APS105: Computer Fundamentals

Arnav Patil

Department of Electrical and Computer Engineering, University of Toronto

Contents

1 Introduction to Programming Computers

1.1 The Basic Structure of Computers

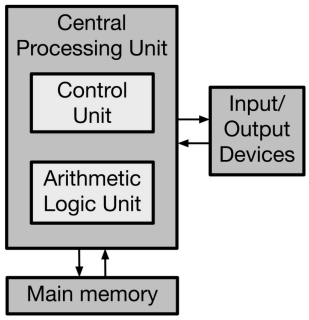

Figure 1: The basic structure of a computer

The main memory stores data and the program being executed. Everything is stored in binary representation.

The CPU is responsible for executing the program's instructions. It is made up of two units: the arithmetic logic unit (ALU) and the control unit.

Input/Output devices are used to pass information into the computer and communicate with the outside world.

1.2 Binary Representation in Memory

Binary is a base-2 number system, with two states: on (1) or off (0).

Memory in RAM is organized in cells, each of which stores a byte and has an address.

1.3 Development Cycle

1. Write the program in an Integrated Development Environment (IDE). In APS105 we use Visual Studio Code (God bless Microsoft). Other popular IDEs you may encounter are PyCharm for Python, Eclipse for Java, and XCode for Swift.
2. Compile the program using a compiler like GCC (GNU Compiler Collection). The compiler takes the human-readable code and converts it into something the machine can understand.
3. Run the program.
4. Debug the program (good luck LOL). Make sure to test your program for edge cases.

Difference Between Compile-time and Run-time

Compile-time is when the function is being compiled, or being converted from human-readable text and numeric code into machine language (1s and 0s) that the machine can actually understand. Run-time, on the other hand, is when the compiled machine code is being executed and input/output is being exchanged with the user.

1.4 Write Simple C Programs

Visit the textbook [click here!] because I refuse to take notes on a "Hello, World!" program.

2 Data Representation and Operations

2.1 Double Data Type for Real Numbers

We'll mainly use two numerical data types in C:

1. `int`: Stores a whole number/integer. Format specifier is `%d`.
2. `double`: Stores a floating point number (decimal). Format specifier is `%lf`.

2.2 Data Types and Representation

There are four types of integer variables in C:

1. `short` - uses 16 bits (2 bytes)
2. `unsigned int` - 32 bits (4 bytes), but the sign bit is not used (limited to positive numbers)
3. `long` - uses 64 bits (4 bytes)
4. `long long` - see above

However, there's only two types of integer variables in C:

1. `float` - 32 bits (4 bytes)
2. `double` - 64 bits (4 bytes), is also the far more commonly used variable for this course

Characters use format specifier `%c` and need 1 byte of storage. Furthermore, they're not actually saved as letters, but rather their ASCII code.

ASCII codes for uppercase letters go from 65 (A) to 90 (Z) and for lowercase letters go from 97 (a) to 122 (z).

Declaring vs Initializing Variables

In your code, you declare a variable as such:

```
    int var;
```

However, right now, that variable is not storing any value. More technically, it's storing whatever "garbage" value it was holding prior to declaration. To *initialize* it, we must use the assignment operator `=`.

Taking in Input From the User

Using the `scanf()` function, we can scan for user input and assign it to the variables we choose. When taking in multiple inputs, the user should use **delimiters** such as tabs, enters, or spaces to separate the different variables.

It is also generally good practice to add a space in order to ignore leading space that the user may accidentally input.

```
    #include <stdio.h>

    int main(void) {
        int num;
        scanf(" %d", &num);
        return 0;
    }
```

2.3 Operations

I refuse to discuss math operations in this collection of notes. There are, however, three important points brought up in this subsection that I will note:

Increment and Decrement Operators

1. `i++` and `++i` are NOT the same thing. `i++` returns the post-increment value, while `++i` returns the pre-increment value.
2. `j = i++` is NOT the same as `j = i + 1`. Rather, `j` is set to the value of `i`, and the value of `i` is incremented on the next line.

Typecasting

By either declaring the variable type before its name, or using `( )` notation, you can manually change the type of the variable in a statement. Useful if you're trying to initialize a double as a quotient of two integers, for example.

`sizeof()` Operator

Returns the size of any variable type in number of bytes. Will be **very** useful later when we discuss memory allocation.

2.4 Math Library

It exists, it's useful, check the textbook.

2.5 Random Numbers

We use the `rand()` function from the `stdlib.h` library. The function returns a value from 0 to `RAND_MAX`, which is the maximum value a positive integer can store.

To get random numbers within a range `[MIN, MAX]`, we use the formula

`rand() % (MAX - MIN + 1) + MIN`.

2.6 Constant Variables

There are times when we want to declare constants - variables whose values cannot be changed, for any reason.

`const` keyword

```
1     const double pi = 3.141592;
```

If the program attempts to change the value of `pi`, then the program will raise an error.

`DEFINE` method

```
1     include <stdio.h>
2     DEFINE PI = 3.141592
```

The preprocessor will set this value to wherever the variable is used, through all functions.

3 Decision-Making Statements

3.1 If-Statements

The `if` statement is a conditional statement that executes a block of code if the condition evaluates to `true`. This implies that the condition MUST be a Boolean statement, evaluating either to `true` or `false`.

```
    if (condition) {
        // code executes if condition is true
    } else {
        // code executes if condition is false
    }
```

Note that if the statement to be executed is a single C statement, then we may alternatively write it as:

```
    if (condition1) return x;
    else if (condition2) return y;
    else return z;
```

3.2 Multiple Conditions

Essentially, if have multiple criteria which we want to apply to a conditional statement, we can use the `and` and `or` operators, represented as `&&` and `||` respectively. For e.g., if you wish to buy a red shirt, and wear a size medium, then here's a simple representation of the logic:

```
    if (size == 'M' && colour == 'R') {
        // buy shirt
    } else {
        // keep looking
    }
```

Be sure to use lazy evaluation to save time and memory when evaluating conditions. Check the textbook if you're not already familiar with this concept.

3.3 Nested If-Statements

Nesting one if-statement within another.

```
    if (age >= 18) {
        printf("Adult\n");
    } else {
        if (age >= 13) {
            printf("Teenager\n");
        } else {
            printf("Child\n");
        }
    }
```

4 Repetition

4.1 While Loops

The while loop allows a block of code to be executed repeatedly so long as the given condition is still true.

```
    while (condition) {
        // code is executed
    }
    printf("Text.\n");
```

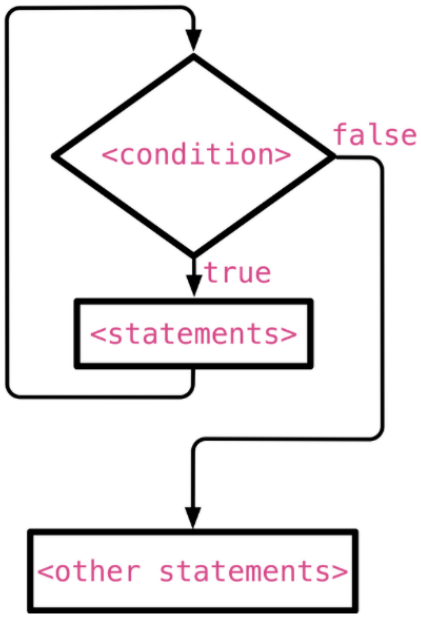

Figure 2: Flow Chart of a While Loop

Note: Be sure not to get yourself stuck in an infinite loop.

4.2 Do-While Loops

The do-while loop is similar to the while loop in that a code block gets executed while the statement is true. However, the difference is that the while loop checks the condition to be true before executing the code. On the other hand, a do-while loop executes the block once, then checks the condition before repeating.

4.3 For Loops

The for loop iterates through a set number of times, which we can control. Programmers generally prefer to use for loops when they know how many times to loop through a code block, and while loops when they don't.

I found this daunting and confusing as well, but with experience, you'll be able to make the correct judgement call.

```
1    for (int i = 0; i < 10; i++) {
2    printf("%d ", i);
3    }
```

Remember that the iteration stops at 1 - 10, so the last number to be printed will be 9.

Another thing to make note of is the scope of the iterating variable. If declared outside of the loop, then you cannot use the variable in a different place unless you re-initialize it.

4.4 Nested Loops

Simple enough, check textbook.

4.5 Debugging For Loops

Learn to use the debugger, it's very straightforward but incredibly powerful. Also, in APS105 there's a lab that requires you to know how to use it.

5 Functions

5.1 Functions

Functions are useful because they break down a larger problem into smaller steps. This method of coding is called **modular programming**.

Though not mentioned in this section (it talks about stars), one note I want to bring up is the recommended way of programming when you have multiple functions.

```
/* Function Declarations */
int square(int x);
double sqrt(int x);
bool isFavourite(void);

/* Main Function */
int main(void) {
    int x = 4;
    int y = square(x);
    double z = sqrt(x);

    bool = favouriteFood;
    favouriteFood = isFavourite();
    return 0;
}

/* Function Definitions */
int square(int x) {
    return x * x;
}

double sqrt(int x) {
    return (double)(pow(x, 0.5));
}

// code goes on
```

5.2 But What is the `main` Function?

The `main` function is the entry point of the entire C program; every program must contain a `main` function. When the program is executed, this is the function that is called. This is why it is best practice to declare other function usages first, then define the `main` function, then define the functions that are called in `main`. If the `main` function returns 0, then the program was executed successfully; if it encounters an error, it will return 1. This is not strictly necessary, however, other programs may rely on the `main` return integer, so it is good practice to include it.

5.3 Communicate From a Function

Covered above, check the textbook if you're still confused.

5.4 Variable Scope

Covered above, check the textbook if you're still confused.

5.5 Pass More Values to a Function

Covered above, check the textbook if you're still confused.

6 Pointers

6.1 Why Do We Need Pointers?

A limitation using using functions with calling by value is that we can only return one value, and have to store it in a variable. Pointers allow us to manipulate multiple values in function, without the need to store the returned value in another variable.

6.2 What Are Pointers?

A pointer, simply put, is a variable that holds the address of another variable.

The reference operator, `&` returns the address of the variable after the operator, and the dereference operator, `*`, returns the value stored in the address after the operator.

```
1    int x = 7; // declares an integer 'x' and initializes to 7.
2    int *p; // declares a pointer to an 'int' type variable.
3    p = &x; // stores the address of 'x' to p.
4    *p = 10; // goes to what is stored in p (address of x) and stores 10
     at that address.
5    printf("%d", x); // would print 10.
```

6.3 How to Use Pointers to Communicate More With Functions?

Consider a function that swaps the value of two variables. Using call by value, we'd only be able to return one variable, but now we can manipulate both variables using pointers.

```
1    int main(void) {
2        int x = 5, y = 8;
3        swap(&x, &y);
4        printf("x=%d, y=%d",x, y);
5        return 0;
6    }
7
8    void swap(int *x, int *y) {
9        int temp = *x;
10       *x = *y;
11       *y = temp;
12       return;
13   }
```

Standard coding practice recommends that between declaration and initialization, the value of the pointer is set to `NULL`, which is a non-valid address (the equivalent of initializing to zero).

6.4 Rules Defining Scope of Variables

Covered above, check the textbook if you're still confused.

7 Arrays

7.1 Why and How to Use Arrays

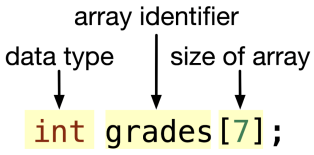

Figure 3: How to Declare an Array

```
1    int array[] = {0, 1, 2, 3, 4} // size of array deduced.
2    int array[5] = {0, 1, 2} // remainder set to zero.
3    int arrays[2] = {0, 1, 2} // "warning: excess elements in array"
```

Remember, array sizes, once declared/initialized, are FIXED.

7.2 What are Arrays, and How are They Stored?

Arrays are a collection of variables with one name and same data type, that are also appearing physically consecutively in memory.

The name of an array is actually a pointer to the first item in the array. Essentially, `array[]` and `*array` are the same.

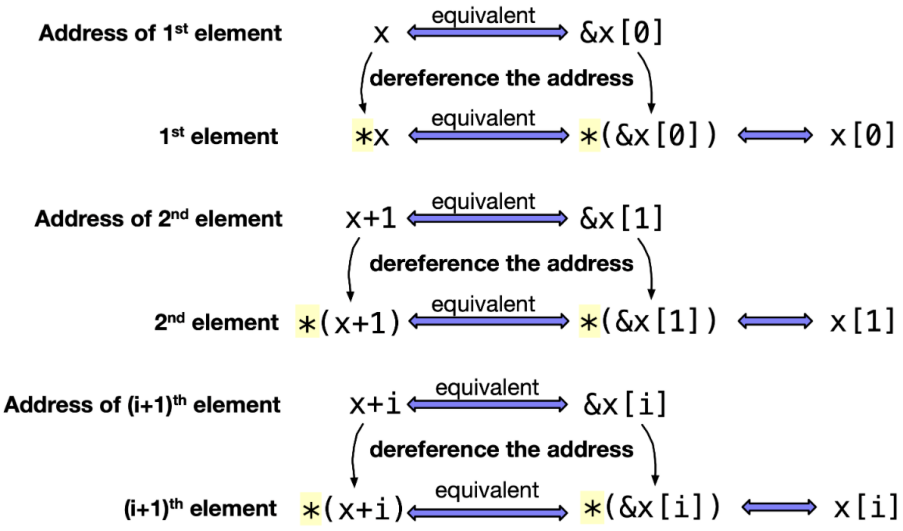

Figure 4: Array Storage and Pointer Arithmetic

7.3 How Do We Pass an Array to a Function?

To pass an array into a function, we only need to pass the pointer to the first element in the array. You must also pass the `size` of the array into the function as a parameter.

```
int sumData(int list[], const int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum = sum + list[i];
    }
    return sum;
}
```

Another thing to note is that for multidimensional arrays of the form `book[chapter][page][sentence][word]`, when passing into a function, we omit the first (or left-most) subdivision, but MUST include the remainder.

Therefore, we would actually pass: `book[][page][sentence][word]`.

8 Dynamic Memory Allocation

8.1 What is Dynamic Memory Allocation?

A computer's memory space has four main segments:

1. Segment that stores code.
2. Segment that stores constants and global variables.
3. Segment that stores dynamically allocated memory, called the **heap**.
4. Segment that stores local variables of a function, called the **stack**.

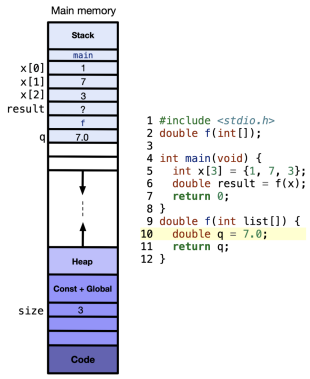

Figure 5: Breakdown of Main Memory

When the size of the required array is unknown at compile-time, we have different options. First, we can set the size of the array to a very large value. Second, we can ask the user to input the size of the array at run-time. Lastly, we can dynamically allocate memory using the `malloc()` or `calloc()` functions.

`malloc()` returns a pointer to an address in memory where we can find the specified amount of available space. It must be typecasted to the variable type that it points to. `calloc()` does the same, but initializes the variable to zero or NULL.

NOTE: if either function is unable to find the specified space in consecutive locations, it will return `NULL`.

NOTE: you must free memory when you use `malloc()` or `malloc()` using the `free()` function. After passing the pointer to the variable to be freed in the `free()` function, set the value of the pointer to `NULL` to fully free the variable.

9 Multi-Dimensional Arrays

9.1 Why and How to Use 2D Arrays?

To declare a 2D array, we need to define both the number of rows and columns.

```
int matrix[2][3]; // 2 rows, 3 columns
```

There are three ways to initialize a 2D array:

```
// enclose individual rows and the whole array
int myArray[2][3] = {{1, 2, 3}, {4, 5, 6}};

// enclose only the whole array
int myArray[2][3] = {1, 2, 3, 4, 5, 6};

// only specify the number of columns
int myArray[][3] = {1, 2, 3, 4, 5, 6};
```

The following figure describes multi-dimensional array representation in main memory.

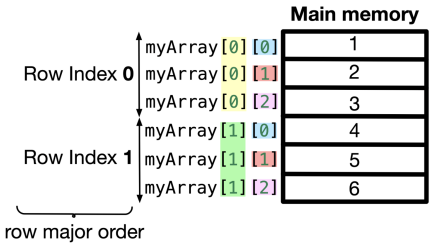

Figure 6: Row Major Representation in Main Memory

9.2 How do We Pass a 2D Array to a Function?

```
void func(int list[][cols], int cols, int rows) {
    list[4][5] = 6;
    return;
}
```

9.3 Dynamic Memory Allocation of 2D Arrays

As discussed in Chapter 8, we use dynamic memory allocation when:

1. We do not know the number of elements in the array before run-time.
2. We want the life-time of the array to be flexible and not dependent on the scope of the array.

Method 1: Dynamic Allocation of an Array of Pointers

1. Dynamically allocate an array of rows pointers.
2. Dynamically allocate each row.
3. Assign a value to each element.
4. Free dynamically allocated space.

Method 2: Static Allocation of an array of pointers

1. Statically allocate an array of pointers.
   (a) The problem with this method is that the pointers will be on the stack instead of the heap.
2. Make each of the above pointers point to a dynamically allocated 1D array.
3. Access the elements of the 2D array.
4. Free only the three 1D arrays, and set the pointers from step 1 to `NULL`.

Method 3: Dynamic Allocation of a 1D Array

Dynamically allocate a 1D array having `row X cols` elements. Then, you can access any element using the formula `*(arr + row * cols + col)`.

When done with the array, free only `array`, which takes care of all elements in the "2D" array.

10 Strings

10.1 What are Strings?

To store a string in a variable, we use an array (of `char` types) that are *null-terminated*. This means that we store the NULL character, `'\0'`, which has an ASCII code of 0, as the last character in the array. There are three ways to declare and initialize strings:

1. Declare and initialize

```
char str[] = "Hello";
char str[5 + 1] = "Hello";
```

2. Declare now and initialize later

```
char myString[4];
myString[0] = 'T';
myString[1] = 'h';
...
```

3. Declare a pointer to a constant string.

```
char *myStr = "WOW";
```

Just like in Python, strings are immutable, meaning you cannot change the characters or contents of the string.

10.2 Input/Output Strings

We can use `printf()` with the format specifier `%s` for strings. Alternatively, we may also use the function `put(str)`, where `str` is the string we wish to print.

When it comes to reading strings, `scanf()` has some limitations. First, it will stop scanning after the first whitespace or newline. It will also ignore any leading whitespace, which may be undesirable at times.

We can, however, use the `gets()` function, which will read any input until the newline character, and automatically adds the NULL character at the end of the character array.

10.3 String Functions

Length of a String

The `strlen()` function returns the number of non-null characters in a string as an integer.

Copying Strings

The `strcopy()` function copies one string into another.

```
char *strcpy(char *dest, char *source);
```

The `strncpy()` function copies the first n characters in the string into the destination string.

Concatenating Strings

The `strcat()` concatenates the source string onto the end of the destination string. Similarly with `strcpy()`, the `strncat()` function only concatenates the first n characters onto the end of the destination string.

Comparing Strings

```
int strcmp(const char *s1, const char *s2);
```

This function checks the string character by character and stops when at the null character, or when it comes across two characters that aren't the same. It returns an integer $s1 - s2$, and would return 0 if the strings are exactly the same. The `strncmp()` compares the first n characters of the string.

Searching for a Character

The `strchr()` function returns a pointer to the first occurrence of a character in a given string. The function assumes that the string is null-terminated.

```
char *strchr(const char *str, char c);
```

There is another function, `strstr()`, which returns a pointer to the first occurrence of `s2` in `s1`.

10.4 Array of Strings

There's two ways that we can make an array of strings, similar to how looked at 2D arrays.

2D Array of Characters

```
int main(void) {
    char months[][10] = {"Jan", "Feb", "Mar", ... , "Dec"};
}
```

1D Array of `char *`

Alternatively, we can also create a 1D array of pointers to other `char *`.

11 Recursion

Recursion is a bit of an odd chapter, so I won't divide it into its textbook sections. I'll just copy from my class notes instead (don't worry, they're essentially the same).

In recursion, we break a pattern into a smaller version that we solve repeatedly to get the final answer. In C, we observe this by having a function call itself repeatedly, on a smaller input each time.

NOTE: All recursive functions must have a base or terminating case, along with a recursive call. The base case is when the function returns a value without calling itself. An example of this is the first and second term of the Fibonacci Sequence:

```
    int fibonacci(int n) {
        if (n == 1) return 0;
        if (n == 2) return 1; // base cases for Fibonacci Sequence
        return fibonacci(n - 1) + fibonacci(n - 2); // recursive calls
    }
```

12 Data Structures

12.1 What are Data Structures?

A data structure is a way to store data of different types in a single variable name.

Defining Data Structures

Method 1: Define data structure and declare variable separately

```
    struct Neuron {
        int neuronNum;
        double input1, input2;
        char areaName[20];
    }

    int main(void) {
        struct Neuron neuron1;
        return 0;
    }
```

Method 2: Define and declare in the same statement

```
    struct Neuron {
        int neuronNum;
        double input1, input2;
        char areaName[20];
    } neuron;
```

We use dot notation to access members of a data structure when dealing with the variable directly. If we're using a pointer, we use arrow notation `->`.

We can alias a data structure using the `typedef` keyword.

```
    typedef char byte;

    int main(void) {
        byte oneLetter = 'S';
        byte sentence = 'Snefru';
        return 0;
    }
```

`typedef` allows us to redefine the name of the data structure, so that instead of typing `struct Neuron neuron1` we can just type `Neuron neuron1`.

12.2 Pointers to Data Structures

As mentioned above, if we're using a pointer to a data structure, we can use arrow notation to access elements of the structure.

If dynamically allocating space to a `Neuron` object, we can `malloc()` the same way:

```
Neuron *neutronPtr = (Neuron *)malloc(sizeof(Neuron));
```

13 Linked Lists

13.1 Why Linked Lists?

Sometimes we want to create lists of variable size, and be able to manipulate them very easily. Instead of using regular arrays, we can create special data structures which hold a value, and point to the next value in the list. We call these linked lists.

13.2 Form a Linked List

```
typedef struct node {
    int data;
    struct node *next;
} Node;
```

We anchor the linked list by setting the `node.next` element of the very last node to `NULL`. Since the NULL is an invalid address, we can use it as a signal for a function/loop to stop when working with linked lists.

13.3 Insert Nodes Into a Linked List

Create a Node in a Linked List

```
Node *createNode(int value) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (newNode != NULL) {
        newNode->data = value;
        newNode->next = NULL;
    }
    return newNode;
}

int main(void) {
    Node *head = (Node *)malloc(sizeof(Node));
    int value = 5;
    head = createNode(value);
    return 0;
}
```

Inserting a Node at the Beginning of the Linked List

```
void insertHead(Node *head, Node *newNode) {
    newNode -> head;
    return newNode;
}
```

Insert a Node at the End of the Linked List

```
Node *insertTail(Node *head, Node *newNode) {
    if (head == NULL) return newNode;
    Node *temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    return head;
}
```

Insert a Node Into an Ordered Linked List

```
Node *insertSorted(Node *head, Node *newNode) {
    Node *current = head, *prev = NULL;
    while ((current != NULL) && (newNode->data > current->data)) {
        prev = current;
        current = current->next;
    }
    if ((current != NULL) && (newNode->data == current->data)) {
        free(newNode); // to avoid duplicate value insertion
    } else {
        newNode->next = current;
        if (previous == NULL) head = newNode;
        else previous->next = newNode;
    }
    return head;
}
```

13.4 Delete Nodes in a Linked List

Deleting a Node at the Front of the List

```
void deleteFront(LinkedList *list) {
    if (list->head == NULL) {
        return;
    }
    Node *newHead = list->head->next;
    free(list->head);
    list->head=newHead;

    return;
}
```

Deleting a Node at the End of the Linked List

```
void deleteTail(LinkedLisst *list) {
    if (list->head == NULL) return;

    if (list->head->next == NULL) {
        deleteFront(list);
        return;
    }

    Node *current = list->head;
    while (current->next->next != NULL) {
        current = current->next;
    }
    free(current->next);
    current->next = NULL;
    return;
}
```

14 Sorting Algorithms

14.1 Insertion Sort

Pseudocode

1. Choose from the unsorted and keep a copy of it.
2. Find position to insert in sorted section.
3. Move elements up to free insert position.
4. Insert the copy of the chosen element.
5. Repeat until sorted.

The insertion sort algorithm has average time complexity of $O(n^2)$.

Implementation

```
void insertSort(int list[], int size) {
    for (int top = 1; top < size; top++) {
        int item = list[top];
        int i = top;
        while (i > 0 && item < list[i-1]) {
            list[i] = list [i - 1];
            i--;
        }
        list[i] = item;
    }
    return;
}
```

14.2 Selection Sort

Pseudocode

1. `topLoc = size - 1`
2. Find the largest in the unsorted list.
3. Swap with `topLoc`, then `topLoc--`.
4. Repeat steps 2 & 3 until the list is sorted.

Selection sort algorithm has complexity $O(n^2)$.

Implementation

```
    void selectSort(int list[], int size) {
        for (int top = size - 1; top > 0; top--) {
            int largeLoc = 0;
            for (int i = 1; i <= topLoc; i++) {
                if (list[i] > list[largeLoc]) largeLoc = i;
            }
            int temp = list[top];
            list[top] = list[largeLoc];
            list[largeLoc] = temp;
        }
        return;
    }
```

14.3 Bubble Sort

Pseudocode

1. For each pair of adjacent elements, if left ¿ right, then swap. Repeat `n - 1` times.

NOTE: If the lsit is already sorted, no need to continue checking and could stop. Use a `bool flag`.

The bubble sort algorithm has average time complexity of $O(n^2)$.

Implementation

```
    void bubbleSort(int list[], int size) {
        bool flag = false;
        for (int top = size - 1; top > 0 && !sorted; top--) {
            sorted = true;
            for (int i = 0; i < top; i++) {
                if (list[i] > list[i+1]) {
                    sorted = false;
                    int temp = list[i];
                    list[i] = list[top];
                    list[top] = temp;
                }
            }
        }
        return;
    }
```

14.4 Quick Sort (AKA Partition-Exchange Sort)

The Quick Sort algorithm was developed in 1960 by Tony Hoare. It has an average complexity of $O(n \cdot \log n)$. It's known as a "divide and compare" algorithm.

Pseudocode

1. Pick an element, which we'll call the `pivot`.
2. Partition:
   a) Move elements ¡ pivot to a list called `less` or `leftSide`.
   b) Move elements ¿ pivot to a list called `more` or `rightSide`.
3. Apply steps 1 & 2 recursively.

When selecting our pivot, we can select either the first or last element in the list, or the middle element. Each has its advantages.

Implementation

```
    void qSort(int A[], int leftSide, int rightSide) {
        if (leftSide >= rightSide) return;
        int left = leftSide, right = rightSide;
        int pivot = A[(left + right) / 2];
        while (left <= right) {
            while (A[left] < pivot) left++;
            while (A[right] > pivot) right--;
            if (left <= right) {
                swap(A, left, right);
                left++;
                right--;
            }
        }
        qSort(A, leftSide, right);
        qSort(A, left, rightSide);
        return;
    }

    void swap(int A[], int left; int right) {
        int temp = A[left];
        A[left] = A[right];
        A[right] = temp;

        return;
    }
```

15 Searching

15.1 Linear/Sequential Search

Linear search is named so because it searches for a given value in a list in a linear, sequential manner. It checks every element in a list until either the list is exhausted or the element is found.

The algorithm is as follows:

1. Start at `idx = 0`.
2. Check if `list[idx]` equals `value`.
3. If equal, then return `idx`, otherwise, `idx++`.
4. Repeat until `idx` equals `size` or the element is found.

```
int linearSearch (int list[], int size, int value) {
    for (int idx = 0; idx < size; idx++) {
        if (value = list[idx]) return idx;
    }
    return 1; // if not found raise error
}
```

15.2 Binary Search

Searching for a given element in a sorted array is much faster. Say you start by checking the middle value, and find that it is less than the value you're looking for, then you can ignore the part of the list to the left of the middle. Essentially, which each new value check, we can eliminate half of the array.

Implementation is given below:

```
treeNode *search(int data, treeNode *ptr) {
    if (ptr != NULL) {
        if (ptr->data = data) return data;
        if (ptr->data > data) return search(data, ptr->left);
        if (ptr->data < data) return search(data, ptr->right);
    }
    return NUlLL;
}
```

16 APS105 Summer 2024 Note Excerpts

At this point, we have covered the content discussed in the textbook. There were a lot of topics that were skipped, such as operations, variable scope, and so on; this is because this notes package relies on the reader having either elementary coding experience, or an interest deep enough to elicit searching said topics up on their own.

There are some extra topics, however, that were discussed in the Summer 2024 offering of this course, which I would like to append to this notes document here. The following topics are:

1. Binary Trees, and
2. Complexity.

16.1 Binary Trees

A binary tree is a type of data structure used in many applications. They are dynamic like linked lists, and have three elements: a cargo, and a left and right value, which are pointers to the next nodes.

Code implementation - type definition of a binary tree:

```
typedef struct bTree {
    int data;
    struct bTree *left, *right;
} treeNode;
```

Code implementation - creating a binary tree:

```
treeNode *createTreeNode(int data) {
    treeNode *newNode;
    newNode = (treeNode*)malloc(sizeof(treeNode));
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->data = data;
    return newNode;
}
```

Code implementation - inserting a node into a sorted tree:

```
treeNode *insertSortedTree(int data, treeNode *currNode) {
    if (currNode == NULL) {
        currNode = createTreeNode(data); // if the list is empty
    } else if (currNode->data > data) {
        currNode->left = insertSortedTree(data, currNode->left);
    } else {
        currNode->right = insertSortedTree(data, currNode->right);
    }
    return currNode;
}
```

Code implementation - deleting the entire tree:

```
void destroyTree(treeNode *leaf) {
    if (leaf!=NULL) {
        destroyTree(leaf->left);
        destroyTree(leaf->right);
        free(leaf);
    }
    return;
}
```

16.2 Complexity of Algorithms

We use complexity as a metric to determine how long it takes for a program to execute. This depends on many factors such as:

1. Hardware (machine capabilities),
2. Size of the input (we call this n),
3. Programming language used,
4. Whether or not recursion was employed, and
5. Type of the algorithm we're using.

We use the notation $O(x)$ to represent complexity, where x is replaced with an expression in terms of n that describes the complexity. Common complexities are $O(n)$, $O(n^2)$, and $O(n \cdot \log n)$.

16.3 Libraries Used in APS105

`<stdio.h>`	Provides access to print and input functions.
`<stdlib.h>`	Provides access to the random functions, as well as memory allocation functions.
`<stdbool.h>`	Provides access to Boolean variables.
`<string.h>`	Provides access to string functions.
`<math.h>`	Provides access to math functions.

Table 1: Libraries Used in APS105