**Introduction to C Programming**

Welcome to the introductory lecture on C programming. Since you already have experience with Java, we will focus on the differences and similarities between Java and C.

**1. Keywords**

Similar to Java, C has a set of reserved keywords that carry special meanings. Some of the common C keywords include:

* int – for the integer data type
* char – for the character data type
* if, else, while, for – control flow statements
* return – used to return a value from a function
* void – used for functions that do not return a value
* const – for declaring constant variables
* sizeof – used to determine the size of a data type or variable
* struct – used to define structures (similar to classes but without methods)

These keywords are reserved and cannot be used as identifiers, such as variable or function names, just like in Java.

**2. Data Types**

C has a simpler set of data types compared to Java. The basic data types in C include:

* **Integer Types**:
  + int: Stores whole numbers, typically 2 or 4 bytes, depending on the system.
  + short and long: Variants of integers that can store smaller or larger values.
* **Floating-Point Types**:
  + float: Stores single-precision decimal numbers.
  + double: Stores double-precision decimal numbers.
* **Character Type**:
  + char: Stores a single character (1 byte), represented by ASCII values.
* **Void Type**:
  + void: Represents the absence of a value, used mostly for functions that do not return anything.

**Example**:

int count = 5;

char grade = 'A';

float radius = 3.14;

double salary = 123456.789;

C lacks some of the higher-level types in Java, such as String and Boolean. Instead, C represents strings as arrays of characters and uses integer values (0 for false, non-zero for true) for Boolean logic.

**3. Conditionals**

C uses conditional statements similar to Java for decision-making. The primary control structures include:

* **if-else**: Works the same as in Java.

**Syntax**:

if (condition) {

// Code block executed if condition is true

} else {

// Code block executed if condition is false

}

**Example**:

int number = 10;

if (number > 0) {

printf("number is positive\n");

} else {

printf("number is not positive\n");

}

* **switch-case**: Also similar to Java but requires a break statement after each case to prevent fall-through.

**Syntax**:

switch (variable) {

case value1:

// Code block for value1

break;

case value2:

// Code block for value2

break;

default:

// Code block if no cases match

}

**Example**:

int dayOfWeek = 3;

switch (dayOfWeek) {

case 1:

printf("Monday\n");

break;

case 3:

printf("Wednesday\n");

break;

default:

printf("Invalid day\n");

}

**4. Loops**

C provides three main looping constructs:

* **while**: Works like Java's while loop.

**Syntax**:

while (condition) {

// Code block executed repeatedly while the condition is true

}

**Example**:

int index = 0;

while (index < 5) {

printf("Current index: %d\n", index);

index++;

}

* **for**: Similar to Java’s for loop.

**Syntax**:

for (initialization; condition; increment) {

// Code block executed for each iteration

}

**Example**:

for (int step = 0; step < 5; step++) {

printf("Step: %d\n", step);

}

* **do-while**: Executes the code block at least once before checking the condition, similar to Java.

**Syntax**:

do {

// Code block executed

} while (condition);

**Example**:

int counter = 0;

do {

printf("Counter: %d\n", counter);

counter++;

} while (counter < 5);

**5. Functions**

In C, functions are similar to methods in Java but without the concept of classes. Functions in C can return values, and they must be declared before use, unless the function prototype is declared before the main function.

**Syntax**:

return\_type function\_name(parameters) {

// Function body

return value; // Optional if return type is not void

}

**Example**:

int addNumbers(int firstNumber, int secondNumber) {

return firstNumber + secondNumber;

}

int main() {

int result = addNumbers(5, 10);

printf("Sum: %d\n", result);

return 0;

}

Functions in C can return basic data types (like int, float, etc.) or void if they don’t return anything.

**6. One-Dimensional Arrays**

In C, arrays are declared differently compared to Java, and the size of the array is fixed once declared.

**Syntax**:

data\_type array\_name[array\_size];

**Example (integer array)**:

int scores[5] = {85, 90, 78, 92, 88};

You can access array elements using their indices, starting from 0, just like in Java.

**Example (character array)**:

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'}; // '\0' denotes the null terminator in C strings

A char array can be used to represent strings in C, with the null terminator '\0' marking the end of the string.

**Accessing Array Elements**:

int scores[5] = {85, 90, 78, 92, 88};

printf("First score: %d\n", scores[0]); // Outputs 85

printf("Last score: %d\n", scores[4]); // Outputs 88

**What Happens When You Index Outside the Bounds of an Array in C**

In C, arrays do not perform bounds checking, which means accessing elements outside the valid range of an array does not generate an immediate error. This can lead to unexpected behaviors, including the corruption of nearby variables or even crashes. Let's explore what happens when you index outside the bounds of an array and how it affects memory and other variables.

**1. How Arrays Work in C**

In C, arrays are stored in **contiguous memory locations**. When you declare an array, C allocates a block of memory large enough to store all its elements. However, C does **not** automatically prevent you from accessing memory outside this allocated block.

Consider the following array:

int numbers[3] = {10, 20, 30};

The memory layout looks like this:

| **Memory Address** | **Value** |
| --- | --- |
| numbers[0] | 10 |
| numbers[1] | 20 |
| numbers[2] | 30 |

If you try to access an element like numbers[3], you are attempting to access memory that is beyond the allocated array. Since C does not check whether the index is within bounds, the program will still compile and run but could lead to unpredictable results.

**2. Accessing Out-of-Bounds Elements**

When you access an element outside the bounds of an array, the program may access **random memory** that could belong to other variables or parts of the program. This is known as **undefined behavior**.

Let’s see an example:

#include <stdio.h>

int main() {

int numbers[3] = {10, 20, 30};

int otherVariable = 100;

printf("numbers[0]: %d\n", numbers[0]);

printf("numbers[1]: %d\n", numbers[1]);

printf("numbers[2]: %d\n", numbers[2]);

// Accessing out of bounds

printf("numbers[3]: %d\n", numbers[3]); // Undefined behavior

printf("otherVariable: %d\n", otherVariable);

return 0;

}

Output:

numbers[0]: 10

numbers[1]: 20

numbers[2]: 30

numbers[3]: -1138550688 // Random garbage value

otherVariable: 100

In the above example, numbers[3] does not exist, but the program still accesses some memory location, producing a **random garbage value**. The value depends on whatever happens to be stored in that memory at runtime.

**3. How Out-of-Bounds Access Can Corrupt Variables**

Accessing memory outside of an array can also affect other variables that are stored near the array in memory. In C, local variables are often stored next to each other in memory. Accessing beyond the bounds of an array can overwrite these nearby variables.

Consider this modified example:

#include <stdio.h>

int main() {

int otherVariable1 = 100;

int numbers[3] = { 10, 20, 30 };

int otherVariable2 = 100;

printf("Before:\n");

printf("numbers[0]: %d\n", numbers[0]);

printf("numbers[1]: %d\n", numbers[1]);

printf("numbers[2]: %d\n", numbers[2]);

printf("otherVariable1: %d\n", otherVariable1);

printf("otherVariable2: %d\n", otherVariable2);

int num = 3;

// Writing out of bounds

for (int i = 0; i < 100; i++)

{

// This does not exist, overwriting something else!

numbers[i] = 999;

}

printf("\nAfter accessing out of bounds:\n");

printf("numbers[0]: %d\n", numbers[0]);

printf("numbers[1]: %d\n", numbers[1]);

printf("numbers[2]: %d\n", numbers[2]);

// Could be corrupted

printf("otherVariable1: %d\n", otherVariable1);

printf("otherVariable2: %d\n", otherVariable2);

return 0;

}makefile

Before:

numbers[0]: 10

numbers[1]: 20

numbers[2]: 30

otherVariable: 100

After accessing out of bounds:

numbers[0]: 10

numbers[1]: 20

numbers[2]: 30

otherVariable: 999 // otherVariable is corrupted

In this case, the program writes 999 to numbers[3], which does not exist. However, since otherVariable is located right after the numbers array in memory, writing to numbers[3] **overwrites** the value of otherVariable. This causes otherVariable to be incorrectly updated to 999.

**4. Why Does This Happen?**

C is a low-level language that gives you direct control over memory. The **lack of bounds checking** is a performance feature; it prevents the overhead of constantly checking whether array accesses are valid. However, this comes at the cost of **safety**.

When you access an index outside the array’s bounds, the program will simply treat it as a normal memory location, not knowing (or caring) whether it was part of the array or not. This can lead to:

* **Garbage values**: Reading from uninitialized or out-of-bounds memory often returns random data.
* **Variable corruption**: Writing to out-of-bounds memory can overwrite other variables or even critical data, causing bugs or crashes.
* **Program crashes**: Sometimes, out-of-bounds accesses touch memory that the operating system has marked as protected, causing a segmentation fault (program crash).

**5. Real-World Implications**

* **Security Vulnerabilities**: Out-of-bounds access can lead to **buffer overflows**, a common security issue. Attackers can exploit these to inject malicious code or crash the program.
* **Program Instability**: Indexing outside the bounds can introduce hard-to-diagnose bugs, causing crashes or corrupting data.

**6. Avoiding Out-of-Bounds Errors**

To avoid out-of-bounds errors, you should always check that your indices are within the valid range for the array. This can be done by using conditional checks before accessing or modifying array elements.

**Example**:

#include <stdio.h>

int main() {

int numbers[3] = {10, 20, 30};

int index = 4;

if (index >= 0 && index < 3) {

printf("numbers[%d]: %d\n", index, numbers[index]);

} else {

printf("Error: Index out of bounds\n");

}

return 0;

}

Output:

Error: Index out of bounds

In this example, the code checks whether index is within the valid range (0 to 2, since the array has 3 elements) before accessing the array. If it’s out of bounds, the program prints an error message instead of accessing invalid memory.

**Conclusion**

In C, accessing or modifying array elements outside their bounds leads to **undefined behavior**, which can manifest in various ways, such as:

* Returning garbage values
* Overwriting nearby variables
* Crashing the program

Because C does not provide built-in bounds checking, it is up to the programmer to ensure array accesses are valid. This can be done by explicitly checking array indices, which helps avoid data corruption and other unpredictable consequences.

**Summary**

* **Keywords**: C uses similar keywords to Java, but some are unique to C.
* **Data Types**: C has fewer, simpler data types like int, char, float, and double.
* **Conditionals**: The if-else and switch statements in C work the same as in Java.
* **Loops**: C provides while, for, and do-while loops, similar to Java.
* **Functions**: C functions are like Java methods but without classes.
* **Arrays**: One-dimensional arrays in C are statically defined, and character arrays are used to represent strings.
* **Out of bounds in Arrays**: When you access an element outside the bounds of an array