String Recursion Exercises

2024 Winter APS 105: Computer Fundamentals

Jon Eyolfson

Lecture 27

1.0.0

A Recursive Function Calls Itself

We need two things:

1. a base case: a simple solution we know
2. a recursive step: reduces the problem to a smaller version of itself

Recursion with Strings

There are 3 major ways to think about recursively using strings:

1. A character followed by a smaller string
2. A smaller string preceding a character
3. Two characters enclosing a smaller string

Can We Recursively Check if a String is a Palindrome?

Reminder: a palindrome is a string that’s the same forwards as backwards

A Recursive Solution to Checking a Palindrome

```c
bool is_palindrome_helper(const char *s, int first, int last) {
    if (first >= last) {
        return true;
    }
    else if (s[first] != s[last]) {
        return false;
    }
    else {
        return is_palindrome_helper(s, first + 1, last - 1);
    }
}

bool is_palindrome(const char *s) {
    return is_palindrome_helper(s, 0, strlen(s) - 1);
}
```

The following is more C features that you shouldn’t use for this course
However, you may need to read them, or use them in the future

There is a Ternary Conditional Operator

It’s another expression with the syntax:

```
<conditional> ? <value_if_true> : <value_if_false>
```

Where you replace:

- `<conditional>` by a boolean expression
- `<value_if_true>` is the result of the expression if the conditional is true
- `<value_if_false>` is the result of the expression if the conditional is false

There is a Ternary Conditional Operator

It's another expression with the syntax:

```
<conditional> ? <value_if_true> : <value_if_false>
```

Where you replace:

- `<conditional>` by a boolean expression
- `<value_if_true>` is the result of the expression if the conditional is true
- `<value_if_false>` is the result of the expression if the conditional is false

Examples:

$\texttt{true ? 1 : 0} \rightarrow \texttt{1}$

$\texttt{false ? 1 : 0} \rightarrow \texttt{0}$

You should only use these for very simple expressions
  otherwise, the equivalent `if` and `else` is clearer

You Can Give Your Own Meaning to Numbers with `enum`

You can create your own type with `enum`, its syntax is:

```
enum <category_name> {
  <value1_name> = <value1_int>,
  <value2_name> = <value2_int>,
  <...>,
};
```

Where you replace:

- `<category_name>` with the name of what the values represent
- `<value1_name>` with the name of something you want to give a value to
- `<value2_int>` with the number you want C to use for that name
  - You can create as many values as you want separated by commas

You should define an `enum` just below the includes, and not within a function

We Could Create an `enum` That Represents a Month

```
enum month {
    JANUARY = 1,
    FEBRUARY = 2,
    MARCH = 3,
    APRIL = 4,
    MAY = 5,
    JUNE = 6,
    JULY = 7,
    AUGUST = 8,
    SEPTEMBER = 9,
    OCTOBER = 10,
    NOVEMBER = 11,
    DECEMBER = 12,
};
```

An enum is Basically an int, But Instead You Can Use Names

```
bool isWinterSemester(enum month month) {
    return month == JANUARY
           || month == FEBRUARY
           || month == MARCH
           || month == APRIL;
}

int main(void) {
    enum month month;
    printf("Enter a month (1-12): ");
    scanf("%d", &month);
    if (isWinterSemester(month)) {
        printf("The month is probably the winter semester\n");
    }
    else {
        printf("The month is not in the winter semester\n");
    }
    return EXIT_SUCCESS;
}
```

We Could Create an `enum` That Represents a Direction

```
enum direction {
    NORTH = 1,
    EAST,
    SOUTH,
    WEST,
};
```

- If we don’t specify an integer value for the rest of the values,
  - C creates values by just incrementing the integers sequentially
    - If you don’t specify any values, the first value is by default `0`

The above is equivalent to:

```
enum direction {
    NORTH = 1,
    EAST = 2,
    SOUTH = 3,
    WEST = 4,
};
```

Creating a Function to Print What the Value Represents

```c
void printDirection(enum direction d) {
    if (d == NORTH) {
        printf("North\n");
    }
    else if (d == EAST) {
        printf("East\n");
    }
    else if (d == SOUTH) {
        printf("South\n");
    }
    else if (d == WEST) {
        printf("West\n");
    }
    else {
        exit(EXIT_FAILURE);
    }
}
```

Instead of Many `if`s that Check a Value, Use a `switch`

The syntax of a `switch` statement is:

```
switch (<variable>) {
case <value1>:
case <value2>:
<...>
}
```

C will skip to the `case` statement for the matching value and start running code

- It'll continue running (any other `case` statement is ignored) until:
  - a `break;` statment, skipping to the closing } for the `switch`, or
  - it runs until the closing } for the `switch`

We can use `default:` to represent where to go if there is not a match

- Otherwise, if there's no match, we skip to the end

Re-writing the Previous Function to Use a `switch` Statement

```c
void printDirection(enum direction d) {
    switch (d) {
    case NORTH:
        printf("North\n");
        break;
    case EAST:
        printf("East\n");
        break;
    case SOUTH:
        printf("South\n");
        break;
    case WEST:
        printf("West\n");
        break;
    default:
        exit(EXIT_FAILURE);
    }
}
```

You Can Rename Types with `typedef`

The syntax of a `typedef`, is:

```
  typedef <type> <new_name>;
```

Where you replace:

- `<new_name>` by the name of whatever you'd like to name your type
- `<type>` by the type you would like to use when you use `<new_name>`

You Can Rename Types with `typedef`

The syntax of a `typedef`, is:

```
typedef <type> <new_name>;
```

Where you replace:

`<new_name>` by the name of whatever you'd like to name your type
`<type>` by the type you would like to use when you use `<new_name>`

For example, you could write:

```
typedef int number_t;
```

Aftewards, you could declare variables with type `number_t`, then later change all your types by modifying to `typedef double number_t;`

Note, usually you append `_t` to the name to indicate it's a type

Generally, Creating a `typedef` For Numbers is a Bad Idea

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

typedef int number_t;

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

What happens if we change to `typedef double number_t;`?

A Typical Use of `typedef` Is to Save Us from Writing `enum`

You’re able to create an `enum` without giving it a name, you may write:

```
typedef enum {
    NORTH = 1,
    EAST,
    SOUTH,
    WEST,
} direction_t;
```

Afterwards, you can create a variable with:

```
direction_t direction = NORTH;
```

Final Exercise, Going Back to String Recursion

Can we implement `strchr` recursively?