Object Oriented Programming using C++

Functions

What is a function?

A function is a self-contained block of code designed to perform a specific task. It can also be described as a set of instructions to accomplish a particular goal.

The general format of a function is:

```
return-type function-name(parameter-list) {
   // function body
}
```

Body of the function

- `ret-type` specifies the type of data the function returns. It can return any type of data except an array. By default, the return type is assumed to be `int`.

- Function name is the identifier for the function and must follow the rules for naming variables or identifiers.

- Parameter list is defined within parentheses and takes the form:

type varname1, type varname2, ..., type varnameN

The parameters should be separated by commas and enclosed within parentheses.

Example:

f(int i, int k, int j) // correct

f(int i, k, float j) // correct

f(int i, int k, float) // incorrect

Scope rules of Functions

- Scope rules in a language dictate whether a specific piece of code can access or recognize another piece of code or data.

- Local variables are those defined within a function.

- A local variable is created when the function is entered and is destroyed when the function exits.

- An exception to this rule occurs when a variable is declared with the `static` storage class specifier, which affects its lifetime and scope.

Function Arguments

- Formal parameters are variables declared in a function to accept values passed to the function.

- For example, in the function:

```
int is_in(char *s, char c) {
    while (*s) {
        if (*s == c)
            return 1;
        else
            return 0;
    }
}
```

The function `is_in` has two parameters: `s` and `c`. This function checks if the character `c` is present in the string `s`. It returns `1` if `c` is found; otherwise, it returns `0`.

Call by Value, Call by Reference

In a computer language, there are two ways that arguments can be passed to a subroutine i,e by

1. Call by Value.
2. Call by Reference.

Call by Value

- In this method, a copy of the actual value of the argument is passed to the subroutine. Changes made to the parameter within the subroutine do not affect the original argument.

Example:

```
void increment(int x) {
    x = x + 1;
}
int main() {
    int a = 5;
    increment(a);
    // 'a' remains 5; only a copy was modified
}
```

Call by Reference

- In this method, a reference (or address) to the actual argument is passed to the subroutine. This allows the subroutine to modify the original argument directly.

Example:

```
void increment(int &x) {
    x = x + 1;
}
int main() {
    int a = 5;
    increment(a);
    // 'a' is now 6; the original value was modified
}
```

Calling functions with Arrays

- Typically, constants and variables are passed to functions as arguments.

- When an entire array needs to be passed to a function, the reference (address) of the first element of the array is passed instead of copying the entire array. This deviates from the usual call-by-value approach.

- Any modifications made to the array within the function will affect the actual array in the calling function.

```c
#include <stdio.h>
int array_sum(int a[], int n) {
    int i, sum = 0;
    printf("Array Elements are:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]); // Read each element of the array
        sum += a[i];        // Add each element to the sum
    }
    return sum;
}
int main() {
    int n, i;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n]; // Declare an array of size n
    int result = array_sum(arr, n);
    printf("Sum of array elements: %d\n", result);
    return 0;
}
```

The Return Statement

The `return` statement in C++ serves two main purposes:

1. Immediate Exit from the Function:

It causes the function to terminate immediately, transferring control back to the calling function. This is useful for stopping further execution in a function based on a condition.

```
void exampleFunction(int value) {
    if (value < 0) {
        return; // Exit the function if value is negative
    }
    // Rest of the function code
}
```

2. Returning a Value:

It allows a function to return a value to the caller. This value can be of any type specified by the function's return type.

```
int add(int a, int b) {
    return a + b; // Return the sum of a and b
}
int main() {
    int result = add(5, 3); // result will be 8
    return 0;
}
```

In summary, the `return` statement is crucial for controlling the flow of execution in functions and for providing results back to the caller.

Returning from a Function

The termination of a function in C++ can occur in two primary ways:

1. Normal Termination: This happens when the function completes execution of all its statements and reaches the closing curly brace (`}`) of its block. At this point, the function automatically returns to the caller. If the function has a return type other than `void`, it must return a value of the specified type before the function terminates. If the function has a `void` return type, it terminates without returning a value.

```
int add(int a, int b) {
    int sum = a + b; // Calculate sum
    return sum;      // Return sum to caller
}
int main() {
    int result = add(5, 3); // result will be 8
    return 0;
}
```

2. Explicit Termination with the `return` Statement: This occurs when a `return` statement is executed within the function. This statement immediately transfers control back to the caller, optionally returning a value. The function's execution stops at the `return` statement, and the remaining code (if any) is not executed.

```
int divide(int a, int b) {
    if (b == 0) {
        return 0; // Return immediately if divisor is zero
    }
    return a / b; // Return the result of division
}
int main() {
    int result = divide(10, 2); // result will be 5
    return 0;
}
```

In summary, a function can terminate either by reaching its end naturally or by executing a `return` statement.

Example: Reversing a String with a Function

```
#include <stdio.h>
#include <string.h>
// Function prototype
void pr_reverse(char *s);
int main(void) {
    pr_reverse("I like C++"); // Call function to reverse and print string
    return 0;
}
```

```
// Function definition
void pr_reverse(char *s) {
    int len = strlen(s); // Get the length of the string
    for (int t = len - 1; t >= 0; t--) { // Iterate backwards through the string
        putchar(s[t]); // Print each character
    }
}
```

Returning Values

Function Return Values

- Void Functions: A function declared with the `void` return type does not return any value. It performs its task and then terminates without sending any data back to the caller.

- Non-Void Functions: Functions that return a type other than `void` must explicitly return a value of that type. If such a function does not use a `return` statement to provide a value, it will return a garbage value. This is because the function is expected to return a specific type of value, and failing to do so can result in undefined behavior.

Valid and Invalid Expressions

- Valid Expressions:

`printf("greater");` : This statement correctly uses the `printf` function to output the string "greater" to the standard output.

`isdigit(ch);` : This expression correctly uses the `isdigit` function to check if `ch` is a digit.

- Invalid Expression:

`swap(x, y) = 100;` : This is incorrect because `swap(x, y)` is a function call, and you cannot assign a value to a function call. Functions are not assignable; instead, they are called to perform operations or return values.

Types of Function

1. Computational Functions:

- These functions perform calculations or operations based on their arguments and return a result. They are often referred to as "pure" functions because they do not have side effects; their output depends solely on their input.

- Example: `sqrt()` computes the square root, and `sin()` computes the sine of an angle.

2. Information-Manipulating Functions:

- These functions perform operations that manipulate data or resources and return a value indicating the success or failure of the operation.

- Example: `fclose()` is used to close a file. If the operation is successful, `fclose()` returns `0`; if there is an error, it returns `EOF`.

3. Procedural Functions:

- These functions perform a series of actions but do not return a value. They are strictly procedural and are often declared with the `void` return type.

- Example: `exit()` terminates the program and does not return a value.

Example Program

```
#include <stdio.h>

int mul(int a, int b); // Function prototype

int main(void) {

    int x, y, z;

    x = 10;

    y = 20;

    z = mul(x, y); // The return value of mul() is assigned to z

    printf("%d\n", mul(x, y)); // The return value is used by printf() directly

    mul(x, y); // The return value is discarded as it's neither assigned nor used

    return 0;

}

int mul(int a, int b) {

    return a * b; // Returns the product of a and b

}
```

- In the `main` function:
  `z = mul(x, y);` assigns the result of `mul(x, y)` to `z`.
  `printf("%d\n", mul(x, y));` uses the result of `mul(x, y)` directly for output.
  `mul(x, y);` calls the function without using or storing its result.

Returning Pointers

```
#include <stdio.h>
char *match(char c, char *s); // Function prototype
int main(void) {
   char s[80], *p, ch;
   // Input string from user
   printf("Enter a string: ");
   fgets(s, sizeof(s), stdin);
   // Input character to find
   printf("Enter a character to find: ");
   ch = getchar();
   // Find character in string
   p = match(ch, s);
   // Check if character was found
   if (*p) {
      printf("Match found: %s\n", p); // Print substring starting from the match
   } else {
      printf("No match found\n");
   }
   return 0;
}
char *match(char c, char *s) {
   while (c != *s && *s)
      s++;
   return s; // Return pointer to the match or to the null terminator
}
```

Functions of Type void

```
#include <stdio.h>
// Function declaration
void print_vertical(char *str);
int main(void) {
    char str[] = "Hello, World!";
    // Call the function to print the string vertically
    print_vertical(str);
    return 0;
}
```

```
// Function definition
void print_vertical(char *str) {
    while (*str) {        // Loop until the null terminator is reached
        printf("%c\n", *str); // Print the current character followed by a newline
        str++;            // Move to the next character in the string
    }
}
```

What does main() Return?

```
#include <stdio.h>
#include <stdlib.h> // For the exit function
int main(void) {
    int status = 0; // 0 typically indicates successful execution
    printf("Program is running...\n");
    // Simulate some processing
    if (/* some error condition */) {
        status = 1; // Non-zero value indicates an error
    }
    // Return the status code to the operating system
    return status; // This is equivalent to calling exit(status)
}
```