**ASSIGNMENT**



**COURSE –** BCA(Hons) AI & DS

**BATCH** – 2023 -27

Submitted By- Chirag Paliwal 23151598

Submitted To – Mr. Rishi Sir

**ASSIGNMENT**

**Q.1 What are Constants and variables, Types of Constants, Keywords, Rules for identifiers, int, float, char, double, long, void .**

**Sol-1 Constants** are read-only variables whose values cannot be modified once they are declared in the program. They can be of different types such as integer, floating-point, string, or character constants. In C language, the “const” keyword is used to define constants.

**Variables** are memory locations that store values that can be changed during program execution. They can be of different types such as integer, floating-point, character, or void.

**Keywords** are reserved words in C language that have predefined meanings and cannot be used as variable names or function names. Examples of keywords include if, else, while, for, int, float, char, double, long, and void.

**Identifiers** are names given to variables, functions, and other user-defined entities in a program. Identifiers must follow certain rules such as starting with a letter or underscore, not containing spaces or special characters, and not being a keyword.

“int”, “float”, “char”, “double”, “long”, and “void” are data types in C language. int is used to represent integers, float is used to represent floating-point numbers, char is used to represent characters, double is used to represent double-precision floating-point numbers, long is used to represent long integers, and void is used to represent no type.

There are different types of constants in C language such as integer constants, floating-point constants, character constants, and string constants. Integer constants can be decimal, octal, or hexadecimal. Floating-point constants can be represented in decimal or exponential form. Character constants are enclosed in single quotes, and string constants are enclosed in double quotes.

**Q2. Explain with examples Arithmetic Operators, Increment and Decrement Operators, Relational Operators, Logical Operators, Bitwise Operators, Conditional Operators, Type Conversions, and expressions, Precedence, and associativity of operators.**

**Sol-2 Arithmetic Operators** are used to perform mathematical operations on operands. There are 5 arithmetic operators in C programming language: +, -, \*, /, and %. Here’s an example of arithmetic operators in C:

int a = 10, b = 20, c;

c = a + b;

c = a - b;

c = a \* b;

c = a / b;

c = a % b;

**Increment and Decrement Operators** are used to increment or decrement the value of a variable by 1. There are two types of increment and decrement operators: ++ and --. Here’s an example of increment and decrement operators in C:

int a = 10;

a++;

a--;

**Relational Operators** are used to compare two values. They return either true or false. There are 6 relational operators in C programming language: ==, !=, >, <, >=, and <=. Here’s an example of relational operators in C:

int a = 10, b = 20;

if (a == b) {

printf("a is equal to b");

} else if (a > b) {

printf("a is greater than b");

} else {

printf("a is less than b");

}

**Logical Operators** are used to combine two or more conditions. There are 3 logical operators in C programming language: && (AND), || (OR), and ! (NOT). Here’s an example of logical operators in C:

int a = 10, b = 20, c = 30;

if (a > b && a > c) {

printf("a is the largest number");

} else if (b > a && b > c) {

printf("b is the largest number");

} else {

printf("c is the largest number");

}

**Bitwise Operators** are used to perform bitwise operations on binary numbers. There are 6 bitwise operators in C programming language: & (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), and >> (right shift). Here’s an example of bitwise operators in C:

unsigned int a = 60;

unsigned int b = 13;

int c;

c = a & b;

c = a | b;

c = a ^ b;

c = ~a;

c = a << 2;

c = a >> 2;

**Conditional Operator** is a ternary operator that takes three operands. It is used to evaluate a condition and return one of two values depending on whether the condition is true or false. Here’s an example of conditional operator in C:

int a = 10, b = 20, c;

c = (a > b) ? a : b;

**Type Conversions** are used to convert one data type to another. There are two types of type conversions in C programming language: **implicit type conversion** and **explicit type conversion**. Here’s an example of type conversion in C:

int a = 10;

float b = 20.5;

float c = a + b;

int d = (int) b;

**Precedence** and **associativity** of operators determine the order in which operators are evaluated in an expression. Here’s a table that shows the precedence and associativity of operators in C programming language:

| **Operator** | **Description** | **Associativity** |
| --- | --- | --- |
| () | Parentheses | Left to right |
| ++, -- | Increment, Decrement | Right to left |
| +, - | Unary plus, Unary minus | Right to left |
| \*, /, % | Multiplication, Division, Modulus | Left to right |
| +, - | Addition, Subtraction | Left to right |
| <<, >> | Left shift, Right shift | Left to right |
| <, <=, >, >= | Rel |  |

**Q.3 Explain with Example conditional statements if, if-else, elseif, nested if else statement .**

**Sol-3** Conditional statements are used in programming to execute a specific block of code based on certain conditions. There are four types of conditional statements: **“if”, “if-else”, “elseif”,** and **“nested if-else”.**

* **if statement**: The “if” statement is used to execute a block of code if a condition is true. If the condition is false, the code block is skipped. Here’s an example :

int age = 18;

if (age >= 18) {

printf("You are eligible to vote.");

}

* **if-else statement**: The “if-else” statement is used to execute one block of code if a condition is true and another block of code if the condition is false. Here’s an example :

int age = 16;

if (age >= 18) {

printf("You are eligible to vote.");

} else {

printf("You are not eligible to vote.");

}

* **elseif statement**: The “elseif” statement is used to execute a block of code if a condition is true and another block of code if the condition is false. You can use multiple “elseif” statements to check for multiple conditions. Here’s an example :

int grade = 75;

if (grade >= 90) {

printf("A");

} else if (grade >= 80) {

printf("B");

} else if (grade >= 70) {

printf("C");

} else {

printf("D");

}

* **nested if-else statement**: The “nested if-else” statement is used to execute a block of code if a condition is true and another block of code if the condition is false. You can also use an “if-else” statement inside another “if-else” statement. Here’s an example :

int age = 18;

if (age >= 18) {

if (age == 18) {

printf("You just became eligible to vote.");

} else {

printf("You are eligible to vote.");

}

} else {

printf("You are not eligible to vote.");

}

**Q.4** **Explain Switch Case statement with example.**

**Sol-4** The “switch-case” statement is used to execute a block of code based on the value of a variable or an expression.There is another type of conditional statement called the “switch-case” statement. Here’s an example :

#include <stdio.h>

int main() {

char operator;

double num1, num2;

printf("Enter an operator (+, -, \*, /): ");

scanf("%c", &operator);

printf("Enter two operands: ");

scanf("%lf %lf", &num1, &num2);

switch (operator) {

case '+':

printf("%.1lf + %.1lf = %.1lf", num1, num2, num1 + num2);

break;

case '-':

printf("%.1lf - %.1lf = %.1lf", num1, num2, num1 - num2);

break;

case '\*':

printf("%.1lf \* %.1lf = %.1lf", num1, num2, num1 \* num2);

break;

case '/':

printf("%.1lf / %.1lf = %.1lf", num1, num2, num1 / num2);

break;

default:

printf("Error! Invalid operator.");

}

return 0;

}

**Q.5 Explain Loops, for loop, while loop, do while loop with examples.**

**Sol-5 Loops** are used in programming to execute a block of code repeatedly until a certain condition is met. There are three types of loops in C programming language: “**for loop”**, “**while loop”**, and “**do-while loop”**.

**For loop** is used when we know the number of times, we want to execute a block of code. Here’s an example of a for loop :

for (int i = 1; i <= 10; i++) {

printf("%d ", i);

}

**While loop** is used when we don’t know the number of times we want to execute a block of code. Here’s an example of a while loop :

int i = 1;

while (i <= 10) {

printf("%d ", i);

i++;

}

**Do-while loop** is similar to the while loop, but the difference is that the code inside the loop is executed at least once, even if the condition is false. Here’s an example of a do-while loop :

int i = 1;

do {

printf("%d ", i);

i++;

} while (i <= 10);

**Q.6 Explain with examples debugging importance, tools common errors: syntax, logic, and runtime errors, debugging, and Testing C Programs.**

**Sol-6** **Debugging** is the process of finding and resolving errors, or bugs, in a software program. Debugging is important for ensuring that the software works correctly and efficiently, and does not cause any harm or inconvenience to the users or the system. Debugging can also help improve the quality and maintainability of the software, as well as reduce the development time and cost.

There are various tools available for debugging, such as debuggers, trace tools, and profilers. Debuggers are programs that allow developers to control and monitor the execution of a software program, and inspect and modify its state. Trace tools are programs that record the events and messages related to the software program, which can be used to analyse its behaviour and performance. Profilers are programs that measure the time and space complexity of a software program, which can be used to identify and optimize the bottlenecks and resource consumption. There are three common types of errors that can occur in a software program**: syntax errors, runtime errors,** and **logic errors**.

**Syntax errors** are errors that violate the rules and structure of the programming language, such as missing or misplaced punctuation, incorrect spelling, or invalid syntax. Syntax errors can be easily detected and corrected by the compiler or the interpreter, which will report the location and the nature of the error. For example, in C, a syntax error can occur if a semicolon is missing at the end of a statement, or if a variable is declared with an invalid name.

**Runtime errors** are errors that occur when the software program is executed, and cause the program to terminate abnormally, or produce incorrect or unexpected results. Runtime errors can be caused by various factors, such as invalid input, memory allocation failure, division by zero, or accessing an out-of-bound array element. Runtime errors can be difficult to identify and resolve, as they may not always occur under the same conditions, or may not be reported clearly by the system. For example, in C, a runtime error can occur if a pointer is dereferenced without being initialized, or if a file is opened without checking its existence.

**Logic errors** are errors that occur when the software program does not perform the intended task or function, or does not produce the desired output. Logic errors can be caused by faulty design, incorrect algorithm, or human mistake. Logic errors can be the most challenging to identify and resolve, as they may not cause any visible or noticeable symptoms, or may only appear under certain circumstances. Logic errors can only be detected by testing and debugging the software program, and comparing its output with the expected output. For example, in C, a logic error can occur if a loop condition is incorrect, or if a variable is assigned with a wrong value.

**Debugging** and **Testing** are related but not synonymous processes. Testing is the process of checking and verifying the functionality and quality of a software program, by running it under various scenarios and inputs, and comparing its output with the expected output. Testing can help identify the presence and the impact of errors in a software program, but it cannot help locate or fix them. Debugging, on the other hand, is the process of finding and resolving errors in a software program, by examining and modifying its code and state, and re-running it to ensure that the errors are fixed. Debugging can help improve the functionality and quality of a software program, but it cannot guarantee that there are no errors left.

Some examples of debugging and testing C programs:

* To debug a C program using a debugger, such as GDB, one can use the following steps:
  1. Compile the C program with the -g flag, which enables debugging information.
  2. Run the debugger with the executable file as an argument, such as gdb a.out.
  3. Set breakpoints at the locations where the errors are suspected, such as break main or break 10.
  4. Run the program until it reaches a breakpoint, such as run.
  5. Examine and modify the values of variables, expressions, and memory locations, such as print x or set x = 10.
  6. Step through the program line by line, or skip over functions, such as next or step.
  7. Continue the program execution until the next breakpoint or the end, such as continue.
  8. Quit the debugger, such as quit.
* To test a C program using a unit testing framework, such as CUnit, one can use the following steps:
  1. Include the CUnit header file, such as #include <CUnit/CUnit.h>.
  2. Define the test functions that perform the assertions, such as void test\_add(void) { CU\_ASSERT(add(2, 3) == 5); }.
  3. Register the test functions to the test suite, such as CU\_add\_test(suite, "test\_add", test\_add); .
  4. Initialize the CUnit test registry, such as CU\_initialize\_registry();
  5. Create the test suite, such as CU\_pSuite suite = CU\_add\_suite("suite", NULL, NULL); .
  6. Run the test suite, such as CU\_basic\_run\_suite(suite); .
  7. Clean up the CUnit test registry, such as CU\_cleanup\_registry(); .

**Q.7 What is the user defined and pre-defined functions. Explain with example call by value and call by reference .**

**Sol-7** **Functions** are a set of instructions that perform a specific task. Functions can be classified into two types: “**user-defined functions”** and “**predefined functions”**.

**User-defined functions** are functions that are created by the programmer to perform a specific task. They are defined by the programmer and can be called from any part of the program. Here’s an example of a user-defined function that adds two numbers:

int add(int a, int b) {

return a + b;

}

**Predefined functions** are functions that are already defined in the C library. They are also known as standard library functions. Some examples of predefined functions are : printf(), scanf(), sqrt(), etc.

Here’s an example of a function using **call by value**:

void swap(int a, int b) {

int temp = a;

a = b;

b = temp;

}

int main() {

int x = 10, y = 20;

swap(x, y);

printf("x = %d, y = %d", x, y);

return 0;

}

Here’s an example of a function using **call by reference**:

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int x = 10, y = 20;

swap(&x, &y);

printf("x = %d, y = %d", x, y);

return 0;

}

**Q.8** **1) Explain with Passing and returning arguments to and from Function.**

**2) Explain Storage classes, automatic, static, register, external.**

**3) Write a program for two strings S1 and S2. Develop a C Program for following operations**

**a) Display a concatenated output of S1 and S2 .**

**b) Count the number of characters and empty spaces in S1and S2.**

**Sol-8** **1)** Passing and returning arguments to and from functions are two ways of communicating data between functions. Arguments are the values that are passed to a function when it is called. Return values are the values that are returned by a function when it is finished. There are two methods of passing arguments to a function by value and by reference.

* **By value**: This method copies the value of the argument into a local variable of the function. Any changes made to the local variable do not affect the original argument. This is the default method of passing arguments . For example:

#include <stdio.h>

int add(int x, int y) {

x = x + 10;

y = y + 10;

return x + y;

}

int main() {

int a = 5, b = 6;

int c = add(a, b);

printf("a = %d, b = %d, c = %d\n", a, b, c);

return 0;

}

* **By reference**: This method passes the address of the argument to the function. The function can access and modify the original argument using a pointer. This is useful when you want to change the value of the argument or pass large data structures without copying them. For example:

#include <stdio.h>

void swap(int\* x, int\* y) {

int temp = \*x;

\*x = \*y;

\*y = temp;

}

int main() {

int a = 5, b = 6;

swap(&a, &b);

printf("a = %d, b = %d\n", a, b);

return 0;

}

Returning a value from a function in C is done by using the return keyword followed by the value or expression to be returned. The return type of the function must match the type of the value or expression. The function can return only one value at a time. For example:

#include <stdio.h>

int square(int x) {

return x \* x;

}

int main() {

int a = 5;

int b = square(a);

printf("a = %d, b = %d\n", a, b);

return 0;

}

**2)** Storage classes are used to describe the features of variables and functions, such as their scope, visibility, and lifetime. There are four types of storage classes : auto, extern, static, and register.

* **Automatic**: This is the default storage class for all the variables declared inside a function or a block. Auto variables can only be accessed within the block or function they are declared in, and not outside them. They are assigned a garbage value by default when they are declared. For example:

void func() {

auto int x = 10;

printf("%d\n", x);

}

int main() {

func();

printf("%d\n", x);

return 0;

}

* **External**: This storage class is used to declare variables that are defined elsewhere, usually in another file or outside of all functions. Extern variables are global variables that can be accessed by any function or block in the program. They are initialized only once and their value can be changed in different blocks. For example:

int x = 20;

#include <stdio.h>

extern int x;

int main() {

printf("%d\n", x);

x = 30;

printf("%d\n", x);

return 0;

}

* **Static**: This storage class is used to declare variables that preserve their value even after they go out of their scope. Static variables are initialized only once and exist until the end of the program. They can be either local or global, depending on where they are declared. Local static variables are only visible within the function or block they are declared in, while global static variables are visible throughout the program. Static variables are assigned the value 0 by default when they are declared. For example:

#include <stdio.h>

void func() {

static int x = 10;

x += 5;

printf("%d\n", x);

}

int main() {

func();

func();

printf("%d\n", x);

return 0;

}

* **Register**: This storage class is used to declare variables that are stored in the CPU registers instead of the memory, for faster access. Register variables are similar to auto variables, except that they have a hint to the compiler that they should be stored in the registers if possible. However, modern compilers are very good at optimizing the code, and there is no guarantee that using register variables will make the program faster. Register variables can only be accessed within the block or function they are declared in, and they are assigned a garbage value by default when they are declared. For example:

#include <stdio.h>

int main() {

register int x = 10;

printf("%d\n", x);

return 0;

}

**3) Write a program for two strings S1 and S2:**

#include <stdio.h>

#include <string.h>

int main()

{

char S1[100], S2[100];

char S3[200];

int len1, len2, len3;

int i, j, k;

int spaces1, spaces2;

printf("Enter the first string: ");

fgets(S1, sizeof(S1), stdin);

printf("Enter the second string: ");

fgets(S2, sizeof(S2), stdin);

len1 = strlen(S1);

len2 = strlen(S2);

i = 0;

j = 0;

k = 0;

while (i < len1 - 1)

{

S3[k] = S1[i];

i++;

k++;

}

S3[k] = ' ';

k++;

while (j < len2 - 1)

{

S3[k] = S2[j];

j++;

k++;

}

S3[k] = '\0';

len3 = strlen(S3);

**a) Display the concatenated output:**

printf("The concatenated output is: %s\n", S3);

spaces1 = 0;

spaces2 = 0;

for (i = 0; i < len1 - 1; i++)

{

if (S1[i] == ' ')

{

spaces1++;

}

}

for (j = 0; j < len2 - 1; j++)

{

if (S2[j] == ' ')

{

spaces2++;

}

}

**b)** **Display the number of characters and spaces in the strings:**

printf("The number of characters in the first string is: %d\n", len1 - 1 - spaces1);

printf("The number of spaces in the first string is: %d\n", spaces1);

printf("The number of characters in the second string is: %d\n", len2 - 1 - spaces2);

printf("The number of spaces in the second string is: %d\n", spaces2);

return 0;

}

**Q.9 Explain with examples debugging importance, tools common errors: syntax, logic, and runtime errors, debugging, and Testing C Programs.**

**Sol-9** Debugging is an important skill for any programmer, as it helps to identify and fix errors or bugs in a software system. Debugging can improve the quality, performance, and reliability of a software system, and can also help to learn from mistakes and enhance one’s programming skills. [There are various tools and techniques that can be used for debugging, such as code inspection, debugging tools, unit testing, integration testing, system testing, monitoring, and logging](https://www.geeksforgeeks.org/differences-between-testing-and-debugging/). Some of the common errors that can occur in C programs are:

* **Syntax errors**: These are errors that occur due to incorrect grammar or spelling of the C language. Syntax errors are usually detected by the compiler and prevent the program from running. For example, missing a semicolon, using an undefined variable, or mismatching parentheses are syntax errors. To fix syntax errors, one needs to carefully check the code and correct the mistakes.
* **Logic errors**: These are errors that occur due to incorrect logic or algorithm of the program. Logic errors are not detected by the compiler, but they cause the program to produce wrong or unexpected results. For example, using the wrong operator, looping infinitely, or using incorrect conditions are logic errors. To fix logic errors, one needs to debug the code and trace the flow of execution and data.
* **Runtime errors**: These are errors that occur during the execution of the program. Runtime errors are caused by external factors, such as invalid user input, insufficient memory, or hardware failure. Runtime errors can cause the program to crash or terminate abnormally. For example, dividing by zero, accessing an invalid memory location, or opening a non-existent file are runtime errors. To fix runtime errors, one needs to handle the exceptions and errors using appropriate error handling techniques.

Debugging and testing are two related but different processes in software development. Testing is the process of verifying and validating that a software system meets the requirements and specifications, while debugging is the process of finding and fixing the bugs or errors in the software system. Testing can be done at different levels, such as unit testing, integration testing, system testing, and acceptance testing, using various methods, such as black-box testing, white-box testing, and grey-box testing. Debugging can be done using various tools, such as debuggers, trace tools, and profilers, that can help to monitor, control, and analyse the execution of the program. Some of the examples of debugging tools for C programs are:

* GDB
* Online C Debugger
* Visual Studio

**Q.10** **Explain with example with Structure, Declaration, and Initialization, Structure Variables, Array of Structures, and Use of typedef, Passing Structures to Functions. Define union declaration, and Initialization Passing structures to functions. Explain difference between Structure and Union. Write a program on details of a bank account with the fields account number, account holder’s name, and balance. Write a program to read 10 people’s details and display the record with the highest bank balance.**

**Sol-10** Structure, declaration, and initialization, structure variables, array of structures, and use of typedef, passing structures to functions are some of the important concepts in C programming language. Here is a brief explanation and example of each concept:

**Structure** is a user-defined data type that can group items of different types into a single type. For example, we can define a structure to store the details of a student, such as name, roll number, and marks. The struct keyword is used to define a structure in C.

**Structure declaration** is the syntax to specify the name and the members of a structure. For example, we can declare a structure named student as follows:

struct student {

char name[20];

int roll;

float marks;

};

**Structure initialization** is the process of assigning values to the members of a structure variable. We can initialize a structure variable using assignment operator, initializer list, or designated initializer list. For example, we can initialize a structure variable named stu1 as follows:

struct student stu1;

stu1.roll = 1;

strcpy(stu1.name, "Alice");

stu1.marks = 85.5;

struct student stu2 = {"Bob", 2, 90.0};

struct student stu3 = {.name = "Charlie", .roll = 3, .marks = 95.0};

**Structure variable** is a variable of structure type that can store the values of the structure members. We can declare structure variables either with the structure declaration or after the structure declaration. For example, we can declare structure variables as follows:

struct student {

char name[20];

int roll;

float marks;

} stu1, stu2, stu3;

struct student stu4, stu5, stu6;

**Array of structures** is an array whose elements are of structure type. We can declare an array of structures using the following syntax:

struct structure\_name array\_name[number\_of\_elements];

For example, we can declare an array of structures named stu as follows:

struct student stu[10];

We can initialize and access the elements of an array of structures using the same methods as for a single structure variable. For example, we can initialize and access the first element of the array stu as follows:

stu[0] = {"David", 4, 80.0};

printf("Name: %s\n", stu[0].name);

printf("Roll: %d\n", stu[0].roll);

printf("Marks: %.2f\n", stu[0].marks);

The **typedef** keyword is used to create an alias or a new name for an existing type. We can use typedef to simplify the declaration of structure variables. For example, we can use typedef to create a new name student\_t for the structure type struct student as follows:

typedef struct student student\_t;

We can declare structure variables using the new name student\_t instead of struct student. For example, we can declare a structure variable named stu7 as follows:

student\_t stu7;

**Passing structures** to functions is the process of passing structure variables or pointers as arguments to a function. We can pass structures to functions either by value or by reference. For example, we can define a function named print\_student that takes a structure variable as an argument and prints its details as follows:

void print\_student(struct student s) {

printf("Name: %s\n", s.name);

printf("Roll: %d\n", s.roll);

printf("Marks: %.2f\n", s.marks);

}

**Union declaration** is the syntax to specify the name and the members of a union. A union is a user-defined data type that can store different data types in the same memory location. For example, we can declare a union named data as follows:

union data {

int integer;

float decimal;

char character;

};

U**nion initialization** is the process of assigning a value to one of the members of a union variable. We can initialize a union variable using assignment operator, initializer list, or designated initializer list. For example, we can initialize a union variable named d as follows:

union data d;

d.integer = 10;

union data d = {10};

union data d = {.integer = 10};

The difference between **Structure** and **Union** in C is that a structure can store values for all its members, whereas a union can store a value for only one of its members at a time. A structure allocates memory for each of its members separately, whereas a union allocates memory for all its members in the same location, which is equal to the size of its largest member. For example, if we have a structure and a union with the same members as follows:

struct data {

int integer;

float decimal;

char character;

};

union data {

int integer;

float decimal;

char character;

};

Then, the size of the structure data is 9 bytes , whereas the size of the union data is 4 bytes. Therefore, a structure can store and access values for all its members simultaneously, whereas a union can store and access a value for only one of its members at a time.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct bank\_account {

int account\_number;

char account\_name[20];

float balance;

};

void read\_account(struct bank\_account \*account) {

printf("Enter the account number: ");

scanf("%d", &account->account\_number);

printf("Enter the account holder's name: ");

scanf("%s", account->account\_name);

printf("Enter the balance: ");

scanf("%f", &account->balance);

}

void print\_account(struct bank\_account account) {

printf("Account number: %d\n", account.account\_number);

printf("Account holder's name: %s\n", account.account\_name);

printf("Balance: %.2f\n", account.balance);

}

int compare\_account(struct bank\_account account1, struct bank\_account account2) {

if (account1.balance > account2.balance) {

return 1;

} else {

return 0;

}

}

#define N 10

int main()

{

struct bank\_account accounts[N];

int max\_index = 0;

for (int i = 0; i < N; i++) {

printf("Enter the details of account %d:\n", i + 1);

read\_account(&accounts[i]);

}

for (int i = 1; i < N; i++) {

if (compare\_account(accounts[i], accounts[max\_index])) {

max\_index = i;

}

}

printf("The account with the highest balance is:\n");

print\_account(accounts[max\_index]);

return 0;

}