes recursion, often turning it into a loop

Bottom line: Use whatever makes the most sense to you, and let the compiler take care of efficiency

Resources

- Programming in C, Chapter 7
- YouTube: Meat-a-Morphis – Introduction to Functions https://www.youtube.com/watch?v=VUTXsPFx-qQ
- WikiPedia: Subroutine https://en.wikipedia.org/wiki/Subroutine
- WikiPedia: C Standard Library: https://en.wikipedia.org/wiki/C_standard_library
- Wikipedia: Recursion (computer science) https://en.wikipedia.org/wiki/Recursion_(computer_science)
- C-FAQ Library Functions: http://c-faq.com/lib/index.html
- Linux MAN pages online (library functions): http://man7.org/linux/man-pages/dir_all_alphabetic.html