Functions

CSC 240 - Data structures with C++ - Lyon College, FA24

Marcus Birkenkrahe

September 30, 2024

Overview

You should already know something about these derived data types: arrays, pointers, functions. We review by coding and analyzing code.

In this section, we focus on functions. Functions are the main way to modularize code. We review how to pass data to and retrieve data from functions. New topics include recursion and the quicksort algorithm.

- **Objectives:**
  - Prototyping functions
  - Arguments, parameters, array parameters and VLA parameters
  - Compound literals to initialize arrays
  - Cast operators, lvalues and rvalues
  - Exiting from a function in the case of failure
  - Recursion principle and Fibonacci application
  - Quicksort algorithm explained and coded
  - Code along and solve different exercises
  - Review questions, glossary, and summary

Code along:

1. Open your Linux VM
2. Open Emacs (terminal or GUI)
3. Create an Org-mode file with `C-x C-f 3_functions.org`

4. Add `#+property` top header for C with `:main yes :includes <stdio.h> :results output`
5. Cursor on the header: run it with `C-c C-c`
6. Create a C code chunk with `<s <TAB> C`
7. Write and run a "hello world" test program: `puts("hello world");`
8. Keep running chunks with `C-c C-=c` to check for syntax errors.
9. If any of these steps don't work for you, let me know!
10. To warm up, define an array `x` of 6 elements 1,3,2.5,3,4,3, print them in reverse order and print the number of times 3 occurs in `x`.

```
// declare and define floating-point array of 6 elements
float x[6] = {1,3,2.5,3,4,3};
// define counter and loop variable
int count=0, i;
// loop over array in reverse order
for (i = 5; i>=0; i--) {
  printf("%g ",x[i]);
  // check if array element is equal to 3
  if (x[i]==3) count++;
 }
// print result
printf("\n%d", count);
```

```
3 4 3 2.5 3 1
3
```

11. Would you know how to add functions `reverse` that reverse the array and `count` that do the counting? That'll be this week's homework!

Function prototypes

- A function prototype is a declaration of the function with its return type and its typed parameters ahead of the `main` function.
- Once the function has a prototype, it can be defined anywhere in the program, even after the `main` program.

- Example: An adding function `add` - it takes two integers and returns their sum.

  1. Without prototype (note the change in the code block header):

```
int add(int a, int b)
{
  return a + b;
}

int main(void)
{
  int x = 1, y = 1;
  printf("%d + %d = %d\n", x, y, add(x,y));
  return 0;
}
```

```
1 + 1 = 2
```

  2. With prototype:

```
int add(int, int); // prototype

int main(void)
{
  int x = 1, y = 1;
  printf("%d + %d = %d\n", x, y, add(x,y));
  return 0;
}

int add(int a, int b)
{
  return a + b;
}
```

```
1 + 1 = 2
```

```
1 + 1 = 2
```

Hello world as a (prototyped) function

- To reacquaint yourself with functions, turn the `hello world` program below into a program that prints `hello world` by calling a function `hello_world`. When successful, find different ways to break the code.
- The hello world program - a timeless, beautiful classic:

```
puts("Hello, world!");
printf("Hello, world!");
```

```
Hello, world!
Hello, world!
```

- Print "hello world" with a prototyped function `hello_world`:

```
void hello_world(void); // prototype

int main(void)
{
  hello_world();  // no arguments, no return type
  return 0;
}

void hello_world(void)
{
  puts("Hello, world!");
}
```

```
Hello, world!
```

- Break the function (non-trivially = no syntax errors):
  1. Define the function with ; at the end
  2. Try to `return` something from the `void` function
  3. Mismatch of defined function type with prototype
  4. Calling function with argument (instead of `void`)

```
void hello_world(void); // prototype

int main(void)
{
  hello_world();  // no arguments, no return type
  return 0;
}

void hello_world(void)
{
  puts("Hello, world!");
  return 1;
}


Hello, world!
```

- Solution:

  1. No arguments need to be passed to the function (it's `void` of arguments) so it can be called as `hello_world(void)`.
  2. The function does not `return` any values so it can be defined as `void hello_world(void)`.
  3. The function does not change any of its `void` arguments so it could be labelled as `const` but since there are no arguments, this attribute is meaningless.
  4. The compiler uses a *prototype declaration* to provide the compiler with a glimpse at a function whose full definition will come later. The prototype declaration resembles the function definition with `return` type, name, and parameter list.

     ```
     #include <stdio.h> // I/O library

     void hello_world(void); // prototype declaration

     int main(void) // main function
     {
       hello_world(); // function call
       return 0;
     }
     ```

```
void hello_world(void) // function definition
{
  // print string
  printf("Hello, world!");
}
```

```
Hello, world!
```

5. Break the function:
   (a) Add `return 0;` to its definition and run the program.
   (b) Change the prototype or the definition (but not both) `return` type to `int` and run the program.
   (c) Change the prototype or the definition argument (but not both) from `void` to `int i`.
   (d) Call the function with the argument `"hi"`.

Function documentation

- Remember the required solution components. Functions should be accompanied by a short summary at the top:
  1. Name and purpose of the function
  2. Return type and parameters
  3. Anything noteworthy (edge cases, version dependencies etc.)
- Example: Hello world program

```
#include <stdio.h>

// Function: hello_world
// Purpose: Prints "Hello, world!" to the console.
// Returns: void (no return value)
// Parameters: void (no parameters)
void hello_world(void); // prototype

int main(void)
{
```

```
  hello_world(); // function call
  return 0;
}

void hello_world(void) // function definition
{
  printf("Hello, world!");
}
```

```
Hello, world!
```

Arguments vs. Parameters

- *Parameters* appear in function *definitions*. They are dummy names that represent values to be supplied when the function is called.
- *Arguments* are expressions that appear in function *calls*. In C, they are **passed by value**: when a function is called, each argument is evaluated and its value assigned to the corresponding parameter.
- Since the parameter contains a **copy** of the argument's value, changes made to the parameter during execution of the function don't affect the argument.
- Because arguments are passed by value, parameters can be modified without affecting the corresponding argument, and we need fewer variables.

Example: `power` function

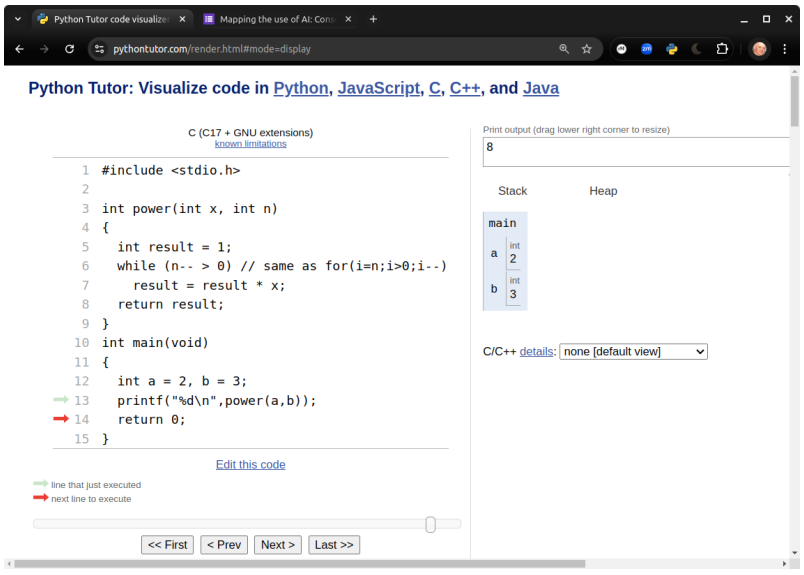
- Example: a `power` function to raise `x` to the power of `n`

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

- Note: you can (and should) type and test the function syntax first.
- Since `n` is a copy of the original exponent, we can modify it inside the function without affecting it outside, and save the variable `i`:

```
int power(int x, int n)
{
  int result = 1;
  while (n-- > 0) // same as ‘for(i=n;i>0;i--)‘: ‘n‘ is loop veriable
    result = result * x;
  return result;
}
```

- To analyse what's going on inside of `power2`, you can put it in a `main` program, tangle it as `power2.c`, and visualize it in `pythontutor.com`, or compile as `gcc -g` and run it with `gdb`.




```
int power(int x, int n)
{
  int result = 1;
  while (n-- > 0) // same as for(i=n;i>0;i--)
    result = result * x;
  return result;
```

```
}
int main(void)
{
  int a = 3, b = 2;
  printf("%d\n",power(a,b));
  return 0;
}


9
```

- We expect this upon entering the function, for x = 2, n = 3:

result	n
1	3
2	2
4	1
8	0

- Test the two power functions as `power` and `power2` with one `main` program! Don't forget:
  1. Two function prototypes
  2. One function description
  3. One `main` function with calls to both functions
  4. Two function definitions
- Test result:

```
: 2**3 = 8
: 2**3 = 8
```

- Solution template:

```
// Function: power, power2
// Purpose: Compute power of integer
// Returns: n-th power of integer argument
// Parameters: integer variable, integer power
```

```
// Main program
int main(void)
{

  return 0;
}

// Function definition

// Function definition
```

- Solution: Declare and test two power functions

```
// Function: power, power2
// Purpose: Compute power of integer
// Returns: n-th power of integer argument
// Parameters: integer variable, integer power
int power(int,int);
int power2(int,int);

int main(void)
{
  int x = 2, n = 3;
  printf("%d**%d = %d\n", x, n, power(x,n));
  printf("%d**%d = %d\n", x, n, power2(x,n));
  return 0;
}

int power(int x, int n)
{
  int i, result = 1;
  for (i=1;i<=n;i++)
    result*=x;
  return result;
}

int power2(int x, int n)
{
  int result = 1;
```

```
  while (n-- > 0)
    result*=x;
  return result;
}
```

```
2**3 = 8
2**3 = 8
```

Example: `decompose` function

- Passing arguments by value makes it difficult to write functions that produce two different results because only one number can be returned.
- Example: The function `decompose` splits its argument in an integer and a fractional part.

```
// Function: decompose
// Purpose: Splits argument x in integer and fractional part
// Returns: nothing
// Parameters: float x, integer part, float fractional part
// Uses: casting function
void decompose (float, int, float);

int main(void) {
  int i = 0;
  float d = 0.;
  float x = 3.14159;
  decompose(x, i, d);
  printf("x: %g, i: %d, d: %g\n", x, i, d);
  return 0;
}

void decompose (float x, int int_part, float frac_part)
{
  int_part = (int) x; // cast x as integer, drops fractional part
  frac_part = x - int_part; // fractional part
}
```

```
x: 3.14159, i: 0, d: 0
```

- When the function is called, `3.14159` is copied into `x`, the value of `i` is copied into `int_part` and the value of `d` is copied into `frac_part`. But when the function returns, `i` and `d` are unchanged.
- Do you remember how to fix this?

```
void decompose (float, int *, float *); // pass pointers

int main(void) {
  int i = 0;
  float d = 0.;
  float x = 3.14159;
  decompose(x, &i, &d); // pass value, and references
  printf("x: %g, i: %d, d: %g\n", x, i, d);
  return 0;
}

void decompose (float x, int *int_part, float *frac_part)
{
  (*int_part) = (int) x; // cast x as int integer, drops fractional part
  (*frac_part) = x - *int_part; // fractional part
}
```

```
x: 3.14159, i: 3, d: 0.14159
```

The variables `i` and `d` are now passed as references, and the parameters catch them as typed *pointer* variables that point at the memory locations of `i` and `d` - when the value the pointers point to, `*int_part` and `*frac_part` are changed the original variables are changed, too. We'll review pointers next!

- *Note:* In the code block, I wrote `(*int)` because Org-mode gets confused with `*` at the start of a line (expects bullet points).

Practice: add two integers - pass by reference

- Here is another version of the function to add two integer values:

```
// Function: add
// Purpose: Adds two integer values
// Returns: sum a + b
// Parameters: int a, int b (passed by value)
int add(int, int);

// main program
int main(void)
{
  int x = 1, y = 1;
  printf("%d + %d = %d\n", x, y, add(x,y));
  return 0;
}

// function definition
int add(int a, int b)
{
  return a + b;
}
```

```
1 + 1 = 2
```

- The function call passes arguments `x` and `y` by value. Create a new `void` function `add2` that passes the integers by reference using pointers, print the `sum` inside the function, and set the original arguments to 0.
- Test output:

```
: x = 1, y = 1
: 1 + 1 = 2
: x = 0, y = 0
```

- Solution template:

```
// Function: add
// Purpose: Adds two integer values
// Returns: Nothing
// Parameters: int *a, int *b (passed by reference)
```

```
// Main program

// Function definition
```

- Solution:

```
// Function: add
// Purpose: Adds two integer values
// Returns: Nothing
// Parameters: int *a, int *b (passed by reference)
void add2(int*,int*);

// main program
int main(void)
{
  // define variables
  int x = 1, y = 1;
  // print original values
  printf("x = %d, y = %d\n", x, y);
  // call function (pass by reference)
  add2(&x,&y);
  // print modified values
  printf("x = %d, y = %d\n", x, y);
  return 0;
}

// function definition
void add2(int *a, int *b)
{
  int sum = *a + *b;
  printf("%d + %d = %d\n", *a, *b, sum);
  // modify a an b
  (*a) = 0;
  (*b) = 0;
}


x = 1, y = 1
```

```
1 + 1 = 2
x = 0, y = 0
```

Bonus assignment: Say hello by reference

- **Programming assignment (bonus):** modify a simple 'hello world' program from pass-by-value to pass-by-reference and modify the original character.

```
// Function: say_hello
// Purpose: Prints "Hello, <char>!" message
// Returns: nothing
// Parameters: char c (passed by value)
void say_hello(char);

int main(void) {
  char initial = 'W';

  say_hello(initial); // Passing by value
  printf("After function call: initial = %c\n", initial); // 'W' remains unchanged

  return 0;
}

void say_hello(char c) {
  printf("Hello, %c!\n", c);
}


Hello, W!
After function call: initial = W
```

Example: 1d Array Arguments

This is the concept that you need to complete the next mandatory programming assignment.

- When given an array `a` of `int` values, `sum_array` returns the sum of the elements in `a`. The length of `a` is supplied as a second argument:

```
// Function: sum_array
// Purpose: Returns the sum of an integer array
// Parameters: int a[], int n (array and its length)
int sum_array(int[],int);

// main function
int main() {

  int a[5]={1,1,1,1,1};

  printf("Sum of array of %d elements: %d\n",
     5, sum_array(a,5)); // no [] brackets when passing the array
  return 0;
}

// function definition
int sum_array(int a[], int n) { // brackets after the parameter
  int i, sum = 0;
  for (i = 0; i < n; i++)
    sum += a[i];
  return sum;
}


Sum of array of 5 elements: 5
```

- A function has no way to check that we've passed it the correct array length (so you must be careful - easy to cause memory leaks).

Practice: Passing array arguments and printing them

- Objective: Write a function `print_array` that takes an integer array and its length as arguments and prints the elements of the array.
  1. Implement the `print_array` function that takes an array and its length.
  2. Use a loop to print each element of the array inside the function.
- Sample output: For the array `{1, 2, 3, 4, 5}`, the output will be:

```
: Element 0: 1
: Element 1: 2
: Element 2: 3
: Element 3: 4
: Element 4: 5
```

- Solution:

```
// Function: print_array
// Purpose: Print one-dimensional array
// Parameters: int a[],  int n (array and its length)
void print_array(int [], int);

int main(void)
{
  int a[5]={1,2,3,4,5};
  print_array(a,5);
  return 0;
}

void print_array(int a[], int n)
{
  for (int i = 0; i < n; i++)
    printf("Element %d: %d\n", i, a[i]);
}
```

```
Element 0: 1
Element 1: 2
Element 2: 3
Element 3: 4
Element 4: 5
```

Example: 2d Array Arguments

- For multi-dimensional arrays, only the length of the first dimension may be omitted when declaring the parameter:

```
#define LEN 2
// Function: sum2
```

```
// Purpose: Sum elements of 2-d array
// Parameters: n x LEN dimensional array, row length int n
int sum2(int [][LEN], int n);

int main()
{
  int a[][LEN]={[0][0]=1,[1][1]=1}; // 2 x 2 identify matrx
  for (int i=0;i<2;i++) { // loop over rows
    if(i!=0) puts(" ");
    for (int j=0;j<2;j++) { // loop over columns
printf("%d ", a[i][j]);
    }
  }
  printf("\nSum of 2d array: %d\n", sum2(a,2));
  return 0;
}
int sum2(int a[][LEN], int n)
{
  int i, j, sum = 0;
  for (i=0; i<n;i++) { // loop over rows
    for (j=0;j<LEN;j++) { // loop over columns
sum+=a[i][j];
    }
  }
  return sum;
}


1 0
0 1
Sum of 2d array: 2
```

- How can the 2d array be printed in matrix format?

```
int main()
{
  int a[2][2]={1,0,0,1};
  for (int i=0;i<2;i++) {
    if (i!=0) puts(" ");
    for (int j=0;j<2;j++) {
```

```
printf("%d ", a[i][j]);
    }
  }
  return 0;
}
```

```
1 0
0 1
```

Bonus assignment: Print two-dimensional array with a function

- **Homework assignment (bonus):** Modify the function `print_array` to print a 2-dimensional array.

Variable-Length Array Parameters

- Variable-length arrays allow to state the length of an array in a function argument. They are most useful for multidimensional arrays.
- In this function definition, there is no direct link between `n` and the length of the array `a`. The array could in fact be larger or smaller than `n`, and then the function would not work.

  ```
  int sum_array(int a[], int n) {
    // ...
  }
  ```

- Using a variable-length array parameter, we can explicitly state that the length of `a` is `n`:

  ```
  int sum_array(int n, int a[]) {
    // ...
  }
  ```

- But now the order of parameters is important: `int n, int a[n]` is OK, but `int a[n], int n` is illegal.

- **Practice:** Sum an array of length 10 without VLA (`sum_array`), and with VLA (`sum_array_vla`) then call the function with the values 5,10,11 for n.

```
int sum_array_vla(int n, int a[n]);
int sum_array(int a[], int n);

int main(void)
{
  int a[10]={[0 ... 9]=1};
  printf("vla: %d\n", sum_array_vla(11,a));
  printf("regular: %d ", sum_array(a,11));
  return 0;
}

int sum_array_vla(int n, int a[n])
{
  int i, sum=0;
  for (i=0;i<n;i++)
    sum+=a[i];
  return sum;
}

int sum_array(int a[], int n)
{
  int i, sum=0;
  for (i=0;i<n;i++)
    sum+=a[i];
  return sum;
}


vla: 1894034954
regular: 1894034954
```

- What did you find out?

  The VLA generates a warning for n>10.

- Earlier, we summed the elements in a 2D array. The function was limited to arrays with a fixed number of columns. With a VLA parameter, we can generalize the function to any number of columns:

```
int sum_two_dimensional_array(int n, int m, int a[n][m]);

int main(void)
{
  int n = 4, m = 4, i, j;
  int a[n][m];
  for(i=0;i<n;i++) {
    for(j=0;j<m;j++) {
a[i][j]=i+j;
printf("%d ",a[i][j]);
    }
    printf("\n");
  }
  printf("Sum: %d\n", sum_two_dimensional_array(n, m, a));

  return 0;
}

int sum_two_dimensional_array(int n, int m, int a[n][m])
{
  int i,j,sum=0;
  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
sum+=a[i][j];
  return sum;
}


0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
Sum: 48
```

- These are all alternative ways to declare a VLA: the first one is to be preferred because it suggests the VLA, and also because the parameter list `int a[n], int n` is illegal (`n` is not known yet).

```
int func1(int n, int a[n]);
int func2(int, int []);
```

```
int func1(int n, int a[*]);
int func1(int, int [*]);
```

- VLA parameters can also be expressions to be evaluated, for example in this function where two arrays `a` and `b` are concatenated by copying them successively into an array `c`:

```
int concat(int m, int n, int a[m], int b[n], int c[m+n]);
```

Compound Literals

- The term 'literal' always refers to unchangeable language elements, and the term 'compound' means that keywords are combined, like in `i++`, the compound operator representing `i = i + 1`.

- When summing an array's elements in , the array needs to be declared and initialized. Using a *compound literal*, this can be avoided:

```
// prototype declaration
int sum_array2(int n, int a[n]);

// main function
int main()
{
  printf("total = %d\n",
     sum_array2(5, (int []){3,0,3,4,1}));
  return 0;
}

// function definition
int sum_array2(int n, int a[n])
{
  int i, sum = 0;
  for (i=0;i<n;i++)
    sum += a[i];
  return sum;
}



total = 11
```

- A compound literal resembles a cast `(int[])` applied to an initializer `{}`. It is also an *lvalue*, so the value of its elements can be changed. It can be made read-only by adding `const` before the type.

- A *cast* is an operator that converts one type into another:

```
int i = 1;
float x = 3.14;
char c = 'a';
char *ic = "22";

printf("%f\n", (float)(i)); // cast int to float (widening)
printf("%d\n", (int)(x)); // cast float to int (narrowing/truncates)
printf("%c\n",c); // no cast
printf("%d\n",(int)(c)); // cast char to int
printf("%i\n", atoi(ic)); // cast string (char pointer) to int


1.000000
3
a
97
22
```

- `atoi` is a function from the C Standard Library (`stdlib.h`) that converts a numeric string into an integer value. The syntax looks like this:

  ```
  int atoi(const char *str)
  ```

  The argument is a constant pointer to a string. We'll learn more about strings later.

- An *lvalue* or *locator value* is an expression that represents a memory location, and that can appear on the left hand side of an assignment operator. As an object it persists beyond a single expression and can have a value assigned to it.

  By contrast, an *rvalue* or *right value* represents a data value stored in memory but that is not an assignable object itself.

  For example:

```
int x = 10;   // 'x' is an lvalue, '10' is an rvalue
int *p = &x;  // 'p' is an lvalue, '&x' is an rvalue

(*p) = 20;      // '*p' is an lvalue, '20' is an rvalue
x = x + 5;    // 'x' is an lvalue, 'x + 5' is an rvalue
```

- Are functions and operators the same thing, for example the functions `stdlib::atoi`, `stdio::printf` vs. the operators `sizeof` and `(int)`?

  An operator is compiled to a sequence of instructions by the compiler. But if the code calls a function, it has to jump to a separate piece of code.

The `return` statement

- Functions are the most important building blocks of C programs. Using functions, we can divide a program into smaller parts that are easier to maintain and to understand.
- `return` statements may appear in functions whose return type is `void` as long as no expression is given:

```
void print(int i)
{
  if (i<0)
    return;
  printf("%d",i);
}

int main()
{
  print(1); // prints 1
  print(-1); // prints nothing, return without calling printf
  return 0;
}


1
```

The `exit` Function

- The argument passed to `stdlib::exit` indicates the status at termination: `exit(0)` is normal termination `exit(1)` is failure, but you can use `exit(EXIT_SUCCESS)` or `exit(EXIT_FAILURE)`, two macros defined in `stdlib.h`.
- Exercise: define success as `SUCCESS` and failure as `FAILURE` and demonstrate their use with `exit` in a short program by comparing two numbers which you input via standard input.

  Input file

```
echo 300 200 > input
cat input


300 200
```

```
#define FAILURE EXIT_FAILURE
#define SUCCESS EXIT_SUCCESS

int main() {
  // scan an integer
  int i,j;
  scanf("%d%d", &i, &j);
  // print 0 or 1 depending on input values
  printf("%d", i > j ? SUCCESS : FAILURE);
  return 0;
}
```

- The expression in the `printf` argument resolves to:

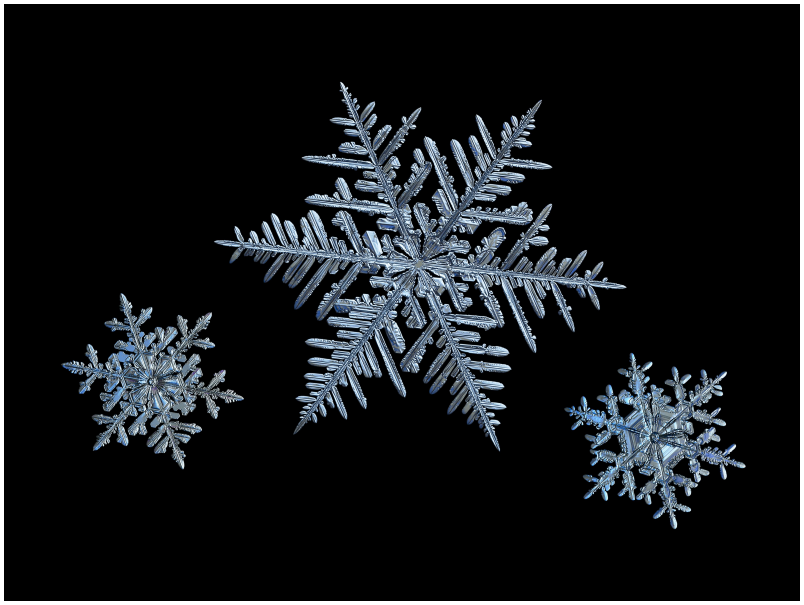
```
if (i > j)
{
  EXIT_SUCCESS; // return 0 for success and exit
} else {
  EXIT_FAILURE  // return 1 for failure and exit
}
```

Recursion

The following is taken from Loudon, Algorithms in C (1999).

- Recursion is a powerful principle that allows something to be defined in terms of smaller instances of itself.
- Recursion is a recurring principle in nature: think of the leaf of a fern - each sprig is a copy of the overall leaf.

- Another example is a snowflake (we'll get back to snowflakes when we introduce hash tables).



- In math and computing, a function is *recursive* if it calls itself. Each successive call works on a more refined set of inputs, bringin us closer to the solution of a problem.
- Algorithmic applications of recursion include tree traversals, graph searches, and sorting.
- Example: computing the factorial n! = n × (n-1) × ... × 1

```
int factorial(int n)
{
  if (n<=1)
    return 1; // abort
  else
    return n * factorial(n-1);
  // return n <= 1 ? 1 : n * factorial(n-1);
}
int main()
{
  int n = 3;
  printf("The factorial of n = %d is %d\n", n, factorial(n));
  return 0;
}
```

```
The factorial of n = 3 is 6
```

- What happens for i = fact(3):

```
call fact(3): 3 > 1
     call fact(2): 2 > 1
          call fact(1): 1 = 1 return 1
     return 2 * 1
return 3 * 2 * 1 = 6
```

- The first part of the recursive process is the "winding phase", ended by the "terminating condition". The second part is the "unwinding phase".

- **Exercise:** compute x^n using the formula $x^n = x \times x^{n-1}$. For example for x = 2, n = 3: $2^3 = 2 \times 2^{3-1} = 2 \times 2^2 = 2 \times (2 \times 2^{2-1}) = 2 \times 2 \times 2 = 8$

```
int power(int x, int n); // function declaration

int main()  // main program
{
  int x = 5; // number to be raised
  int n = 2; // power factor
  printf("%d^%d = %d\n", x, n, power(x,n));
  return 0;
}

int power(int x, int n)
{
  if (n == 0)
    {
return 1;
    }
  else
    {
x =  x * power(x,n-1);
    }
}
```

```
5^2 = 25
```

- What happens for x = 5, n = 3?

```
call power(5,3) : 3 != 0
     call power(5,2) : 2 != 0
          call power(5,1) : 1 != 0
               call power(5,0) : 0 == 0 return 1
          return 5 * 1 = 5
     return 5 * 5 = 25
return 5 * 25 = 125
```

- Here is the short version of both functions:

```
int factorial(int n)
{
  return n <= 1 ? 1 : n * factorial(n-1);
}
int main()
{
  int n = 5;
  printf("The factorial of n = %d is %d\n", n, factorial(n));
  return 0;
}
```

```
The factorial of n = 5 is 120
```

```
int power(int x, int n); // function declaration

int main()  // main program
{
  int x = 5; // number to be raised
  int n = 3; // power factor
  printf("%d^%d = %d\n", x, n, power(x,n));
  return 0;
}

int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x,n-1);
}
```

```
5^3 = 125
```

The Quicksort Algorithms explained

- Recursion arises naturally in divide-and-conquer algorithms, in which a large problem is divided into smaller pieces that are then tackled by the same algorithm.
- A classic example is Quicksort to sort the elements of an array indexed from 1..n
    1. Choose a partitioning element, then arrange array so that elements 1...i-1 <= e <= i+1...n
    2. Sort elements 1...i-1 by using Quicksort recursively
    3. Sort elements i+1...n by using Quicksort recursively
- The first step of partitioning is critical to the method working. This algorithm is not particularly effective but easy to understand:
    1. To markers *low* and *high* keep track of array positions.
    2. Initially, *low* points to the head, and *high* to the tail.
    3. Copy the first element e into a temporary location.
    4. Move *high* across array from right to left until it points to an element that's smaller than e.
    5. Copy that element into the hole left by e (that *low* points to).
    6. Move *low* from left to right looking for an element that's larger than e.
    7. Copy that element into the hole that *high* points to.
    8. Continue process alternating *high* and *low* until they meet.
    9. Copy e into the hole that both *low* and *high* point at.
- Illustration with an array of seven integers:
    1. *low* points to the first, *high* to the last element.

12	3	6	18	7	15	10
low						*high*

    2. Copy $e = 12$ elsewhere

	3	6	18	7	15	10
low						*high*

3. Compare *high* element to e = 12. Since 10 < 12 it's on the wrong side of the array and is moved to the hole.

10	3	6	18	7	15	
	low					*high*

4. *low* points to 3 < 12, then 6 < 12, then 18 > 12, which is on the wrong side of the array: it is moved to the hole and *high* is shifted to the left now.

10	3		6	18	7	15	
		low					*high*

10	3	6		18	7	15	
			low				*high*

10	3	6		7		15	18
			low			*high*	

5. *high* points to 15 > 12 (can stay), then to 7 < 12 which needs to be moved to the hole, then *low* and *high* point to the same hole, and $e = 12$ is moved there:

10	3	6				7	15	18
				low	*high*			

10	3	6	7				15	18
				low	*high*			

10	3	6	7	**12**	15	18

6. We've completed our objective for the first sorting process: all elements to the left of e are less or equal than 12, all elements on the right are greater or equal than 12.
7. Now we apply Quicksort recursively to sort the first (10,3,6,7) and the last partition (15,18).

Quicksort Algorithm coded

Problem

Use Quicksort to sort an array of integers.

Input

Array {9,16,47,82,4,66,12,3,25,51}

Output

Sorted array {3,4,9,12,16,25,47,51,66,82}.

Code

The split is performed with a function `split`, the sorting with a function `quicksort`. The input comes from a file `input` (see below):

```
#define N 10

// function declarations
void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);

// main function
int main(void)
{
  int a[N], i; // declare array, loop variable
  printf("Enter %d numbers to be sorted: \n", N);
  for (i=0;i<N;i++) {    // get input array
    scanf("%d",&a[i]);
    printf("%d ", a[i]);
  }
  quicksort(a, 0, N-1); // call quicksort
  puts("\nIn sorted order:");  // print result
  for (i=0;i<N;i++)
    printf("%d ", a[i]);
  puts("");
  return 0;
}

// quicksort with recursion
void quicksort(int a[], int low, int high)
{
  int middle;
  if (low >= high) return; // do nothing
```

```
  middle = split(a, low, high); // find partitioning element
  quicksort(a, low, middle-1); // move low to right
  quicksort(a, middle+1, high); // move high to left
}
// split to find the partitioning element
int split(int a[], int low, int high)
{
  int part_element = a[low]; // partition starting with lowest element

  for (;;) {  // forever
    while (low < high && part_element <= a[high]) // move high left
      high--;
    if (low >= high) break;
    a[low++] = a[high];

    while (low < high && a[low] <= part_element) // move low right
      low++;
    if (low >= high) break;
    a[high--] = a[low];
  } // end of forever loop

  a[high] = part_element;
  return high;
}
```

```
Enter 10 numbers to be sorted:
300 200 -1075053569 0 1007352057 32764 100 0 4096 0
In sorted order:
-1075053569 0 0 0 100 200 300 4096 32764 1007352057
```

```
echo 9 16 47 82 4 66 12 3 25 51 > input
cat input
```

```
9 16 47 82 4 66 12 3 25 51
```

Improvements

- Instead of taking the first element, it's more efficient to take the median of the first, the middle and the last element.

- It's possible to avoid the `low<high` tests in the `while` loops.
- For smaller arrays (< 25 elements), use a different method.
- It's actually more efficient if the recursion is removed.
- More details in algorithm books like Sedgewick's Algorithms in C (2001), and in CSC 245 Algorithms next term.

Notes

- *Parameter* and *argument* can be used interchangeably. I prefer "argument" for the function call, and "parameter" for the function declaration and definition.
- C/C++ does not allow function definitions to be nested. Python and R, for example, allow nested function definitions:

```
def outer_function():
    print("This is the outer function.")

    def inner_function():
  print("This is the inner function.")

    # Call the inner function
    inner_function()

# Call the outer function
outer_function()
```

```
This is the outer function.
This is the inner function.
```

```
outer_function <- function() {
  print("This is the outer function.")

  inner_function <- function() {
    print("This is the inner function.")
  }
  ## Call the inner function
```

```
  inner_function()
}
## Call the outer function
outer_function()
```

- A function name that is not followed by parentheses is interpreted as a *pointer* by the compiler.
- The names of parameters in the function prototype do not have to match the names given later in the function's definition.
- We bother with function prototypes because
  1. not all functions are being called from `main` so we'd have to watch their order carefully if they weren't declared at top.
  2. If two undeclared functions call each other, one of them will not have been defined yet.
  3. Once programs get larger, it is no longer feasible to put all functions in one file, and we need prototypes to tell the compiler about functions in other files.
- Functions with the same return type can be combined, for example

```
void print_1(void), print_2(int n);
double x, y, average(double a, double b);
```

- If you specify a length for a 1 dim array parameter, the compiler ignores it. It cannot automatically check that arguments have that length (no added security), and it may be misleading:

```
double inner_product(double v[3], double w[3]);
```

- Why can the first dimension in an array parameter be left unspecified but not the other dimensions?
  - If `a` is a 1-dim array passed to a function, it decays to a pointer to the first element. When we write `a[i]=0;`, the address of `a[i]` is computed by multiplying `i` by the size of an array element and adding the result to the pointed to address. This does not depend on the length of `a`.

- If `a` is 2-dim and we write `a[i][j]=0;` the compiler (1) multiplies `i` by the size of a single row of `a`, (2) add the result to address of `a`, (3) multiplies `j` by the size of an array element, and (4) adds the result to the address computed in (2). Only the size of a row in the array (number of columns) is needed at the start and must be declared, not the rows (first index).

- Is it legal for a function `f1` to call a function `f2` which then calls `f1`?

  Yes, this is just an indirect form of recursion but one must make sure that either of these functions eventually terminates!

Review Questions

1. What is a "function prototype"? Give an example.

   A function prototype is a declaration of a function ahead of `main`. The more specific it is, the better the compiler can prepare itself for the function's definition and call.

```
// declare function without arguments and with no return type that
void hello_world(void);
```

2. Why do we bother with prototyping functions?

   (a) We don't have to watch the order of function definitions if functions are called from outside of `main`
   (b) If two undeclared functions call each other, one of them will not have declared yet.
   (c) For large programs, when functions reside in header files, the compiler needs prototype information to properly link the code.

3. What is a function parameter?

   A function parameter is a dummy value that represents a value to be supplied when the function is called.

4. What does "In C, function arguments are passed by value" mean?

C function arguments are passed by value in that each argument is evaluated and passed to the corresponding parameter.

```
main: f(5+2); // evaluate 5+2 and pass 7 to f
f: f(int i); // assign 7 to i
```

5. Are changes to function parameter values reflected by changes in the call argument?

   No. The parameter value is a copy of the original argument value. Changed values need to be returned from the function to alter memory outside the function - unless you make the variable `static`, or pass reference arguments to pointer parameters.

6. How is an array `a[n]` normally passed to a function `f`?

   The function call passes `a` and the length of `a`, `n`. The function parameters are `int a[], int n`.

7. What is a source of "memory leaks" when passing arrays to functions? What does the compiler have to say about this?

   The compiler cannot check that we've passed the correct array length to the function. If we write to memory outside of the defined array, we cause a *memory leak*. This may lead to a *segmentation fault* or *buffer overflow* error later.

8. Which of these lines declares a variable-length array, and what's the point of such an array?

```
int func(int n, int a[n]);
int func(int, int []);
int func(int a[n], int n);
int func(int a[], int n);
```

   Answer:

   The length of VLAs can be specified using a non-constant expression, and VLAs can also be parameters.

```
int func1(int n, int a[n]);
int func2(int, int []);
int func3(int a[n], int n); // not correct: n is not known yet
int func4(int a[], int n); // not a VLA
```

9. What's the meaning of 'compound literal', and what's an example?

   A compound literal is composed of an array term and an initializer list - as a way to save declaring and initializing an array, for example for `f(int n, int a[n])`, the call in `main` could look like this: `f(5, (int []) {1,2,3,4,5})` to initialize `a[5]`.

10. What is a *cast*? Give an example.

    If `int i=1;` is defined as `int`, it can be cast (widening) to a `float` with the `(float)` operator: `(float)(i)`.

11. What does the `exit` function do? Give an example of its use.

    The `exit` function terminates a program and returns a status code to the operating system. The status code indicates whether the program ended successfully or encountered an error.

```
#include <stdlib.h>

int main() {
    int result = 1; // Assume some operation that returns 1 on failure

    if (result != 0) {
        printf("Operation failed.\n");
        exit(EXIT_FAILURE); // Exit with failure status
    }

    printf("Operation succeeded.\n");
    exit(EXIT_SUCCESS); // Exit with success status
}
```

12. Describe the process of recursion. Give a short example.

Recursion is a process where a function calls itself directly or indirectly in order to solve a problem. Each call works on a smaller instance of the same problem, and the process typically includes a base case to terminate the recursive calls.

```
#include <stdio.h>

int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    int num = 5;
    printf("Factorial of %d is %d\n", num, factorial(num));
    return 0;
}
```

13. What is Quicksort and how does it work?

    Quicksort is a divide-and-conquer algorithm used to sort an array. It works by selecting a 'pivot' element and partitioning the array into two sub-arrays: one with elements less than the pivot and one with elements greater than the pivot. The process is recursively applied to the sub-arrays.

Practice Exercises

Function with errors

The following function, which computes the area of a triangle, contains two errors. Locate the errors and show how to fix them. (Hint: There are no errors in the formula.)

```
double triangle_area(double base, height)
double product;
{
```

```
  product = base * height;
  return product / 2;
}
```

Write a function `check(x,y,n)`

Write a function `check(x,y,n)` that returns `1` if both `x` and `y` fall between `0` and `n-1` inclusive, and `0` otherwise. Assume that `x, y`, and `n` are all of type `int`.

Input/Output:

x	y	n	return
0	1	0	0
0	1	1	0
0	1	-1	0
0	0	1	1
1	1	2	1
0	-1	0	1

Greatest Common Denominator

Write a function `gcd(m,n)` that calculates the greatest common divisor of the integers `m` and `n` using Euclid's algorithm.

Hint: The classic algorithm for computing the GCD, known as Euclid's algorithm, goes as follows: Let `m` and `n` be variables containing the two numbers. If `n` is 0, then stop: `m` contains the GCD. Otherwise, compute the remainder when `m` is divided by `n`. Copy `n` into `m` and copy the remainder into `n`. Then repeat the process starting with testing whether `n` is `0`[1].

You can use this program that does the job without a function:

```
printf("Enter two integer numbers: ");
int m, n, gcd;
scanf("%d%d",&m,&n);
printf("m = %d, n = %d",m,n);

if (n==0) {
```

[1] Euclid's algorithm is based on the fact that the GCD of two numbers does not change if the larger number is replaced by its difference with the smaller number. The instructions shortcuts these steps, instead replacing the larger of the two numbers by its remainder when divided by the smaller of the two. With this version, the algorithm never requires more steps than five times the number of digits (base 10) of the smaller integer

```
  gcd = m;
 } else if (m==0) {
  gcd = n;
 } else {
  while (m%n != 0) {
    gcd = m%n;
    m = n;
    n = gcd;
  }
 }
printf("\nGCD = %d\n", gcd);
```

Testing:

```
gcc gcd.c -o gcd
echo 12 20 | ./gcd
```

Programming assignments

1. Write functions that return the following values. (Assume that `a` and `n` are parameters where `a` is an array of `int` values and `n` is the length of the array.

   (a) The largest element in `a`.

   (b) The average of all elements in `a`.

   (c) The number of positive elements in `a`.

2. Write the following function:

   ```
   float compute_GPA(char grades [], int n);
   ```

   The `grades` array will contain letter grades (`A`, `B`, `C`, `D`, or `F=m either upper-case or lower-case); =n` is the length of the array. The function should return the average of the grades (assume that `A=4`, `B=3`, `C=2`, `D=1`, and `F=0`).

   Hint: the `stdlib::toupper` function turns lower case characters into upper case characters.

   ```
   #include <stdlib.h>
   printf("%c\n", toupper('a'));
   ```

3. The following function finds the median of three numbers. Rewrite the function so that it has just one `return` statement:

   ```
   double median(double x, double y, double z)
   {
     if (x <= y)
       if (y <= y) return y;
       else if (x <= z) return z;
       else return x;
     if (z <= y) return y;
     if (x <= z) return x;
     return z;
   }
   ```

4. Write a recursive version of the `gcd` function (see practice exercises) to find the greatest common denominator of two integers. Here's the

strategy to use for computing `gcd(m,n)`: If `n` is 0, `return =m`. Otherwise, call `gcd` recursively, passing `n` as the first argument and `m % n` as the second.

In the `example` block below, sketch what happens for m=12, n=28 when the function `gcd_r` is called

```
call gcd_r(12,28) ...
     call gcd_r(...)...
...
```

Here is the `gcd` function with a `main` function and a shell block to test it (tangle `gcd.c` first):

```
int gcd(int a, int b)
{
  while (b != 0) {
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}

int main(void)
{
  int m, n;
  scanf("%d%d",&m,&n);
  printf("GCD of %d and %d: %d\n",m,n,gcd(m,n));
  return 0;
}
```

```
gcc gcd.c -o gcd
echo 12 28 | ./gcd
```

Glossary

Term/Command	Explanation
`hello_world`	Function to print "hello world"
`void`	Keyword indicating no arguments or no return value
`return`	Statement to exit a function, optionally returning a value
`const`	Keyword indicating the value is constant
prototype declaration	Declaration providing a preview of a function
`int`	Keyword to define an integer type
`double`	Keyword to define a double-precision floating-point type
parameters	Dummy names representing values in function definitions
arguments	Expressions passed to functions in calls
`power`	Function to raise a number to a power
`decompose`	Function to split a double into integer and fractional parts
`sum_array`	Function to sum elements of an integer array
`puts`	Function to write a string to stdout
`#define`	Preprocessor directive to define a macro
`exit`	Function to terminate the program with a status code
`EXIT_SUCCESS`	Macro indicating successful program termination
`EXIT_FAILURE`	Macro indicating unsuccessful program termination
`atoi`	Function to convert a string to an integer
lvalue	Expression representing a memory location, assignable
rvalue	Expression representing a data value, not assignable
`* (dereference)`	Operator to access the value at a pointer address
`& (address-of)`	Operator to get the address of a variable
`printf`	Standard library function to print formatted output
`scanf`	Standard library function to read formatted input
`main`	Main function, entry point of a C program
`malloc`	Function to allocate memory dynamically
`free`	Function to deallocate dynamically allocated memory
`while`	Loop statement to execute code as long as a condition is true
`for`	Loop statement to execute code a fixed number of times
`quicksort`	Recursive function to sort an array using Quicksort algorithm
`split`	Function to find the partitioning element in Quicksort
recursion	Principle of a function calling itself
divide-and-conquer	Algorithm design paradigm dividing a problem into subproblems

Summary

1. Functions are prototyped to provide the compiler with information about the function's return type, name, and parameters before its definition.

2. Parameters in function definitions are placeholders for values to be supplied when the function is called.

3. Arguments in function calls are passed by value, meaning the function operates on copies of the values, not the original variables.

4. Breaking a function can be done by modifying its return type, changing its parameters, or altering its definition inconsistently with its prototype.

5. Recursion involves a function calling itself directly or indirectly, working on smaller instances of the same problem until a base case is reached.

6. Functions can be designed to operate on arrays by passing the array and its length as arguments.

7. Variable-length arrays (VLAs) allow the array size to be specified at runtime, providing flexibility for functions that handle arrays.

8. Compound literals enable array initialization directly within function calls, simplifying the code and avoiding separate declarations.

9. The return statement is used to exit a function and optionally return a value to the calling function.

10. The exit function terminates a program and returns a status code to the operating system, indicating success or failure.