ALX
SOFTWARE ENGINEERING

C Notebook

Student name: *Anrich Tait*

Course: *C Low Level Programming*
Due date: *March, 2024*

Purpose:

What is this book useful for?

1. What is C?
2. Why C?
3. What are the concepts in C and what can they be used for?

```
#include <stdio.h>
int main(void)
{
  int count;
  for (count = 1; count <= 500; count++)
     printf("I will not Throw paper airplanes in class.");
  return 0;
}
```

Question 2

How is C best utlised?

(*a*) Is it the best language?

(*b*) Or should I rather learn to fish?

Answer.

(*a*) Probably not.

(*b*) Very likley.

SubSection:.

Code Block:

Listing 1: Caption

```c
#include <stdio.h>

void greeting(void)
{
        printf("Hello, this is a template");
}

int main(void)
{
        greeting();
        return (0);
}
```

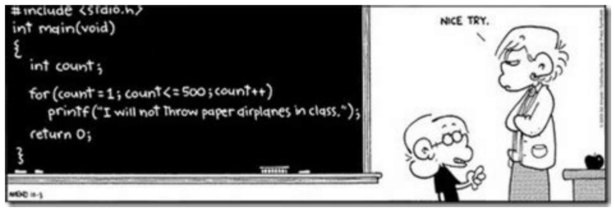
1. What will the output of this code block be?
2. Output = "Hello, this is a template"

The notes start here. Keep in mind they are in no particular order!

1. Functions

The following questions/topics will be addressed:

1. What are functions?
2. How are functions utilised?
3. Examples of using functions.



```
#include <stdio.h>
int main(void)
{
   int count;
   for(count=1; count<=500; count++)
      printf("I will not Throw paper airplanes in class.");
   return 0;
}
```

Functions

Functions break large computing tasks into smaller ones. This helps make code easier to change in the future and can also help with readability. Appropriate functions also hide details of operations from parts of the program that don't need them. C programs generally consist of many small functions rather than fewer large ones. The most basic example of a function is main(), every program has one and it is executed before any other functions in the program. Functions can be used for any task but generally it is best for each function to accomplish a specific task. A function declaration provides the actual body of the function.

The standard library provides numerous builtin functions that your program can call for example:

1. strcat() to concatenate strings
2. memcpy() to copy the memory location of one parameter to another.

A FUNCtional example:

Listing 2: The following example shows a very basic use of functions in C. label

```c
#include <stdio.h>

int main()
{
        int answer;

        printf("You are presented with two paths, "
                        "the road often travelled by "
                        "others, and the road less "
                        "travelled. Neither offers any"
                        "indication of what lays ahead, which"
                        "will you choose?\n");

        printf("1 = Road often travelled\n "
                        "2 = Road less travelled\n");
        scanf("%d", &answer);

        if (answer == 1)
        {
                roadTravelled();
        }
        else if (answer == 2)
        {
                roadLessTravelled();
        }
}

int roadTravelled()
{
        printf("You became a victim of a "
                        "roadside robbery and murder.\n");
        printf("GAME OVER");
        return (1);
}

int roadLessTravelled()
{
        printf("You got lost in the woods.\n "
                        "And died of starvation\n");
        printf("GAME OVER");
        return (2);
}
```

The above example gives the user two choices and then asks for input. based on the input a specified function is called to output the relevant text. While the use of functions is explicitly necessary in this example it shows how code can be broken up into smaller blocks, each with a specific purpose.

Defining a function

Syntax:

```
return_type function_name(parameter list)
{
        body of function
}
```

Here is a break down of each part of this syntax:

1. return_type: The type of value the function will return.
2. function_name: Name of the function.
3. (parameter list): list of parameters passed to the function.
4. body of function: code to be executed

Example:

```
int add(int a, int b)
{
        int result = a + b;
        return (result);
}
```

Break down:

1. int: Return type of function (integer).
2. add: Name of function.
3. (int a, int b) parameters.
4. int result ā + b: declare result variable and code that will be executed.
5. return (result) returns the value of the result variable.

Note: if a function does not return anything it can be specified as void.

Functions can be called from other parts of a program by using the function name and passing in the required parameters.

```
int x = 5;
int y = 7;

int z = add(x, y);
```

The function 'add' is called with x and y as the parameters. The x parameter will replace 'a' and 'y' will replace 'b'. The result is stored in 'z'.

Function types

1. Functions that return a value: Perform a specific task and return a value of a specific data type using the return statement. The previous "add" function is an example of this.

```
char get_first_char(char *str)
{
        return[0];
}
```

2. Functions that return nothing: Perform a task but does not return any value.

```
void print_hello()
{
        printf("Hello, world!")
}
```

3. Functions that take no parameters: Does not take any paramters but perfomrs a specific task.

```
void clear_screen()
{
        system("clear");
}
```

4. Functions that take parameters but return nothing.

```
void greet(char *name)
{
        printf("Hello, %s", name);
}
```

Calling a function

When a function is called, control of the program is transferred to the function until it's return statement or closing brace is reached. Then the main() function is used again.

To call a function you need to pass the required parameters.

Passing parameters to a function basically means providing values or variables to the function.

Going back to the 'add' function as an example.
x and y are the parameters that are passed to the function. When the function is called with arguments the values of the arguments are assigned to the corresponding parameters. So if we call the 'add' function with arguments 3 and 4 the values 3 and 4 are assigned to x and y respectively.

Function arguments

For a function to use arguments it must declare variables that accept the values of the arguments. These variables are called the formal parameters of a function.

Formal parameters behave like other local variables and are created upon entry to the function and are destroyed upon exit.

There are two ways arguments can be passed to a function:

Call/Pass by value:. The function receives a copy of the argument passed, not the original argument. So any changes made to the argument inside the function do not change the original value of the argument.

Listing 3: Call by value example

```c
#include <stdio.h>

void increment(int x)
{
        x++;
}

int main(void)
{
        int num = 10;

        increment(num);
        printf("%d", num);
        return (0);
}
```

In this example the 'increment' function takes an integer argument ('x') and then increments it by one. The following main function calls the increment function and passes "num" to it. This does not change the output as the increment function does not change the original value "num".

Call/Pass by reference:. Calling a function by reference involves passing arguments to a function using the parameters address. This way the actual value is changed, not just a copy of it.

To pass a variable by reference in C, you must declare the function parameter as a pointer type using the * operator.

The following example function takes a pointer to an integer as a parameter.

Listing 4: Call by reference example

```c
#include <stdio.h>

void increment(int *x)
{
        /*function takes a pointer to an integer and increments it by 1*
        (*x)++;
}

int main(void)
{
        int num = 5;

        /*call the increment function and pass a pointer to the memory
         * location of num*/
        increment(&num);

        /*output the new value of num*/
        printf("%d", num);

        return (0);
}
```

Note: by default C passes arguments by value.

Function Prototypes

What is a function prototype. A function prototype is a declaration of a function that describes the function's interface to the rest of the program. It specifies the function's name, return type and parameter types.

These are usually specified at the beginning of a file before any other functions. Syntax:

```
return_type function_name(parameter_list);
```

For example:

```
int max(int numOne, int numTwo);
```

This prototype declares a funtion named max that takes two integer parameters.

Why are function prototypes used?.

1. Type checking: By declaring the function prototype before calling the function the compiler can check that the arguments passed to the function match the expected types.
2. Avoid implicit declaration: If a function is not given an implicit type the compiler assumes it is an integer, this can lead to misleading errors relating to different return types or number of arguments.
3. Optimization: Allows the compiler to generate more efficient code by providing information about the function's interface. This can lead to smaller, faster code.
4. Documentation: Can server as documentation for the function, making it easier for other programmers to understand how to use a function.

Pointers

Syntax:. <type> *<name>

1. type = is the data type that the pointer will point to.
2. name = is the name of the pointer variable.

Example: To declare a pointer variable that will point to an integer value the following syntax is used,

```
int *ptr;
```

This declares a pointer variable named "ptr" that can point to an integer value.

To assign a value to a pointer variable the address-of operator (&) is used to get the memory address of the variable.

Example: Assign the address of an intger variable named "x" to the pointer variable "ptr",

```
int x = 10;
int *ptr = &x;
```

This assigns the address of "x" to "ptr", so now "ptr" points to x. To access the value that is stored at the memory location pointed to by a pointer variable the dereference operator is used (*).

Concept

Pointers are a fundamental concept in the C programming language, and understanding how they work is essential for writing efficient and effective C code. A pointer is a variable that stores the memory address of another variable. By using pointers, C programs can directly access and manipulate the values of other variables in memory, providing a powerful tool for building complex data structures and optimizing program performance.

At its core, a pointer is simply a memory address. When you declare a pointer variable in C, you are creating a variable that can store the address of another variable in memory. To declare a pointer variable, you use the * symbol before the variable name, like this: int *ptr;. This declares a pointer variable named ptr that can store the memory address of an integer variable.

Once you have declared a pointer variable, you can use the & operator to get the memory address of another variable, and assign that address to the pointer variable using the assignment operator =. For example, if you have an integer variable named x, you can get its memory address with &x, and assign it to the pointer variable ptr like this: ptr = &x;. Now, ptr points to the memory location of x, and you can use the pointer to access and manipulate the value of x directly.

One of the most common uses of pointers in C is to pass variables by reference to functions. In C, when you pass a variable to a function as an argument, the function gets a copy of the variable's value. However, by passing the memory address of the variable instead, you can allow the function to directly access and modify the variable's value. This can be useful for functions that need to modify variables outside of their own scope.

Another important use of pointers in C is for dynamic memory allocation. In C, you can use the malloc() function to allocate a block of memory of a specified size at runtime. The malloc() function returns a pointer to the beginning of the allocated memory block, which you can then use to access and manipulate the memory. Once you are done with the memory, you should free it using the free() function to avoid memory leaks.

Pointers can also be used to create complex data structures in C, such as linked lists, trees, and graphs. By using pointers to connect nodes together in these structures, you can create efficient and flexible data structures that can be easily traversed and manipulated.

However, while pointers are a powerful tool in C programming, they can also be dangerous if used improperly. Pointer errors, such as dereferencing a null pointer or accessing memory that has already been freed, can cause program crashes or even security vulnerabilities. To avoid these errors, it is important to carefully manage and validate pointers in your code, and to use best practices for pointer manipulation, such as initializing pointers to null and checking for null before dereferencing them.

In conclusion, pointers are a fundamental concept in C programming that allow you to directly access and manipulate the values of variables in memory. Pointers are used extensively in C for passing variables by reference, dynamic memory allocation, and creating complex data structures. While pointers can be a powerful tool, they can also be dangerous if used improperly, and it is important to carefully manage and validate pointers in your code to avoid errors and vulnerabilities.

Summary:. The syntax for a pointer consists of the data type followed by an asterisk and the name of the pointer variable. The address-of operator (&) is used to assign a value to the pointer, and the dereference operator (*) is used to access the value stored at the memory location pointed to by the pointer.

Examples

How to use pointers:.

Listing 5: Pass variables by reference to functions:

```c
#include <stdio.h>

void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    printf("a=%d, b=%d\n", a, b);  // Output: a=10, b=5
    return 0;
}
```

In this example, swap() is a function that takes two integer pointers as arguments and swaps their values. By passing the memory addresses of a and b to swap() using the & operator, we allow the function to directly modify the values of a and b.

Listing 6: Allocate memory dynamically: language

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

int main() {
    int *arr = malloc(sizeof(int) * 10);
    if (arr != NULL) {
        for (int i = 0; i < 10; i++) {
            arr[i] = i;
        }
    }
    free(arr);
    return 0;
}
```

In this example, we use the malloc() function to dynamically allocate an array of 10 integers, and store its memory address in the pointer variable arr. We then use a loop to assign the values 0 to 9 to the array elements. Finally, we use the free() function to release the allocated memory.

Listing 7: Access out-of-bounds memory: language

```c
#include <stdio.h>

int main() {
    int *ptr = NULL;
    *ptr = 5;  // Invalid: dereferencing a null pointer
    return 0;
}
```

In this example, we declare a pointer variable ptr and assign it the null value using the NULL macro. However, we then attempt to assign the value 5 to the memory location pointed to by ptr, which is an invalid operation because ptr is null and does not point to a valid memory location.

Listing 8: Access out-of-bounds memory: language

```c
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = &arr[0];
    for (int i = 0; i < 10; i++) {
        printf("%d\n", *ptr);  // Invalid: accessing out-of-bounds memory
        ptr++;
    }
    return 0;
}
```

In this example, we declare an integer array arr with 5 elements, and create a pointer variable ptr that points to the first element of the array. However, we then use a loop to increment the pointer variable ptr 10 times, and attempt to print the value pointed to by ptr at each iteration. This is an invalid operation because ptr eventually points to memory outside the bounds of the arr array, which can cause undefined behavior or program crashes.

Arrays:

Arrays consist of contigous memory locatios. The lowest address corresponds to first element and the highest address to the last element.

Syntax:.
```
type arrayName[arraySize] = size;
```

Initializing arrays:. Below is an example of initializing an array called balance with 5 indexes.

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Note the number of indexes inside the { } cannot be bigger than the specified size inside the [ ]. So in the above example the indexes cannot number more than 5.

An array can also be initialized by excluding the size in the [ ]. For example:

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

This creates an array just big enough to store indexes.

You can also asign values to indexes individually:

```
double balance[4] = 50.0;
```

This will assign the 5th index a value of "50.0".

In the below table the array is broken down into the index number and their corresponding value.

Index:	0	1	2	3	4
Value:	1000.0	2.0	3.4	7.0	50.0

Accessing Array Indexes:. An index/element is accessed refrencing the array name. This is done by placing the index of the element after the array name.

```
double salary = balance[9]
```

The above statement assigns the value of the 10th index in the balance array to the salary variable.

Listing 9: This code block shows examples of how to use the above mentioned concepts.

```c
#include <stdio.h>

int main () {

   int n[10]; /* n is an array of 10 integers */
   int i,j;

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = i + 100;
          /* set element at location i to i + 100 */
   }

   /* output each array element's value */
   for (j = 0; j < 10; j++ ) {
      printf("Element[%d] = %d\n", j, n[j] );
   }
   return 0;
}
```

Output:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

The following are essential for understanding arrays in C:

1. C supports multidimensional arrays. The simplest version of the multidimensional array is the two dimensional array. Example:

   ```c
   char arrayName[] = "arrayText";
   ```

2. It is possible to pass to the function a pointer to an array by specifying the array's name without an index.

3. C allows a function to return an array.

4. You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

Strings

In C, strings are declared using a one dimensional array, terminated by a null character ′\0′. Therefore a ′null-terminated′ string contains the string indexes followed by a null.

The following example shows how a string is declared, note that the array size is increased to account for the ′\0′ character.

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\textbackslash0'};
```

The same array can be declared by doing the following:

```
char greeting[] = "Hello";
```

As you can see declaring the null character is not neccessary. The compiler does this for you. When the compiler reaches a null character it will initialize the array.

Listing 10: The below code shows the two different ways to declare a string (using an array) as you will see the second is more efficient.

```
#include <stdio.h>

int main(void)
{
        char greetingOne[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
        char greetingTwo[] = "Hello";

        printf("%s\n", greetingOne); /* Uneccessary declaration */
        printf("%s\n", greetingTwo); /* Better declaration */

        return (0);
}
```

Output:

```
Hello
Hello
```

C has various functions that allow you to manipulate strings, here are a few of the most common:

Function:	Use:
strcpy(s1, s2)	Copies string s2 into string s1
strcat(s1, s2)	Concatenates string s2 onto the end of string s1
strlen(s1)	Returns the length of string s1
strcmp(s1, s2)	Returns 0 if s1 and s2 are the same; less than 0 if s1 < s2; greater than if s1 > s2.
strchr(s1, ch)	Returns a pointer to the first occurrence of character ch in string s1.
strstr(s1, s2)	Returns a pointer to the first occurrence of string s2 in string s1

Listing 11: The following code shows examples of how to use some of the above mentioned functions. language

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

int main(void)
{
        char str1[] = "Hello ";
        char str2[] = "world";
        char str3[100];

        /*Copy str1 into str3*/
        strcpy(str3, str1);
        printf("strcpy: %s\n", str3);

        /*Concatenate str2 onto end of str1*/
        strcat(str1, str2);
        printf("strcat: %s\n", str1);

        /*Return the length of str1*/
        int len = strlen(str1);
        printf("strlen: %d\n", len);

        return (0);
}
```

Output:

```
strcpy: Hello
strcat: Hello world
strlen: 11
```

2. Structures and typedef

2.1. Resources.

1. struct (C programming language) (wiki)

2.1.1. struct. A struct is a composite data type[1] declaration that defines a physically grouped list of variables under one name in a block of memory. This allows different variables to be accessed via a single pointer or by the struct declared name which returns the same address.

A struct directly references a contigious block[2] of physical memory, usually sized by word-length boundaries. Being a block of contigious memory each field within a struct is located at a certain fixed offset from the start.

Thus the 'sizeof' operator must be used to get the number of bytes needed to store a particular type of struct, just as it can be used for primitives[3].

The syntax for a struct is as follows:

```
struct tag_name
{
        type memberOne;
        type memberTwo;
};
```

Similar syntax is used in the context of a 'typedef' declaration of a type alias or the declaration or definition of a variable:

```
typedef struct tag_name
{
        type memberOne;
        type memberTwo;
} struct_alias;
```

There are 3 ways to initialize a structure:

1. Declare the struct with integer members.

```
struct point
{
        int x;
        int y;
};
```

2. Define a variable 'p' of type point and initialize it's first two members in place

[1]A data type that combines several values or variables into a single entity, examples include arrays and structures

[2]A block of memory where all bytes are stored consecutively, with no gaps between them.

[3]A primitive is any basic data type

```
struct point p = { 1, 2 };
```

3. For non contiguous or out of order members list, designated initializer style mayb be used

```
struct point p = { .y = 2, .x = 1 };
```

4. If an initializer is given or if the pbject is statically allocated, omitted elements are initiliazed to 0. A third way of initializing a structure is to copy the value of an existing object of the same type

```
struct point q = p;
```

A struct may be assigned to another struct. For example a compiler might use memcpy() to perform such an assignment.

```
struct point
{
        int x;
        int y;
};

int main(void)
{
        struct point p = {1, 3};
        struct point q;
        q = p;
        return (0);
}
```

Pointers can also be used to refer to a struct by it's address. This is useful for passing structs to a function. The pointer can be dereferemced using the * operator. The -> operator dereferences the pointe to struct (left operand) and then accesses the value of a member of the struct (right operand).

```
struct point
{
        int x;
        int y;
};

struct point my_point = {3, 7};
struct point *p = &my_point; //p is a pointer to my_point
(*p).x = 8; //set the first member of the struct
p->x = 8; //equivalent method to set the first
                        //member of the struct
```