Contents

Part I

String Functions

This is a straightforward PDF of simple implementation of functions present in `string.h` header file. The following are all the functions alongside the type and there parameters. All these are NTBS

- `char *strcat(char *dest, const char *src) {}`
- `char *strncat(char *dest, const char *src, size_t n) {}`
- `char *strchr(const char *str, int c) {}`
- `char *strrchr(const char *str, int c) {}`
- `int strcmp(const char *s1, const char *s2) {}`
- `int strncmp(const char *s1, const char *s2, size_t n) {}`
- `size_t strspn(const char *s1, const char *s2) {}`
- `size_t strcspn(const char *s1, const char *s2) {}`
- `char *strpbrk(const char *s1, const char *s2) {}`
- `char *strstr(const char *s1, const char *s2) {}`
- `char *strtok(char *str, const char *delim) {}`
- `char *strcpy(char *dest, const char *src) {}`
- `char *strncpy(char *dest, const char *src, size_t n) {}`
- `char *strdup(const char *src) {}`
- `char *strndup(const char *src, size_t n) {}`
- `void *memchr(const void *ptr, int ch, size_t n) {}`
- `int memcmp(const void *lhs, const void *rhs, size_t n) {}`
- `void *memset(void *dest, int ch, size_t n) {}`
- `void *memcpy(void *dest, const void *src, size_t n) {}`
- `void *memmove(void *dest, const void *src, size_t n) {}`
- `void *memccpy(void *dest, const void *src, int c, size_t n) {}`

1. `strcat`:
   The function concatenates two strings, or basically appends the second string to the first string. Algorithmically, we want to return `res`, to start off we append characters from destination into result, then append from `src` and then return. In C we do it via pointers:

```c
char *strcat(char *dest, const char *src) {
  while(*dest) dest++;
  while(*src) *dest++ = *src++;
  *dest = '\0';
  return dest;
}

int main() {
  char dest[50] = "Hello, ";
  const char *src = "world!";
  strcat(dest, src);
  printf("%s\n", dest);
  return 0;
}
```

   (a) We assign more than enough space in `dest` in case `src` is large
   (b) As per the function definition it can be seen that `src` is `const` (So no changes to it via the function) but `dest` will have changes being made
   (c) More specifically we first traverse through the character array of `dest`, till we come across null byte `\0`, then we start assigning characters from `src` to `dest`

```c
*dest = *src
dest++;
src++;
```

   (d) At the end we assign `\0` to make it null terminate and return it.

2. `strncat`:
   The function append only part of second string onto first.

```c
char *strcat(char *dest, const char *src, size_t n) {
  while(*dest) dest++;
  while(n-- && *src) *dest++ = *src++;
  *dest = '\0';
  return dest;
}

int main() {
  char dest[50] = "Hello, ";
  const char *src = "world!";
  strcat(dest, src);
  printf("%s\n", dest);
  return 0;
}
```

3. `strchr`:
   Returns pointer to the first occurrence of character in the input string. ALgo-

rithmically, we just traverse the string via a pointer and see when the pointer equals the target character. If we come across null byte (End of string) then we return false

```c
char *strcat(const char *s, int c) {
  while(*s) {
    if(*s == (char)c) return (char *)s;
    s++;
  }
  if(*s == (char)c) return (char *)c;
  return NULL;
}

int main() {
  const char *str = "Hello, world!";
  int target = 'w';
  if(strcat(str, target)) printf("Character %c found %d",
  ↪  target, strcat(str, target) - str);
  else printf("Character %c not found in %s", target, str);
  return 0;
}
```

(a) The function type is `char*` meaning it returns pointer to a character. We use `int` for flexibility (ASCII).
(b) The function itself takes a pointer and traverses the string until it comes across the target character. If not found then returns null.
(c) The string is an array of characters will decays to a pointer so we can subtract it with `char*`. The value of `strcat(str, target) - str`
(d) If `str` points to "Hello, world" and `strcat(str, target)` points to `'w'` then we have `'w' - "Hello, world!"` or basically, `1007 - 1000 / sizeof(char) = 7`. (Where 1000 is starting address of `str[0]`)

4. `strnchr`:
This function also finds the first occurrence of character in string only searching within the first n character. The function takes in another argument for the number of characters.

```c
char *strncat(const char *s, int c, size_t n) {
  while(n-- && *s) {
    if(*s == (char)c) return (char *)s;
    s++;
  }
  if(*s == (char)c) return (char *)c;
  return NULL;
}

int main() {
  const char *str = "Hello, world!";
  int target = 'w';
  if(strncat(str, target, 8)) printf("Character %c found %d",
  ↪  target, strncat(str, target, 8) - str);
  else printf("Character %c not found in %s", target, str);
  return 0;
}
```

5. `strcmp`:
   The function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached.

```c
int strcmp(const char *s1, const char *s2) {
  while(*s1 && (*s1 == *s2)) { s1++; s2++; }
  return (unsigned char)*s1 - (unsigned char)*s2;
}
```

   (a) We continue the loop provided the characters are same and that the string hasn't ended yet. If result is 0 then it signifies that every character in both strings matches until end of one string is reached.
   (b) If it is positive then `s1` is greater than second string `s2` which occurs when ASCII value are different (More in case of `s1`).
   (c) If it is negative then `s1` is lesser than second, which happens when ASCII value of character in `s1` is less than the corresponding one in `s2`.

6. `strncmp`:
   The function just compares both strings up to a specified number of characters

```c
#define uc unsigned char
int strncmp(const char *s1, const char *s2, size_t n) {
  while(n-- && *s1 && *s2) {
    if(*s1 ! *s2) return (uc)*s1 - (uc)*s2;
    s1++; s2++;
  }
  return (n == (size_t)-1) ? 0 : (uc)*str1 - (uc)*str2;
}
```

7. `strspn`:
   Returns the length of the initial portion of `s1` which consists only of characters that are part of `s2`. Algorithmically, we just traverse through `s2` and check if it is in `s1` and check when it doesn't match.

```c
size_t strspn(const char *s1, const char *s2) {
  size_t cnt = 0;
  while(*s1) {
    const char *a = s2;
    while(*a) {
      if(*s1 == *a) { cnt++; break; }
      a++;
    }
    if(!*a) break;
    s++;
  }
  return cnt;
}
```

   (a) We start iterating through `s1`, then check if the current character is also in `s2`. If it is then we increase the count and break outside to continue with the next character in `s1`
   (b) If not then check other characters (As we only need check with the set of

characters in `s2`). (Note, `strspn(1234567890, 18289) = 2`).

(c) If we didn't find a match then we break from the outer loop.

8. `strcspn`:
   Also called the string complement span. It goes and checks which characters are common within the set of characters present within `s1` and `s2` and then rejects the characters present in `s2`. Very similar in working to `strspn`.

```c
size_t strcspn(const char *s, const char *reject) {
  size_t cnt = 0;
  while(*s) {
    const char *r = reject;
    while(*r) {
      if(*s == *r) return cnt;
      r++;
    }
    cnt++; s++;
  }
  return cnt;
}

int main() {
  char key[] = "fcba73";
  char nums[] = "1234567890";
  int i = strcspn(str, nums);
  printf("Incorrect characters: %d. Only letters are
    ↪ allowed.",i+1);
  return 0;
}
```

   (a) The outer loop iterates through the string `s1` and goes on until the null terminator is reached.
   (b) The inner loop is used to iterate through `s2` and checks if current from `s1` matches any in `s2`. If match is found the function returns current count, which represents initial segment that does not include any rejected characters.

9. `strpbrk`:
   It stands for string pointer break and is used to find the first occurrence of a character within a string. It's structure is very similar to the two above. Note, it is not same as `strchr`

   - Use `strchr` when we need to find a specific character
   - Use `strpbrk` when we want to check presence of any character belonging to a set.

```c
char *strpbrk(const char *s, const char *accept) {
  while(*s) {
    const char *a = accept;
    while(*a) {
      if(s* == *a) return (char *)s;
      a++;
    }
    s++
  }
  return NULL:
}
```

10. `strstr`:
    This helps to find the first occurrence of a substring `s2` within another string `s1`. Algorithmically, we traverse via character in `s1` and if there is a match then we increment pointers of `s1` and `s2` or else go to the next character in `s1` or return null.

```c
char *strstr(const char *haystack, const char* needle) {
  while(*haystack) {
    const char *h = haystack;
    const char *n = needle;
    while(*n && (*h == *n)) { h++; n++; }
    if(!*n) return (char *)haystack;
    haystack++;
  }
  return NULL;
}
```

11. `strtok`:
    Stands for string token. It breaks up a string into tokens on the basis of given delimiters. On a first call, the function expects a C string as argument for `str`, whose first character is used as the starting location to scan for tokens. In subsequent calls, the function expects a null pointer and uses the position right after the end of the last token as the new starting location for scanning.

```c
char *strtok(char *str, const char *dl) {
  static char *last = NULL;
  if(str) last = str;
  if(!last) return NULL;
  while(*last && strchr(dl, *last)) last++;
  if(!*last) {
    last = NULL;
    return NULL;
  }

  char *token_start = last;
  while(*last && !strchr(dl, *last)) last++;
  if(*last) {
    *last = '\0';
    last++;
  } else last = NULL;
  return token_start;
}

int main() {
  char str[] = "apple,banana,,orange";
  const char *delim = ", ";
  char *token = strtok(str, delim); // first call
  while(token != NULL) {
    printf("Token: %s\n", token);
    token = strtok(NULL, delimiters);
  }
  return 0;
}
```

(a) The purpose of `static char* last` is used to remember where the last token extraction left off. It's helpful because of the way it is used in actual programs

(b) Actually on first call you provide string to be tokenized. On subsequent calls, you pass `NULL` to continue where you left off. The `const char *dl` is the parameter specifying which characters are considered as delimiters.

(c) The condition `while(*last && strchr(dl, *last))` moves `last` forward until it finds a character that is not a delimiter or reaches end of string. After skipping delimiters we check `if (!*last)` just in case

(d) The line `char *token_start = last;` marks the start of a token and saves current position of start. To find the end of the this token we use `while (*last && !strchr(dl, *last)) last++;`.

(e) After finding a delimiter we replace it with a null terminator, which effectively ends our current token. If no token was found then `last = NULL`.

12. `strdup`:
This function duplicates a string into another string. It is quite easy algorithmically. Basically, in C we need to allocate enough memory for the string, so we use `malloc`. Afterwards we allocate memory and copy characters of original into the duplicate

```c
char *strdup(const char *str) {
  char *dup = (char *)malloc(strlen(s) + 1);
  if(!dup) return NULL;
  strcpy(dup, s);
  return dup;
}

int main() {
  const char *orig = "Hello";
  char *duplicate = strdup(original);
  if (duplicate) {
    printf(...);
    free(duplicate);
  } else printf(...);
}
```

(a) We allocate memory for the new string and one more for the null terminator. We return `NULL` if memory allocation fails.

(b) We copy the string into the newly allocated memory and then return the pointer to the duplicated string

```c
char *strdup(const char *s) {
  const char *t = s;
  while(*temp) temp++;
  size_t len = t - s;
  char *dup = (char *)malloc(len + 1);
  if(!dup) return NULL;
  char *d = dup;
  while(*s) *d++ = *s++;
  *d = '\0';
  return dup;
}
```

(a) Basically, without in-built functions we need to figure out the length of the original string, so we make a new temporary string and use that to find length from first character `s` to null terminator `t`.

(b) Then we allocate that memory and check if the memory allocation failed. Otherwise, we manually copy each character from `s` to `d` via `*d++ = *s++` and finally null terminate it. Then we return it.

13. `strndup`:
This function duplicates the string up to a specified number of characters.

```c
char *strndup(const char *s, size_t n) {
  char *dup = (char *)malloc(n+1);
  if(!dup) return NULL;
  char *d = dup;
  size_t cnt = 0;
  while(cnt < n && *s) {
    *d++ = *s++;
    cnt++;
  }
  *d = '\0';
  return dup;
}
```

14. `memchr`:
    The `memchr` function is used to search for a specific character in a block of memory. It takes in three parameters: A pointer to the memory block, a character to search for and the number of bytes to check. It doesn't stop checking when encountering a null character like in `strchr`, also it's faster.

```c
#define <stddef.h>
void *memchr(const void *s, int c, size_t ) {
  const unsigned char *p = s;
  for(size_t i=0; i < n; i++) {
    if(p[i] == (unsigned char)c) return (void *)(p + i);
  }
  return NULL;
}

int main() {
  const char *str = "Hello, world!";
  char target = 'w';
  size_t len = 13;
  char *res = memchr(str, target, len);
  // ...
}
```

15. `memcmp`:
    This function compares two blocks of memory. It assesses the contents byte by byte and determines their relative ordering based on the values of the bytes. It takes in pointer to first and second block as well as number of bytes to compare.

```c
int memcmp(const void *s1, const void *s2, size_t n) {
  const unsigned char *p1 = s1;
  const unsigned char *p2 = s2;
  for(size_t i=0; i<; i++) {
    if(p1[i] != p2[i]) return p1[i] - p2[i];
    // Returns difference if not equal
  }
  return 0;
}
```

16. `memset`:

The `memset` function is used to fill up a block of memory with a specific value. It takes in the pointer to the memory block to be filled, the value to set in the memory block and number of bytes

```c
void *memset(void *s, int c, size_t n) {
  unsigned char *p = s;
  for(size_t i=0; i < n; i++) {
    ptr[i] = (unsigned char)c;
  }
  return s;
}

int main() {
  char buf[50];
  memset(buf, 'A', sizeof(buffer) - 1);
  buffer[49] = '\0';
  // ...
}
```

17. `memcpy:`
    The `memcpy` function is used to copy a specified number of bytes from one memory location to another. It takes in pointer to the destination memory block, a pointer to the source memory block and number of bytes to copy. It returns pointer to destination memory block.

```c
void *memcpy(void *dest, const void *src, size_t n) {
  unsigned char *d = dest;
  const unsigned char *s = src;
  for(size_t i=0; i < n; i++) d[i] = s[i];
  return dest;
}

int main() {
  char src[] = "Hello, world!";
  char dest[50];
  memcpy(dest, src, strlen(src) + 1);
  return 0;
}
```

18. `memmove:`
    This is used to copy a block of memory from one location to another. This accounts for overlapping regions unlike `memcpy`. It takes in pointers for destination and source memory blocks as well as number of bytes, returns pointer to destination memory block

```c
void *memmove(void *dest, void *src, size_t n) {
  unsigned char *d = dest;
  const unsigned char *s = src;
  if(d < s) {
    for(size_t i=0; i<n; i++) d[i] = s[i];
  } else if(d > s) {
      for(size_t i=n; i>0; i--) d[i-1] = s[i-1];
  }

  return dest;
}

int main() {
  char str[] = "Hello, World!";
  memmove(str + 7, str, 6); // Move "Hello," to "World!"
  printf("Result after memmove: %s\n", str); // Should print
   ↪  "Hello, Hello!"
  return 0;
}
```

19. `memccpy`:
    This is a specialized version of `memcpy` which works the same but stops after encountering a specific character. Thus, it takes in the extra parameter for the character to search for during copying operation. It returns pointer to the byte *immediately* after the last occurrence of character `c` is destination memory block if it is found within specified number of bytes

```c
void *memccpy(void *dest, const void *src, int c, size_t n) {
  unsigned char *d = dest;
  const unsigned char *s = src;
  for(size_t i=0; i<n; i++) {
    d[i] = s[i];
    if(s[i] == (unsigned char)c) return (void *)(d + i + 1);
  }
  return NULL;
}

int main() {
  const char *src = "Hello, World!";
  char dest[50];
  char *result = memccpy(dest, src, ',', sizeof(dest));
  if (result != NULL) {
    *result = '\0';
    printf("Destination: %s\n", dest); // "Hello,"
  } else // ...
  return 0;
}
```

Part II
Appendix