Author: Lokang Jackson

Book Name: C and MySQL

Variable Names

In the C programming language, a variable is a storage location for a value that can be modified during the execution of a program. Variables are used to store data values and perform operations on them.

To declare a variable in C, you need to specify the data type of the variable and its name. For example:

```
int age;
float price;
char grade;
```

Here, **age** is an integer variable, **price** is a floating-point variable, and **grade** is a character variable.

You can also initialize a variable when you declare it by assigning a value to it. For example:

```
int age = 20;
float price = 9.99;
char grade = 'A';
```

After declaring and initializing a variable, you can use it in your program by referencing its name. For example:

```
age = age + 1;
price = price * 0.8;
```

These statements increase the value of **age** by 1 and multiplies the value of **price** by 0.8.

It's important to note that variables in C must be declared before they are used, and they can only hold values of the data type they were declared with.

Data Types

In programming, a data type is a classification that specifies the type of value a variable can hold. Data types are an important concept in programming languages because they determine how a programmer can use and manipulate the values stored in a variable.

In the C programming language, there are several built-in data types, including:

- **int**: Integer data type for storing whole numbers.
- **float**: Floating-point data type for storing decimal numbers.
- **double**: Double-precision floating-point data type for storing larger decimal numbers.
- **char**: Character data type for storing a single character.
- **void**: Special data type used to represent the absence of a value or type.

There are also derived data types in C, which are created by combining the built-in data types in different ways. For example:

- **array**: A collection of variables of the same data type, indexed by a contiguous set of integers.
- **struct**: A collection of variables of different data types, grouped together under a single name.
- **union**: A special data type that allows a single memory location to be interpreted in different ways.

It's important to choose the appropriate data type for your variables based on the values they will hold and the operations you will perform on them. Using the correct data type can help ensure the accuracy and efficiency of your program.

Here is an example of using variables of different data types in a C program:

```
#include <stdio.h>
int main()
{
   int age = 30;
   float height = 1.75;
   double salary = 45000.00;
   char gender = 'M';
   printf("Age: %d\n", age);
   printf("Height: %.2f\n", height);
   printf("Salary: %.2f\n", salary);
   printf("Gender: %c\n", gender);
   return 0;
}
```

In this example, we have declared four variables: **age**, **height**, **salary**, and **gender**. **age** and **gender** are of type **int** and **char**, respectively, while **height** and **salary** are of type **float** and **double**, respectively.

We have also initialized the variables with values when we declared them.

Then, we use the **printf** function to print the values of the variables to the console. The **printf** function takes a format string as its first argument, followed by the values to be printed. The format string specifies how the values should be formatted, using special formatting codes such as **%d** for integers, **%f** for floating-point numbers, and **%c** for characters.

If you compile and run this program, it will print the following output:

```
Age: 30
Height: 1.75
Salary: 45000.00
Gender: M
```

Operators

Operators are special symbols in programming languages that perform specific operations on one or more operands (values or variables). In the C programming language, there are several types of operators, including:

- Arithmetic operators: Perform basic arithmetic operations such as addition, subtraction, multiplication, and division. For example:

```
int a = 10;
int b = 5;
int c = a + b;  // c is 15
int d = a - b;  // d is 5
int e = a * b;  // e is 50
int f = a / b;  // f is 2
```

- Assignment operators: Assign a value to a variable. For example:

```
int a = 10;
int b = 5;
a = b;  // a is now 5
```

- Comparison operators: Compare two values and return a Boolean value indicating whether the comparison is true or false. For example:

```
int a = 10;
int b = 5;
if (a > b) {
   printf("a is greater than b\n");
}
if (a == b) {
   printf("a is equal to b\n");
}
if (a < b) {
   printf("a is less than b\n");
}
```

- Logical operators: Perform logical operations such as AND, OR, and NOT. For example:

```
int a = 1;
int b = 0;
if (a && b) {
   printf("Both a and b are true\n");
} else {
   printf("Either a or b is false\n");
}
if (a || b) {
   printf("Either a or b is true\n");
} else {
   printf("Both a and b are false\n");
}
if (!a) {
   printf("a is false\n");
} else {
   printf("a is true\n");
}
```

- Increment and decrement operators: Increase or decrease the value of a variable by 1. For example:

```
int a = 10;
a++;  // a is now 11
a--;  // a is now 10
```

There are many other types of operators in C, each serving a specific purpose. It's important to understand how these operators work and how to use them effectively in your programs.

if

The **if** statement in C is used to execute a block of code conditionally, based on the value of a Boolean expression. The Boolean expression is evaluated, and if it is true, the code block is executed. If the expression is false, the code block is skipped.

Here is an example of using an **if** statement in a C program:

```
#include <stdio.h>
int main()
{
   int a = 10;
   int b = 5;
   if (a > b) {
       printf("a is greater than b\n");
   }
   return 0;
}
```

In this example, we have two variables, **a** and **b**, and we are using an **if** statement to check if **a** is greater than **b**. If it is, the message "a is greater than b" is printed to the console. If **a** is not greater than **b**, the code block is skipped and nothing is printed.

You can also use an **else** clause with an **if** statement to specify a block of code to be executed if the Boolean expression is false. For example:

```
#include <stdio.h>
int main()
{
   int a = 10;
   int b = 5;
   if (a > b) {
       printf("a is greater than b\n");
```

```
    } else {
        printf("a is not greater than b\n");
    }
    return 0;
}
```

In this example, if **a** is greater than **b**, the message "a is greater than b" is printed. If **a** is not greater than **b**, the message "a is not greater than b" is printed.

You can also use multiple **if** statements with **else if** clauses to create more complex conditional logic. For example:

```
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 5;
    if (a > b) {
        printf("a is greater than b\n");
    } else if (a == b) {
        printf("a is equal to b\n");
    } else {
        printf("a is less than b\n");
    }
    return 0;
}
```

In this example, if **a** is greater than **b**, the message "a is greater than b" is printed. If **a** is equal to **b**, the message "a is equal to b" is printed. If neither of these conditions are true, the message "a is less than b" is printed.

Array

An array in C is a collection of variables of the same data type that are stored in contiguous memory locations and can be accessed using a single identifier. Arrays are useful for storing and manipulating large amounts of data, as they allow you to access and manipulate multiple values using a single loop or function.

Here is an example of declaring and initializing an array in C:

```
#include <stdio.h>
int main()
{
    int a[5] = {1, 2, 3, 4, 5};
    return 0;
}
```

In this example, we have declared an array **a** of 5 integers and initialized it with the values 1, 2, 3, 4, and 5. The array has a fixed size of 5, and the values are stored in contiguous memory locations.

You can access individual elements of the array using the index operator **[]**. The index of an array element starts at 0 and goes up to the size of the array minus 1. For example:

```
#include <stdio.h>
int main()
{
   int a[5] = {1, 2, 3, 4, 5};
   printf("a[0] = %d\n", a[0]);  // prints 1
   printf("a[1] = %d\n", a[1]);  // prints 2
   printf("a[2] = %d\n", a[2]);  // prints 3
   printf("a[3] = %d\n", a[3]);  // prints 4
   printf("a[4] = %d\n", a[4]);  // prints 5
   return 0;
}
```

In this example, we have accessed each element of the array **a** and printed its value to the console.

You can also use a loop to iterate over the elements of an array. For example:

```
#include <stdio.h>
int main()
{
   int a[5] = {1, 2, 3, 4, 5};
   for (int i = 0; i < 5; i++) {
       printf("a[%d] = %d\n", i, a[i]);
   }
   return 0;
}
```

String

A string in C is a sequence of characters stored in an array and terminated with a null character ('\0'). Strings are used to represent text data in C, and they are often used for input and output operations.

In C, strings are represented as arrays of characters, and they are manipulated using string-handling functions provided in the standard library.

Here is an example of declaring and initializing a string in C:

```
#include <stdio.h>
int main()
{
   char str[10] = "hello";
   return 0;
}
```

In this example, we have declared a string **str** as an array of characters with a size of 10. The string is initialized with the value "hello", which is stored in the array along with a null character at the end to mark the end of the string.

You can access individual characters of a string using the index operator **[]**. The index of a character in a string starts at 0 and goes up to the length of the string minus 1. For example:

```
#include <stdio.h>
int main()
{
   char str[10] = "hello";
   printf("str[0] = %c\n", str[0]);  // prints 'h'
   printf("str[1] = %c\n", str[1]);  // prints 'e'
   printf("str[2] = %c\n", str[2]);  // prints 'l'
   printf("str[3] = %c\n", str[3]);  // prints 'l'
   printf("str[4] = %c\n", str[4]);  // prints 'o'
   return 0;
}
```

In this example, we have accessed each character of the string **str** and printed its value to the console.

You can also use a loop to iterate over the characters of a string. For example:

```
#include <stdio.h>
int main()
{
   char str[10] = "hello";
   for (int i = 0; str[i] != '\0'; i++) {
       printf("str[%d] = %c\n", i, str[i]);
   }
   return 0;
}
```

In this example, we have used a **for** loop to iterate over the characters of the string **str**. The loop continues to iterate as long as the current character is not the null character, which indicates the end of the string.

User Input

To read user input in C, you can use the **scanf** function provided in the standard library. The **scanf** function allows you to read data from the standard input (usually the keyboard) and store it in variables.

Here is an example of using **scanf** to read an integer from the user in a C program:

```
#include <stdio.h>
int main()
{
   int a;
   printf("Enter an integer: ");
   scanf("%d", &a);
   printf("You entered: %d\n", a);
   return 0;
}
```

In this example, we have declared a variable **a** to store the integer input from the user. The **printf** function is used to prompt the user to enter an integer, and the **scanf** function is used to read the integer from the standard input. The **%d** specifier in the **scanf** format string indicates that an integer is expected. The address of the variable **a** is passed as the argument to **scanf**, so that the value read from the input can be stored in the variable.

After the **scanf** function is executed, the value of **a** will be the integer entered by the user. The **printf** function is then used to print the value of **a** to the console.

You can use similar syntax to read other data types from the user, such as floating-point numbers, characters, and strings. For example:

```
#include <stdio.h>
int main()
{
   float b;
   char c;
   char str[20];
   printf("Enter a floating-point number: ");
   scanf("%f", &b);
   printf("You entered: %f\n", b);
   printf("Enter a character: ");
   scanf(" %c", &c);
   printf("You entered: %c\n", c);
   printf("Enter a string: ");
   scanf("%s", str);
   printf("You entered: %s\n", str);
   return 0;
}
```

In this example, we have read a floating-point number, a character, and a string from the user using the **scanf** function.

Pointers

A pointer in C is a variable that stores the memory address of another variable. Pointers are used to access and manipulate the data stored at a specific memory location.

Pointers are declared using the * operator, which indicates that the variable is a pointer. For example:

```
int *p;  // p is a pointer to an integer
float *q;  // q is a pointer to a float
char *r;  // r is a pointer to a char
```

To assign a memory address to a pointer, you can use the **&** operator to obtain the memory address of a variable. For example:

```
#include <stdio.h>
int main()
{
   int a = 5;
   int *p;
   p = &a;  // assign the memory address of a to p
   printf("The memory address of a is: %p\n", &a);
   printf("The value stored at the memory address of a is: %d\n", *p);
   return 0;
}
```

In this example, we have declared an integer variable **a** with the value 5 and a pointer variable **p** that can store a memory address. The **&** operator is used to obtain the memory address of **a**, which is then assigned to **p**.

The **printf** function is used to print the memory address of **a** to the console, and the value stored at that memory address is printed using the ***** operator, which is used to dereference a pointer and access the value stored at the memory address it points to.

Pointers are a powerful and important feature of the C language, and they are used in many areas of programming. They allow you to manipulate data stored in memory and provide a way to pass data between functions.

Functions

A function in C is a self-contained block of code that performs a specific task and can be called from other parts of a program. Functions are used to organize and structure code, and they allow you to reuse code in multiple places.

Here is an example of a simple function in C:

```
#include <stdio.h>
int add(int x, int y)
{
   return x + y;
}
int main()
{
   int result = add(3, 4);
   printf("Result: %d\n", result);
   return 0;
}
```

In this example, we have defined a function **add** that takes two integers as arguments and returns their sum. The function is defined with the **int** return type, which indicates that it returns an integer value.

The function is called from the **main** function using the function name and passing the arguments 3 and 4. The return value of the function is assigned to the variable **result** and is printed to the console using the **printf** function.

Functions can also be defined to take no arguments or to return no value. For example:

```
#include <stdio.h>
void print_message()
{
   printf("Hello, world!\n");
}
int main()
{
   print_message();
   return 0;
}
```

In this example, we have defined a function **print_message** that takes no arguments and returns no value. The function is defined with the **void** return type, which indicates that it does not return a value. The function simply prints a message to the console.

Functions can be defined with different parameter lists and return types, and they can be called multiple times from different parts of the program. Functions are a powerful and important feature of the C language, and they are used to organize and structure code in a modular way.

Function Parameters

A function parameter in C is a variable that is declared as part of a function's signature and passed to the function when it is called. Function parameters are used to pass data to a function and to specify the type of data that the function expects to receive.

Here is an example of a function with parameters in C:

```
#include <stdio.h>
int add(int x, int y)
{
   return x + y;
}
int main()
{
   int result = add(3, 4);
   printf("Result: %d\n", result);
   return 0;
}
```

In this example, we have defined a function **add** that takes two integers as arguments and returns their sum. The variables **x** and **y** are the function parameters, and they are declared as part of the function's signature.

When the function is called from the **main** function, the arguments 3 and 4 are passed to the function and are assigned to the parameters **x** and **y**, respectively. The function performs the addition operation and returns the result, which is then assigned to the variable **result** and printed to the console.

Function parameters can be of any data type, and a function can have multiple parameters of different types. For example:

```
#include <stdio.h>

float average(float a, float b, float c)
{
    return (a + b + c) / 3;
}

int main()
{
    float result = average(3.0, 4.0, 5.0);
    printf("Result: %f ", result);
    return 0;
}
```

Function Declaration

A function declaration in C is a statement that provides the compiler with the information it needs to verify that a function is defined correctly and to generate code to call the function. A function declaration specifies the function's name, return type, and parameter list.

A function declaration is also known as a function prototype, and it is typically placed at the beginning of a C program, before the **main** function.

Here is an example of a function declaration in C:

```
#include <stdio.h>
int add(int x, int y);  // function declaration
int main()
{
   int result = add(3, 4);
   printf("Result: %d\n", result);
   return 0;
}
int add(int x, int y)  // function definition
{
   return x + y;
}
```

In this example, we have declared the function **add** before the **main** function using a function declaration. The function declaration specifies the function's name, return type, and parameter list, but it does not include the function's implementation.

The function is then defined after the **main** function, where the implementation is provided. The function definition includes the function's name, return type, parameter list, and the code that is executed when the function is called.

Function declarations are used to inform the compiler about the existence and signature of a function, and they are necessary in C because functions can be defined in any order. Function declarations allow you to call a function before it is defined, as long as the function is defined before the program is executed.

Function declarations are also useful when writing large programs, as they allow you to define functions in separate source files and link them together when the program is compiled. This helps to organize and structure the code in a modular way.

Recursion

Recursion is a programming technique in which a function calls itself repeatedly until a certain condition is met. Recursion is a powerful tool that allows you to solve problems by breaking them down into smaller and simpler subproblems.

Recursion is often used to solve problems that can be naturally divided into smaller subproblems, such as sorting and searching algorithms, tree traversals, and mathematical calculations.

Here is an example of a recursive function in C that calculates the factorial of a number:

```
#include <stdio.h>
int factorial(int n)
{
   if (n == 0) {
       return 1;
   }
   else {
       return n * factorial(n - 1);
   }
}
int main()
{
   int result = factorial(5);
   printf("Result: %d\n", result);
   return 0;
}
```

In this example, we have defined a recursive function **factorial** that calculates the factorial of a given number **n**. The function has a base case that is triggered when **n** is 0, in which case it returns 1. For all other values of **n**, the function calls itself with **n - 1** as the argument, and it returns the product of **n** and the result of the recursive call.

The function is called from the **main** function with the argument 5, and it calculates the factorial of 5 by calling itself repeatedly until the base case is reached. The final result is then printed to the console.

Recursion can be a useful technique for solving problems, but it is important to carefully consider the efficiency and complexity of recursive algorithms and to ensure that they terminate correctly. Recursive functions should always have a base case that stops the recursion, or they may enter an infinite loop.

Math Functions

There are several ways to perform math operations in C. The C language provides a number of built-in operators and functions for performing mathematical calculations, such as addition, subtraction, multiplication, division, and modulus.

Here are some examples of using mathematical operators in C:

```
#include <stdio.h>
int main()
{
   int a = 5;
   int b = 3;
   int c;
   c = a + b;  // addition
   printf("%d + %d = %d\n", a, b, c);
   c = a - b;  // subtraction
   printf("%d - %d = %d\n", a, b, c);
   c = a * b;  // multiplication
   printf("%d * %d = %d\n", a, b, c);
   c = a / b;  // division
   printf("%d / %d = %d\n", a, b, c);
   c = a % b;  // modulus
   printf("%d %% %d = %d\n", a, b, c);
   return 0;
}
```

In this example, we have performed several basic math operations using the built-in operators. The **+** operator is used for addition, the **-** operator is used for subtraction, the ***** operator is used for multiplication, the **/** operator is used for division,

Structures (structs)

In C, a structure is a user-defined data type that can hold a collection of variables of different data types. Structures are used to represent complex data types, such as records or objects, and they allow you to group related data together and access it using a single variable.

Here is an example of defining a structure in C:

```
#include <stdio.h>
struct point {
   int x;
   int y;
};
int main()
{
   struct point p = {0, 0};
   printf("Point: (%d, %d)\n", p.x, p.y);
   return 0;
}
```

In this example, we have defined a structure called **point** that has two integer fields: **x** and **y**. The structure is defined using the **struct** keyword, followed by the structure name and the list of fields in curly braces.

To create a variable of the **point** structure type, we use the **struct** keyword followed by the structure name and a variable name. In this example, we have created a variable **p** of the **point** structure type and initialized it with the values 0 and 0.

To access the fields of a structure, you can use the **.** operator followed by the field name. In this example, we have used the **printf** function to print the values of the **x** and **y** fields of the **p** structure.

Structures are a useful tool for representing complex data types and for organizing and accessing data in a structured way. They are often used in conjunction with pointers to create linked lists, trees, and other data structures.

Switch Statement

The **switch** statement in C is a control flow statement that allows you to execute a block of code based on the value of an expression. It can be used as an alternative to multiple **if** statements when you need to perform different actions based on several possible values of the expression.

Here is the basic syntax of a **switch** statement in C:

```
switch (expression) {
   case value1:
       // code block to execute if expression is value1
       break;
   case value2:
       // code block to execute if expression is value2
       break;
   ...
   default:
       // code block to execute if expression is none of the above values
       break;
}
```

In the **switch** statement, the **expression** is evaluated and compared to each of the **case** values. If a match is found, the corresponding code block is executed. If no match is found, the code block under the **default** label is executed.

It's important to include the **break** statement at the end of each code block, as it breaks out of the **switch** statement and prevents the execution of subsequent code blocks. If you omit the **break** statement, the code will continue to execute until it reaches a **break** or the end of the **switch** statement.

Here is an example of using a **switch** statement in a C program:

```
#include <stdio.h>
int main()
```

```
{
   char grade = 'B';
   switch (grade) {
       case 'A':
           printf("Excellent\n");
           break;
       case 'B':
           printf("Good\n");
           break;
       case 'C':
           printf("Average\n");
           break;
       case 'D':
           printf("Poor\n");
           break;
       default:
           printf("Invalid grade\n");
           break;
   }
   return 0;
}
```

In this example, we have a variable **grade** that stores a letter grade. The **switch** statement compares the value of **grade** to each of the **case** values, and if a match is found, the corresponding message is printed to the console. If no match is found, the message "Invalid grade" is printed.

If you run this program with **grade** set to 'B', it will print the message "Good" to the console.

While

The **while** loop in C is a control flow statement that allows you to execute a block of code repeatedly while a Boolean expression is true. The loop will continue to iterate as long as the expression is true, and it will stop when the expression becomes false.

Here is the basic syntax of a **while** loop in C:

```
while (expression) {
   // code block to be executed
}
```

In the **while** loop, the **expression** is evaluated at the beginning of each iteration. If the expression is true, the code block is executed. If the expression is false, the loop is terminated and control is transferred to the next statement after the loop.

It's important to make sure that the Boolean expression eventually becomes false, or the loop will run indefinitely and cause an infinite loop. You can use variables and operators within the expression to create a conditional that can change over time.

Here is an example of using a **while** loop in a C program:

```
#include <stdio.h>
int main()
{
   int i = 0;
   while (i < 10) {
       printf("%d\n", i);
       i++;
   }
   return 0;
}
```

In this example, we have a variable **i** that is initialized to 0. The **while** loop continues to execute as long as **i** is less than 10. Inside the loop, we print the value of **i** to the console and then increment **i** by 1 using the **++** operator.

This program will print the numbers 0 through 9 to the console, one per line, and then terminate the loop when **i** becomes 10.

The Do/While Loop

The **do...while** loop in C is similar to the **while** loop, but it executes the code block at least once before checking the Boolean expression. This means that the code block will always be executed at least once, regardless of the value of the expression.

Here is the basic syntax of a **do...while** loop in C:

```
do {
   // code block to be executed
} while (expression);
```

In the **do...while** loop, the code block is executed first, and then the **expression** is evaluated. If the expression is true, the code block is executed again. The loop continues to iterate as long as the expression is true, and it stops when the expression becomes false.

Here is an example of using a **do...while** loop in a C program:

```
#include <stdio.h>
int main()
{
   int i = 0;
   do {
       printf("%d\n", i);
       i++;
   } while (i < 10);
   return 0;
}
```

In this example, we have a variable **i** that is initialized to 0. The **do...while** loop executes the code block once, printing the value of **i** to the console and incrementing **i** by 1. Then, the expression **i < 10** is evaluated. If it is true, the code block is executed again, and the loop continues to iterate until

i becomes 10.

The **do...while** loop is useful when you want to ensure that the code block is executed at least once, regardless of the value of the expression. It's important to note that the expression is evaluated after the code block is executed, so the code block may be executed multiple times if the expression is true.

for

The **for** loop in C is a control flow statement that allows you to execute a block of code repeatedly for a specified number of iterations. It is often used when you need to perform a task multiple times with a loop variable that changes each time through the loop.

Here is the basic syntax of a **for** loop in C:

```
for (initialization; condition; increment) {
   // code block to be executed
}
```

In the **for** loop, the **initialization** statement is executed first, and is usually used to initialize a loop variable. The **condition** is then evaluated, and if it is true, the code block is executed. After the code block is executed, the **increment** statement is executed and the loop variable is updated. The loop continues to iterate as long as the **condition** is true, and it stops when the **condition** becomes false.

Here is an example of using a **for** loop in a C program:

```
#include <stdio.h>
int main()
{
   for (int i = 0; i < 10; i++) {
       printf("%d\n", i);
   }
   return 0;
}
```

In this example, we have a loop variable **i** that is initialized to 0 in the **initialization** statement. The **condition** is **i < 10**, which means the loop will continue to iterate as long as **i** is less than 10. The **increment** statement is **i++**, which increments **i** by 1 after each iteration.

This program will print the numbers 0 through 9 to the console, one per line, and then terminate the loop when **i** becomes 10.

The **for** loop is a convenient way to perform a task multiple times with a loop variable that changes each time through the loop. It's important to make sure that the loop variable is updated in a way that will eventually cause the **condition** to become false, or the loop will run indefinitely and cause an infinite loop.

Break

The **break** statement in C is a control flow statement that is used to exit a loop or a switch statement prematurely. It is often used to terminate a loop early when a certain condition is met, or to exit a switch statement when a matching **case** label is found.

Here is an example of using the **break** statement to exit a loop early in a C program:

```
#include <stdio.h>
int main()
{
   for (int i = 0; i < 10; i++) {
       if (i == 5) {
           break;
       }
       printf("%d\n", i);
   }
   return 0;
}
```

In this example, we have a **for** loop that iterates from 0 to 9, and a **if** statement that checks if **i** is equal to 5. If **i** is equal to 5, the **break** statement is executed and the loop is terminated early.

This program will print the numbers 0 through 4 to the console, one per line, and then terminate the loop when **i** becomes 5. The number 5 will not be printed because the **break** statement is executed before the **printf** statement.

The **break** statement can also be used to exit a **switch** statement early.

Continue

The **continue** statement in C is a control flow statement that is used to skip the remainder of the current iteration of a loop and move on to the next iteration. It is often used to skip certain iterations of a loop when a certain condition is met, or to skip over certain code blocks in a loop.

Here is an example of using the **continue** statement to skip an iteration of a loop in a C program:

```
#include <stdio.h>
int main()
{
   for (int i = 0; i < 10; i++) {
       if (i % 2 == 1) {
           continue;
       }
       printf("%d\n", i);
   }
   return 0;
}
```

In this example, we have a **for** loop that iterates from 0 to 9, and an **if** statement that checks if **i** is odd (has a remainder of 1 when divided by 2). If **i** is odd, the **continue** statement is executed and the remainder of the current iteration is skipped.

This program will print the even numbers 0 through 8 to the console, one per line, and then terminate the loop when **i** becomes 10. The odd numbers 1 through 9 will be skipped because of the **continue** statement.

goto

The **goto** statement in C is a control flow statement that is used to transfer control to a labeled statement in the same function. It is often used to break out of nested loops or to skip over certain code blocks.

The **goto** statement should be used with caution, as it can make code more difficult to understand and maintain. It is generally considered a poor programming practice, and there are often better alternatives to using **goto**.

Here is the basic syntax of the **goto** statement in C:

```
goto label;
...
label:
   // code block to be executed
```

In the **goto** statement, the **label** is a identifier that represents a labeled statement in the same function. When the **goto** statement is executed, control is transferred to the labeled statement.

Here is an example of using the **goto** statement to break out of a nested loop in a C program:

```
#include <stdio.h>
int main()
{
   int i, j;
   for (i = 0; i < 10; i++) {
       for (j = 0; j < 10; j++) {
           if (i == 5 && j == 5) {
               goto end;
           }
           printf("%d, %d\n", i, j);
       }
   }
end:
   printf("Done\n");
   return 0;
}
```

C maths

Let's see a simple example of math functions found in math.h header file.

```
#include<stdio.h>
#include <math.h>
int main(){
    printf("\n%f",ceil(3.6));
    printf("\n%f",ceil(3.3));
    printf("\n%f",floor(3.6));
    printf("\n%f",floor(3.2));
    printf("\n%f",sqrt(16));
    printf("\n%f",sqrt(7));
    printf("\n%f",pow(2,4));
    printf("\n%f",pow(3,3));
    return 0;
}
```

Dynamic memory allocation

Dynamic memory allocation in C programming refers to the process of allocating memory during runtime—i.e., while the program is running—rather than prior to the program's execution. This is crucial in scenarios where the amount of memory to be used is not known to the developer at compile time. For example, when the memory requirement depends on user input. In C, dynamic memory allocation is performed using a set of functions in the C standard library, namely **malloc()**, **calloc()**, **realloc()**, and **free()**, which are available in the **stdlib.h** header file.

Here’s a brief overview of these functions:

malloc():

Stands for "memory allocation".

Allocates a specified number of bytes and returns a pointer to the allocated memory.

The memory allocated by **malloc** is uninitialized, meaning that the values at that memory location are indeterminate.

```
int *arr = (int*)malloc(10 * sizeof(int)); // allocates memory for an
array of 10 integers
```

calloc():

Stands for "contiguous allocation".

Similar to **malloc()**, but it initializes the allocated memory to zero.

Requires two arguments: the number of elements to allocate and the size of each element.

```
int *arr = (int*)calloc(10, sizeof(int)); // allocates memory for an
array of 10 integers and initializes all bytes to zero
```

realloc():

Stands for "re-allocation".

Used to resize the previously allocated memory block without losing old data.

Requires two arguments: a pointer to the memory previously allocated with **malloc**, **calloc** or

realloc and the new size.

```
arr = (int*)realloc(arr, 15 * sizeof(int)); // resizes the previous
memory allocation to an array of 15 integers
```

free():

Deallocates the memory previously allocated by **malloc()**, **calloc()**, or **realloc()**.

Helps to prevent memory leaks by freeing up memory when it is no longer needed.

```
free(arr); // deallocates memory for the array
```

When using dynamic memory allocation, there are some best practices and precautions that should be kept in mind:

- Always check if the memory allocation was successful by verifying that the returned pointer is not **NULL**.
- Avoid memory leaks by ensuring every allocation has a corresponding **free()** when the memory is no longer needed.
- Be cautious when working with uninitialized memory, such as that allocated by **malloc()**, to prevent undefined behavior.
- Avoid assumptions about the layout of dynamic memory (e.g., alignment, padding).
- Be aware of the potential for fragmentation in dynamic memory allocation, particularly when allocating many small blocks of memory.

Dynamic memory plays a vital role in creating complex data structures such as trees, graphs, and linked lists, which have variable sizes and structures that are often unknown until runtime.

comments

In C programming, comments are used to provide explanations or annotations in the source code. Comments make the code more understandable for developers who may work on the codebase later. The C language supports two types of comments:

Single-line comments: These comments start with **//** and continue to the end of the line. Anything written after **//** on the same line is considered a comment and is ignored by the compiler.

```
// This is a single-line comment.
int main() {
   // This is also a single-line comment.
   return 0;  // Comment after code.
}
```

Multi-line comments: These comments start with **/*** and end with ***/**. Anything between these delimiters is a comment and will be ignored by the compiler.

```
/*
This is a multi-line comment.
It spans several lines.
```

```
*/
int main() {
   /* This is also a multi-line comment,
      but it's on multiple lines to fit beside code. */
   return 0;
}
```

Important Notes:

Nested multi-line comments are not supported. That is, you cannot start a new multi-line comment before closing the previous one.

```
/* This is the start of the first comment
/* This is incorrect and will cause a compilation error */
This is still part of the first comment */
```

Comments do not affect the behavior of the program; they are only for human readers.

Be careful when commenting out code. Using multi-line comments around code that already contains multi-line comments can lead to compilation errors.

Best Practices:

- Use comments to explain why certain code exists, not what it does (unless it's particularly complex).
- Keep comments up-to-date when modifying the code; outdated comments can be misleading.
- Avoid commenting out large sections of code; use version control systems for that.
- Use consistent commenting style throughout the codebase.

AI

Implementing AI for pattern recognition, prediction, decision-making, and learning from data in C requires a significant amount of work and is much more complex than implementing basic arithmetic operations.

C is a lower-level programming language and does not have as many libraries and tools for AI and machine learning as languages like Python. However, it is still possible to implement machine learning algorithms in C from scratch or by using certain libraries.

Here’s a rough guide on how you might approach this:

1. Pattern Recognition:

You might implement algorithms like Neural Networks or Support Vector Machines from scratch or use libraries like FANN (Fast Artificial Neural Network Library).

2. Prediction:

For prediction, you might use regression algorithms. If you have time series data, you might use algorithms like ARIMA. Implementing these from scratch in C would require a solid understanding of the underlying mathematics.

3. Decision-Making:

Decision trees or reinforcement learning algorithms could be implemented for decision-making tasks.

4. Learning from Data:

You would need to implement or use existing implementations of machine learning algorithms, ensuring that they can learn and update their models based on input data.

Example: A Very Simple Linear Regression in C

Here’s an example of a very simple linear regression implementation in C for prediction:

```c
#include <stdio.h>
float mean(int arr[], int n) {
   float sum = 0;
   for (int i = 0; i < n; i++)
       sum += arr[i];
   return sum / n;
}
float covariance(int x[], int y[], int n, float x_mean, float y_mean) {
   float covar = 0;
   for (int i = 0; i < n; i++)
       covar += (x[i] - x_mean) * (y[i] - y_mean);
   return covar;
}
float variance(int arr[], int n, float mean) {
   float var = 0;
   for (int i = 0; i < n; i++)
       var += (arr[i] - mean) * (arr[i] - mean);
   return var;
}
float coefficients(int x[], int y[], int n) {
   float x_mean, y_mean, b1, b0;
   x_mean = mean(x, n);
   y_mean = mean(y, n);
   b1 = covariance(x, y, n, x_mean, y_mean) / variance(x, n, x_mean);
   b0 = y_mean - b1 * x_mean;
   printf("Coefficients: b0 = %.2f, b1 = %.2f\n", b0, b1);
   return b0 + b1 * x[n-1];  // For simplicity, let's predict the next
value
}
int main() {
   int x[] = {1, 2, 3, 4, 5};
   int y[] = {2, 4, 5, 4, 5};
   int n = sizeof(x) / sizeof(x[0]);
   float prediction = coefficients(x, y, n);
   printf("Predicted value: %.2f\n", prediction);
   return 0;
}
```

This example implements a simple linear regression model. It predicts a value based on a given dataset. This is a very basic example and real-world AI applications require more sophisticated algorithms and data preprocessing.

Implementing AI in C is feasible but challenging, especially for complex tasks. For most AI applications, especially those requiring deep learning, a high-level language like Python, with access to libraries like TensorFlow or PyTorch, is generally more appropriate and efficient.

MySQL

Creating a CRUD application in C with a MySQL database involves using the MySQL C API. Here's a step-by-step guide to create such an application:

Setup

Install MySQL and MySQL C Connector:

Make sure MySQL is installed on your system. Also, you need to install the MySQL C Connector:

```
sudo apt-get install libmysqlclient-dev
```

Create MySQL Database and Table:

Open your MySQL client and create a database and a table:

```
CREATE DATABASE crud_db;
USE crud_db;
CREATE TABLE users (
 id INT AUTO_INCREMENT PRIMARY KEY,
 name VARCHAR(255),
 email VARCHAR(255)
);
```

Code

Include Necessary Headers:

Create a file crud.c and include necessary headers:

```
#include <mysql/mysql.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Define MySQL Connection Function:

```
MYSQL *connect_db() {
   MYSQL *conn = mysql_init(NULL);
```

```
    if (conn == NULL) {
        fprintf(stderr, "mysql_init() failed\n");
        exit(EXIT_FAILURE);
    }
    if (mysql_real_connect(conn, "localhost", "root", "password",
"crud_db", 0, NULL, 0) == NULL) {
        fprintf(stderr, "mysql_real_connect() failed\n");
        mysql_close(conn);
        exit(EXIT_FAILURE);
    }
    return conn;
}
```

Define CRUD Operations:

```
void create_user(MYSQL *conn, const char *name, const char *email) {
    char query[256];
    snprintf(query, sizeof(query), "INSERT INTO users (name, email) VALUES
('%s', '%s')", name, email);
    if (mysql_query(conn, query)) {
        fprintf(stderr, "INSERT error: %s\n", mysql_error(conn));
    } else {
        printf("User added.\n");
    }
}
void read_users(MYSQL *conn) {
    if (mysql_query(conn, "SELECT * FROM users")) {
        fprintf(stderr, "SELECT error: %s\n", mysql_error(conn));
        return;
    }
    MYSQL_RES *result = mysql_store_result(conn);
    if (result == NULL) {
        fprintf(stderr, "mysql_store_result() failed: %s\n",
mysql_error(conn));
        return;
    }
    int num_fields = mysql_num_fields(result);
    MYSQL_ROW row;
    while ((row = mysql_fetch_row(result))) {
        for (int i = 0; i < num_fields; i++) {
            printf("%s ", row[i] ? row[i] : "NULL");
        }
        printf("\n");
    }
    mysql_free_result(result);
}
void update_user(MYSQL *conn, int id, const char *name, const char
*email) {
    char query[256];
    snprintf(query, sizeof(query), "UPDATE users SET name='%s', email='%s'
WHERE id=%d", name, email, id);
```

```
   if (mysql_query(conn, query)) {
       fprintf(stderr, "UPDATE error: %s\n", mysql_error(conn));
   } else {
       printf("User updated.\n");
   }
}
void delete_user(MYSQL *conn, int id) {
   char query[256];
   snprintf(query, sizeof(query), "DELETE FROM users WHERE id=%d", id);
   if (mysql_query(conn, query)) {
       fprintf(stderr, "DELETE error: %s\n", mysql_error(conn));
   } else {
       printf("User deleted.\n");
   }
}
```

Main Function:

```
int main() {
   MYSQL *conn = connect_db();
   int choice, id;
   char name[255], email[255];
   while (1) {
       printf("1. Create User\n");
       printf("2. Read Users\n");
       printf("3. Update User\n");
       printf("4. Delete User\n");
       printf("5. Exit\n");
       printf("Enter your choice: ");
       scanf("%d", &choice);
       switch (choice) {
           case 1:
               printf("Enter name: ");
               scanf("%s", name);
               printf("Enter email: ");
               scanf("%s", email);
               create_user(conn, name, email);
               break;
           case 2:
               read_users(conn);
               break;
           case 3:
               printf("Enter user ID: ");
               scanf("%d", &id);
               printf("Enter new name: ");
               scanf("%s", name);
               printf("Enter new email: ");
               scanf("%s", email);
               update_user(conn, id, name, email);
               break;
           case 4:
               printf("Enter user ID: ");
               scanf("%d", &id);
               delete_user(conn, id);
               break;
           case 5:
               mysql_close(conn);
               exit(EXIT_SUCCESS);
           default:
               printf("Invalid choice.\n");
       }
   }
   mysql_close(conn);
   return 0;
}
```

Compilation and Execution

Compile the program:

```
gcc -o crud crud.c -lmysqlclient
```

Run the program:

```
./crud
```

This will provide you with a basic terminal-based CRUD application using C and MySQL. The program allows you to create, read, update, and delete users from the database.