Introduction to Computing

String operations, Preprocessors

Recap

- Pointers
  - Value
  - Name
  - Address
  - Size and type of pointers
- Array and pointers
- Function and pointers

- Functions calling functions
- Recursion
- Passing Array to functions
  - Problems with sending size
- Character Array
- Strings

Character Arrays and Strings

- Character arrays are very useful in storing data
  - Even though they are basically integers underlying, but the range of the values are limited
  - This allows to have some additional functionalities (for convenience, of course)
  - Strings are declared and defined the same way as any other array types
  - Since the values are in range of 0-127 (sometimes more, but still, limited), we have the convenience make some of the characters for special use such as:
    - newline(\n)
    - backspace (\b), etc.
  - In the case of character arrays we use a special character called the null character
    - Represented as ‘\0’ (backslash-zero)
    - Ascii value of this character is 0
    - It prints nothing on the computer screen

Character array and strings

- Character variable
  - char ch1, ch2 = ‘a’;
- Character array
  - char ca1[10];
  - char ca2[3] = {‘S’,‘D’,‘B’};
  - char ca3[5] = {‘S’,‘D’,‘B’};

A string is a character array for which the last valid character is the null character.

- char ca4[10] = {‘S’,‘o’,‘u’,‘m’,‘a’,‘d’,‘i’,‘p’,‘\0’};
- char ca5[10] = “Soumadip”;
  - *Both the above statements are equivalent*
  - This type of initialization makes sure that the null character is automatically appended at the end

You *can't do the following after declaration* though

ca1 = "word1"; // not allowed – why?
ca4 = "word2"; // not allowed – what is the type of ca1 or ca4?

-- More on what can and can't be done, later

String is basically short for “a string of characters”

- A single character in C is written within single quotes e.g. ‘a’, ‘3’, ‘Z’, ‘%’, etc.
- A string is written in C within double quotes, e.g., “a_string”, “with spaces”, “and with $”, etc.

Strings and scanf

- scanf also provides a shortcut for strings format **%s**
  - scanf (“%s”, ch_arr); ⇒ this allows you to read a string from user without spaces
  - scanf ("%[^ ^\n]%*c", ch_arr);
    - This is equivalent to **%s**; reads the characters until space ( ) or the newline character (\n) is encountered
  - scanf ("%[^\n]%*c", ch_arr);
    - reads a string with spaces until a newline(\n); so, it can read strings with spaces

Note: *All the method discussed here will add a ‘\0’ to the end of the scanned characters - making it a string*

String Operations (Two ways)

- Normal assignment operators ***do not work*** on strings *(Nor on any kind of arrays for that matter)*
- You need to define different operation on strings by writing your own functions
  - Compare two strings for equality
  - Copy one string to another
  - Concatenate two strings
  - Check if a input string is integer or float

- Alternatively, you can **#include** *a new header file called* **string.h** *and use built-in functions for such operations*

int **strlen**(const char *str)
int **strncmp**(const char *str1, const char *str2, int n)
char* **strstr**(const char *haystack, const char *needle)
char* **strcat**(char *dest, const char *src)

String Operations Without using string.h

Solved Examples:

- Finding string length
- Concatenating strings
- Comparing strings

Try yourselves

- Copy one string to another
- Check if a input string is integer or float
- Duplicate strings
- Change a string to uppercase/lowercase

Manual String Length:

```
int str_length(char str[]) {
  int length = 0;
  while (str[length] != '\0') {
    length++;
  }
  return length;
}
```

More String Operations Without using string.h

Manual String Concatenation:

```
void concat(char dest[], char src[])
{
  int i = 0, j = 0;
  while (dest[i] != '\0') i++;
  while (src[j] != '\0') {
    dest[i] = src[j];
    i++;
    j++;
  }
  dest[i] = '\0';
}
```

Manual String Comparison:

```
int compare(char str1[], char str2[])
{
  int i = 0;
  while (str1[i] == str2[i] && str1[i] !=
'\0') i++;
  if (str1[i] == '\0' && str2[i] == '\0')
return 0;
  return str1[i] - str2[i];
}
```

String Operations with #include<string.h>

- **Some Built-in Functions**:
  - `strlen()`: String length
  - `strcpy()`: Copy strings
  - `strcat()`: Concatenate strings
  - `strcmp()`: Compare strings
  - strstr(): Locate substring in another string

Usage Example:

```
char str1[20] = "Hello";
char str2[20];
strcpy(str2, str1);  // Copy str1 into str2
```

Find a Substring (`strstr()`):

```
char str[] = "I love programming";
char *sub = strstr(str, "love");
if (sub != NULL) {
  printf("Found substring at: %s\n", sub);
}
```

Output: "Found substring at: love programming"

#include<string.h>

```
char str1[20]="A string", str2[20]="Another string"; char ch='r'; int n=4;

strlen (str1)                 // gives the length of the string ⇒ 8
strcpy (str2,str1)            // copies str1 into str2
strncpy (str2, str1, n)       // copies first n characters from str1 into str2
strcmp (str1, str2)           // returns 0 if both strings are the same
strcmpi (str1, str2)          // compares two strings ignoring the case
strcat (str1, str2)           // concatenates str2 at the end of str1
strchr (str1, ch)             // finds the position(pointer) of first ch in str1
```

Link to more/all string.h functions with examples.

More operations on strings

- Split strings into words
- Split strings based on a given delimiter
- Find the longest string in an array of strings
- Sort an array of strings alphabetically
- Counting Words in a Sentence
- Join an array of words into a single string with space

Useful ways to handle multiple strings in your code

- *Array of strings*

```
char arr[3][10] = {"IACS", "UG", "2020"};
```

- *Array of pointers to strings*

```
char *arr[] = {"IACS", "UG", "2020"};
```

--- *we will learn more on these^^^ declarations later*

Preprocessors/ Macro

- Preprocessor is not a part of the compiler
- It is a step in the compilation process
- a C Preprocessor is just a text substitution tool
- It instructs the compiler to do required pre-processing before the actual compilation
- They are also known as *macro*

Examples:

- #inlcude <string.h>
- #define SIZE 10
- #define SQUARE(x) ((x)*(x))
- #ifdef <macro>.. #endif
- etc.

Preprocessor Directives in Depth

- **#define**: Used to define symbolic constants or macros.
  - Example: `#define PI 3.14`
  - Usage: Replace `PI` with `3.14` throughout the code.
- **#include**: Used to include header files.
  - Example: `#include <stdio.h>`
  - Usage: Inserts the content of the specified file into the program before compilation.
- **#undef**: Undefine a previously defined macro.
  - Example: `#undef PI`

- **#ifdef and #ifndef**: Conditional compilation based on whether a macro is defined or not.
  - Example:

```
#define PI 3.14
#ifdef PI
  printf("PI is defined\n");
#endif
#undef PI
#ifndef PI
  printf("PI is not defined\n");
#endif
```

Conditional Compilation

- **#if, #elif, #else, #endif**
  - Allows sections of code to be conditionally included or excluded.
- **Advantages**:
  - Helps in debugging by selectively compiling parts of the code.
  - Allows platform-specific code.

```
#define LEVEL 2

#if LEVEL == 1
  printf("Beginner level\n");

#elif LEVEL == 2
  printf("Intermediate level\n");

#else
  printf("Advanced level\n");

#endif
```

Macro Functions & Predefined Macros

- **Defining Macros with Arguments**:
  #define SQUARE(x) ((x)*(x))
- **Best Practices**: Use parentheses around macro arguments to avoid precedence issues.

int a = 5; int result = SQUARE(a + 1);

```
#define SQUARE(x) x * x       //bad practice
--> a + 1 * a + 1 ==> 5 + 1*5 +1 = 11
#define SQUARE(x) ((x) * (x)) //best practice
--> (a+1) * (a+1) ==> (5+1)*(5+1) = 25
```

Common Predefined Macros:

- `__FILE__`: Current file name.
- `__LINE__`: Current line number.
- `__DATE__`: Compilation date.
- `__TIME__`: Compilation time.

```
printf("Compiled on %s at %s\n",
                    __DATE__, __TIME__);
```

Practical Applications of Preprocessors

Debugging:
Use `#ifdef DEBUG` blocks to include debugging information.

```
#define DEBUG
#ifdef DEBUG
  printf("Debugging info\n");
#endif
```

Cross-Platform Code:

```
#ifdef _WIN32
  printf("Windows\n");
#else
  printf("Other OS\n");
#endif
```

`#pragma` Directive

- The `#pragma` directive is used to give special instructions to the compiler, such as enabling optimizations, managing warnings, or controlling memory alignment. These are compiler-specific and may not be portable across different compilers.

- **Disabling/Enabling Warnings**

```
#pragma warning(push)
#pragma warning(disable : 4996) // Disable a specific warning
printf("Warning disabled\n");
#pragma warning(pop) // Restore previous warning state
```

- **Optimization Control**

```
#pragma optimize("", off) // Turn off optimization void
my_function() { // code }
#pragma optimize("", on) // Turn on optimization
```

- **Pack Struct Alignment**:

```
#pragma pack(1) // Align structure members to 1-byte
boundaries
struct my_struct { char a; int b; };
#pragma pack() // Reset alignment to default
```