Lecture #08 – More Strings, Pointers
Recap

• Characters
• Strings
• Common String Operations
  – Comparing
  – Copying
  – Concatenating
  – Substrings
Plan for Today

• String Diamond
• Searching in Strings
• Pointers

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli’s Stanford CS107 class
Lecture Plan

• String Diamond
• Searching in Strings
• Pointers
String Diamond

• Write a function `diamond` that accepts a string parameter and prints its letters in a "diamond" format as shown below.
  • For example, `diamond("COMP201")` should print:

```
C
CO
COM
COMP
COMP2
COMP20
COMP201
OMP201
MP201
P201
201
01
1
```
Practice: Diamond
Lecture Plan

• String Diamond
• Searching in Strings
• Pointers
C Strings

C strings are arrays of characters ending with a null-terminating character \"\0\".

String operations such as strlen use the null-terminating character to find the end of the string.

Side note: use strlen to get the length of a string. Don’t use sizeof!
## Common `string.h` Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strlen(str)</code></td>
<td>returns the # of chars in a C string (before null-terminating character).</td>
</tr>
<tr>
<td><code>strcmp(str1, str2)</code>, <code>strncmp(str1, str2, n)</code></td>
<td>compares two strings; returns 0 if identical, &lt;0 if <code>str1</code> comes before <code>str2</code> in alphabet, &gt;0 if <code>str1</code> comes after <code>str2</code> in alphabet. <code>strncmp</code> stops comparing after at most <code>n</code> characters.</td>
</tr>
<tr>
<td><code>strchr(str, ch)</code></td>
<td>character search: returns a pointer to the first occurrence of <code>ch</code> in <code>str</code>, or NULL if <code>ch</code> was not found in <code>str</code>. <code>strrchr</code> finds the last occurrence.</td>
</tr>
<tr>
<td><code>strrchr(str, ch)</code></td>
<td></td>
</tr>
<tr>
<td><code>strstr(haystack, needle)</code></td>
<td>string search: returns a pointer to the start of the first occurrence of <code>needle</code> in <code>haystack</code>, or NULL if <code>needle</code> was not found in <code>haystack</code>.</td>
</tr>
<tr>
<td><code>strcpy(dst, src)</code>, <code>strncpy(dst, src, n)</code></td>
<td>copies characters in <code>src</code> to <code>dst</code>, including null-terminating character. Assumes enough space in <code>dst</code>. Strings must not overlap. <code>strncpy</code> stops after at most <code>n</code> chars, and does not add null-terminating char.</td>
</tr>
<tr>
<td><code>strcat(dst, src)</code>, <code>strncat(dst, src, n)</code></td>
<td>concatenate <code>src</code> onto the end of <code>dst</code>. <code>strncat</code> stops concatenating after at most <code>n</code> characters. Always adds a null-terminating character.</td>
</tr>
<tr>
<td><code>strspn(str, accept)</code>, <code>strcspn(str, reject)</code></td>
<td><code>strspn</code> returns the length of the initial part of <code>str</code> which contains only characters in <code>accept</code>. <code>strcspn</code> returns the length of the initial part of <code>str</code> which does not contain any characters in <code>reject</code>.</td>
</tr>
</tbody>
</table>
Searching For Letters

strchr returns a pointer to the first occurrence of a character in a string, or NULL if the character is not in the string.

```c
char daisy[6];
strcpy(daisy, "Daisy");
char *letterA = strchr(daisy, 'a');
printf("%s\n", daisy); // Daisy
printf("%s\n", letterA); // aisy
```

If there are multiple occurrences of the letter, strchr returns a pointer to the first one. Use `strrchr` to obtain a pointer to the last occurrence.
Searching For Strings

`strstr` returns a pointer to the first occurrence of the second string in the first, or NULL if it cannot be found.

```c
char daisy[10];
strcpy(daisy, "Daisy Dog");
char *substr = strstr(daisy, "Dog");
printf("%s\n", daisy); // Daisy Dog
printf("%s\n", substr); // Dog
```

If there are multiple occurrences of the string, `strstr` returns a pointer to the first one.
String Spans

strspn returns the *length* of the initial part of the first string which contains only characters in the second string.

```c
char daisy[10];
strcpy(daisy, "Daisy Dog");
int spanLength = strspn(daisy, "aDeoi"); // 3
```

“How many places can we go in the first string before I encounter a character *not in* the second string?”
String Spans

`strcspn(c = “complement”)` returns the *length* of the initial part of the first string which contains only characters *not* in the second string.

```c
char daisy[10];
strcpy(daisy, "Daisy Dog");
int spanLength = strcspn(daisy, "driso");    // 2
```

“How many places can we go in the first string before I encounter a character *in* the second string?”
When we pass a string as a parameter, it is passed as a `char *`. We can still operate on the string the same way as with a `char[]`. *(We’ll see why today!)*

```c
int doSomething(char *str) {
    char secondChar = str[1];
    ...
}
```

// can also write this, but it is really a pointer
```c
int doSomething(char str[]) { ...
```
Arrays of Strings

We can make an array of strings to group multiple strings together:

```c
char *stringArray[5];    // space to store 5 char *s
```

We can also use the following shorthand to initialize a string array:

```c
char *stringArray[] = {
    "Hello",
    "Hi",
    "Hey there"
};
```
Arrays of Strings

We can access each string using bracket syntax:

```c
printf("%s\n", stringArray[0]); // print out first string
```

When an array is passed as a parameter in C, C passes a *pointer to the first element of the array*. This is what `argv` is in `main`! This means we write the parameter type as:

```c
void myFunction(char **stringArray) {

// equivalent to this, but it is really a double pointer
void myFunction(char *stringArray[]) {
```
Write a function `verifyPassword` that accepts a candidate password and certain password criteria and returns whether the password is valid.

```c
bool verifyPassword(char *password, char *validChars, char *badSubstrings[], int numBadSubstrings);
```

`password` is valid if it contains only letters in `validChars`, and does not contain any substrings in `badSubstrings`. 
bool verifyPassword(char *password, char *validChars, char *badSubstrings[], int numBadSubstrings);

Example:
char *invalidSubstrings[] = { "1234" };

bool valid1 = verifyPassword("1572", "0123456789", invalidSubstrings, 1);  // true
bool valid2 = verifyPassword("141234", "0123456789", invalidSubstrings, 1);  // false
Practice: Password Verification

verify_password.c
Lecture Plan

• String Diamond
• Searching in Strings
• Pointers
Pointers

• A *pointer* is a variable that stores a memory address.
• Because there is no pass-by-reference in C like in C++, pointers let us pass around the address of one instance of memory, instead of making many copies.
• One (8 byte) pointer can refer to any size memory location!
• Pointers are also essential for allocating memory on the heap, which we will cover later.
• Pointers also let us refer to memory generically, which we will cover later.
Memory

• Memory is a big array of bytes.
• Each byte has a unique numeric index that is commonly written in hexadecimal.
• A pointer stores one of these memory addresses.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x105</td>
<td>'\0'</td>
</tr>
<tr>
<td>0x104</td>
<td>'e'</td>
</tr>
<tr>
<td>0x103</td>
<td>'l'</td>
</tr>
<tr>
<td>0x102</td>
<td>'p'</td>
</tr>
<tr>
<td>0x101</td>
<td>'p'</td>
</tr>
<tr>
<td>0x100</td>
<td>'a'</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Memory

• Memory is a big array of bytes.
• Each byte has a unique numeric index that is commonly written in hexadecimal.
• A pointer stores one of these memory addresses.
Looking Closely at C

• All parameters in C are “pass by value.” For efficiency purposes, arrays (and strings, by extension) passed in as parameters are converted to pointers.

• This means whenever we pass something as a parameter, we pass a copy.

• If we want to modify a parameter value in the function we call and have the changes persist afterwards, we can pass the location of the value instead of the value itself. This way we make a copy of the address instead of a copy of the value.
Pointers

int x = 2;

// Make a pointer that stores the address of x.
// (& means "address of")
int *xPtr = &x;

// Dereference the pointer to go to that address.
// (* means "dereference")
printf("%d", *xPtr);    // prints 2
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);    // 3!
    ...
}
```
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...  
}
```
**Pointers**

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x); // 3!
    ...
}
```

STACK

```
main
x 2
```
Pointers

A pointer is a variable that stores a memory address.

void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```

...
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);   // 3!
    ...
}
```
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x); // 3!
    ...
}
```
A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);     // 3!
    ...
}
```

![STACK diagram](image-url)
Pointers

A pointer is a variable that stores a memory address.

void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```

Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x); // 3!
    ...
}
```
Pointers

A pointer is a variable that stores a memory address.

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f0</td>
<td>3</td>
</tr>
</tbody>
</table>
```

...
Pointers Summary

• If you are performing an operation with some input and do not care about any changes to the input, **pass the data type itself**. This makes a copy of the data.

• If you are modifying a specific instance of some value, **pass the location** of what you would like to modify. This makes a copy of the data’s location.

• If a function takes an address (pointer) as a parameter, it can **go to** that address if it needs the actual value.
Pointers

Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);  // 2!
    ...
}
```
Pointers

Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x); // 2!
    ...
}
```
Pointers

Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);  // 2!
    ...
}
```

STACK

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f0</td>
<td>2</td>
</tr>
<tr>
<td>0x10</td>
<td>2</td>
</tr>
</tbody>
</table>
Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);  // 2!
    ...
}
```
Pointers

Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);  // 2!
    ...
}
```

STACK
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x</td>
<td>0x1f0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>val</td>
<td>0x10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Pointers

Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x); // 2!
    ...
}
```

Pointers

Without pointers, we would make copies.

```c
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);    // 2!
    ...
}
```

STACK

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f0</td>
<td>2</td>
</tr>
</tbody>
</table>

main()
Recap

• String Diamond
• Searching in Strings
• Pointers

Next time: Arrays and Pointers