Chp28.

28. VARIABLE ARGUMENTS

Sometimes, you may come across a situation, when you want to have a function, which can take variable number of arguments, i.e., parameters, instead of predefined number of parameters. The C programming language provides a solution for this situation and you are allowed to define a function which can accept variable number of parameters based on your requirement. The following example shows the definition of such a function.

```
int func(int, ... )
{
.
.
.
}
int main()
{
func(1, 2, 3);
func(1, 2, 3, 4);
}
```

It should be noted that the function **func()** has its last argument as ellipses, i.e., three dotes (**...**) and the one just before the ellipses is always an **int** which will represent the total number variable arguments passed. To use such functionality, you need to make use of **stdarg.h** header file which provides the functions and macros to implement the functionality of variable arguments and follow the given steps:

1. 1. Define a function with its last parameter as ellipses and the one just before the ellipses is always an **int** which will represent the number of arguments.
2. 2. Create a **va_list** type variable in the function definition. This type is defined in stdarg.h header file.
3. 3. Use **int** parameter and **va_start** macro to initialize the **va_list** variable to an argument list. The macro va_start is defined in stdarg.h header file.
4. 4. Use **va_arg** macro and **va_list** variable to access each item in argument list.

1. 5. Use a macro va_end to clean up the memory assigned to va_list variable.

Now let us follow the above steps and write down a simple function which can take the variable number of parameters and return their average:

```
#include <stdio.h>
#include <stdarg.h>
double average(int num,...)
{
va_list valist;
double sum = 0.0;
int i;
/* initialize valist for num number of arguments */
va_start(valist, num);
/* access all the arguments assigned to valist */
for (i = 0; i < num; i++)
{
sum += va_arg(valist, int);
}
/* clean memory reserved for valist */
va_end(valist);
return sum/num;
}
int main()
{
printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

When the above code is compiled and executed, it produces the following result. It should be noted that the function average() has been called twice and each time the first argument represents the total number of variable arguments being passed. Only ellipses will be used to pass variable number of arguments.

```
Average of 2, 3, 4, 5 = 3.500000
Average of 5, 10, 15 = 10.000000
```

Chp29.

This chapter explains dynamic memory management in C. The C programming language provides several functions for memory allocation and management. These functions can be found in the **<stdlib.h>** header file.

S.N. Function and Description 1 **void *calloc(int num, int size);** This function allocates an array of **num** elements each of which size in bytes will

be **size**. 2 **void free(void *address);** This function releases a block of memory block specified by address. 3 **void *malloc(int num);** This function allocates an array of **num** bytes and leave them initialized. 4 **void *realloc(void *address, int newsize);** This function re-allocates memory extending it upto **newsize**. **Allocating Memory Dynamically** While programming, if you are aware of the size of an array, then it is easy and you can define it as an array. For example, to store a name of any person, it can go up to a maximum of 100 characters, so you can define something as follows: `char name[100];` But now let us consider a situation where you have no idea about the length of the text you need to store, for example, you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later, based on requirement, we can allocate memory as shown in the below

example: `#include <stdio.h>`


```
#include <stdlib.h> #include <string.h> int main() { char name[100]; char *description; strcpy(name, "Zara Ali"); /* allocate memory dynamically */ description = malloc( 200 * sizeof(char) ); if( description == NULL ) { fprintf(stderr, "Error - unable to allocate required memory\n"); } else { strcpy( description, "Zara ali a DPS student in class 10th"); } printf("Name = %s\n", name ); printf("Description: %s\n", description ); }
```

When the above code is compiled and executed, it produces the following result.

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```

Same program can be written using **calloc();** only thing is you need to replace malloc with calloc as follows: `calloc(200, sizeof(char));` So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size is defined, you cannot change it.

Resizing and Releasing Memory When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function **free()**. Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**. Let us check the above program once again and make use of realloc() and free() functions:

```
#include <stdio.h> #include <stdlib.h> #include <string.h> int main() { char
```

```
name[100]; char *description; strcpy(name, "Zara Ali"); /* allocate
memory dynamically */ description = malloc( 30 * sizeof(char) );
if( description == NULL ) { fprintf(stderr, "Error - unable to
allocate required memory\n"); } else { strcpy( description, "Zara ali
a DPS student."); } /* suppose you want to store bigger description */
description = realloc( description, 100 * sizeof(char) );
if( description == NULL ) { fprintf(stderr, "Error - unable to
allocate required memory\n"); }
```



```
else { strcat( description, "She is in class 10th"); } printf("Name =
%s\n", name ); printf("Description: %s\n", description ); /* release
memory using free() function */ free(description); }
```

When the above code is compiled and executed, it produces the following result. `Name = Zara Ali Description: Zara ali a DPS student.She is in class 10th` You can try the above example without re-allocating extra memory, and strcat() function will give an error due to lack of available memory in description.

Chp9

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially

when you want to control your program from outside instead of hard coding those values inside the code. The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly: `#include <stdio.h> int main( int argc, char *argv[] ) { if( argc == 2 ) { printf("The argument supplied is %s\n", argv[1]); } else if( argc > 2 ) { printf("Too many arguments supplied.\n"); } else { printf("One argument expected.\n"); } }` When the above code is compiled and executed with a single argument, it produces the following result. `$./a.out testing The argument supplied is testing`

When the above code is compiled and executed with two arguments, it produces the following result. `$./a.out testing1 testing2 Too many arguments supplied.` When the above code is compiled and executed without passing any argument, it produces the following result. `$./a.out One argument expected` It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument, then **argc** is set at 2. You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes: `#include <stdio.h> int main( int argc, char *argv[] ) { printf("Program name %s\n", argv[0]); if( argc == 2 ) { printf("The argument supplied is %s\n", argv[1]); } else if( argc > 2 ) { printf("Too many arguments supplied.\n"); } else { printf("One argument expected.\n"); }`

`}` When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result. `$./a.out "testing1 testing2" Programm name ./a.out The argument supplied is testing1 testing2`