Homework Assignment 4

Junesh Gautam

11/7/2024

Professor: Zainab Albujasim, Ph.D.

Review Questions

Question 5: Define ordinal, enumeration, and subrange types.

- **Ordinal Types:** An ordinal type is a data type in which the values are ordered and can be counted. Examples include integer and character types.
- **Enumeration Types:** Enumeration types define a finite set of named values, often representing a fixed set of states or categories. For instance, `enum Day { Mon, Tue, Wed };` in C++ defines days of the week.
- **Subrange Types:** A subrange type is a contiguous subset of a base type. For example, in Pascal, `1..10` is a subrange of integers.

Question 9: Define static, fixed stack-dynamic, stack-dynamic, fixed heap-dynamic, and heap-dynamic arrays. What are the advantages of each?

- **Static Arrays:** Arrays with fixed size and memory location throughout the program. *Advantage:* Fast access due to no dynamic memory allocation.
- **Fixed Stack-Dynamic Arrays:** Size is fixed at declaration but memory is allocated at runtime on the stack. *Advantage:* Allows for flexibility within stack memory constraints.
- **Stack-Dynamic Arrays:** Size and memory allocation occur at runtime based on stack usage. *Advantage:* Allows flexible array sizes in function calls.
- **Fixed Heap-Dynamic Arrays:** Size is fixed at allocation but stored on the heap. *Advantage:* Persistent across function calls and provides larger memory capacity.
- **Heap-Dynamic Arrays:** Size and storage are dynamically managed on the heap. *Advantage:* Provides maximum flexibility in size and memory management.

Question 12: What languages support negative subscripts?

Languages like **Python** and **Ruby** support negative subscripts. In Python, for example, a list can be accessed from the end with negative indices (`list[-1]` for the last element).

Question 38: What is a C++ reference type, and what is its common use?

A **C++ reference type** is an alias for an existing variable. Once assigned, it cannot be reassigned to another variable. Commonly used to pass variables by reference in functions, allowing modifications without copying data.

Question 51: What is structure type equivalence?

Structure type equivalence occurs when two types are equivalent based on their structures rather than their names. For example, if two types have the same fields in the same order, they are equivalent.

Question 52: What is the primary advantage of name type equivalence?

The primary advantage of **name type equivalence** is increased type safety, as types must explicitly match by name, reducing accidental misuse of similar structures.

Problem Set

Problem 6: Explain all of the differences between Ada's subtypes and derived types.

In Ada:

- **Subtypes:** Are constrained versions of an existing type but do not create a new type. For example, a subtype might restrict a base integer type to a specific range. Subtypes retain all operations of the base type.
- **Derived Types:** Define a new type based on an existing one, with no implicit conversions. Derived types allow for overloading operators or defining new ones.

Key Difference: Subtypes provide constraints, while derived types provide a new type with additional flexibility in operator redefinition.

Problem 14: Write a short discussion of what was lost and what was gained in Java's designers' decision to not include the pointers of C++.

In Java:

- **Gains:** Eliminating pointers simplifies the language, improves security, and reduces risks of memory leaks and pointer-related errors. It also allows for efficient garbage collection.
- **Losses:** The absence of pointers limits low-level memory manipulation and fine-grained control, which can impact performance in systems programming and real-time applications.

Problem 15: What are the arguments for and against Java's implicit heap storage recovery, when compared with the explicit heap storage recovery required in C++? Consider real-time systems.

- **Arguments For Implicit Heap Storage Recovery:**
  - **Simplicity:** Programmers don't need to manually manage memory, reducing the risk of memory leaks.
  - **Safety:** Automated garbage collection prevents issues with dangling pointers and memory corruption.
- **Arguments Against Implicit Heap Storage Recovery:**
  - **Real-Time Constraints:** Garbage collection can introduce unpredictable delays, which may not be suitable for real-time systems.
  - **Lack of Control:** C++ allows developers to control when and how memory is freed, offering potential performance optimizations.

Programming Exercise

Problem 1: Design a set of simple test programs to determine the type compatibility rules of a C compiler to which you have access. Write a report of your findings.

To determine type compatibility rules of a C compiler, we are considering testing:

- Assignment compatibility between different data types, e.g., assigning 'float' values to 'int' variables.
- Implicit conversions, e.g., between 'int' and 'double' in arithmetic expressions.
- Compatibility between arrays and pointers, especially with 'const' types.

Finally, we will analyze whether implicit type casting is allowed or if explicit casting is required and then summarize our findings to understand the type system behavior and any inconsistencies in type compatibility.

Introduction

In this problem, we design several simple C programs to test the type compatibility rules of the GCC compiler. We aim to understand how the compiler handles assignments, implicit conversions, and compatibility between different data types.

Test Programs and Findings

1. Assignment Compatibility Between `float` and `int`

Test Code:

Listing 1: Assigning a `float` to an `int`

```c
#include <stdio.h>

int main() {
    float f = 3.14f;
    int i = f; // Assigning float to int
    printf("Float value: %f\n", f);
    printf("Integer value after assignment: %d\n", i);
    return 0;
}
```

Compilation and Output:
Compiled with `gcc -Wall` to enable all warnings.

```
warning: conversion from 'float' to 'int', possible loss of data [-Wconversion]
```

Program Output:

```
Float value: 3.140000
Integer value after assignment: 3
```

Analysis:
- The compiler allows assigning a `float` to an `int` with a warning about possible data loss. - The fractional part is truncated during the assignment.

2. Implicit Conversion in Arithmetic Expressions

Test Code:

Listing 2: Implicit conversion between `int` and `double`

```c
#include <stdio.h>

int main() {
    int i = 5;
    double d = 2.0;
    double result = i / d; // Implicit conversion of 'i' to double
    printf("Result of %d / %.1f = %.2f\n", i, d, result);
    return 0;
}
```

Compilation and Output:
No warnings or errors during compilation.
Program Output:

```
Result of 5 / 2.0 = 2.50
```

Analysis:
- The integer `i` is implicitly converted to `double` for the division operation. - The compiler handles the implicit conversion without warnings.

3. Compatibility Between Arrays and Pointers

Test Code:

Listing 3: Assigning an array to a pointer

```c
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr;
    ptr = arr; // Assigning array to pointer
    printf("First element via pointer: %d\n", *ptr);
    return 0;
}
```

Compilation and Output:
No warnings or errors during compilation.
Program Output:

```
First element via pointer: 1
```

Analysis:
- Arrays decay to pointers to their first element when assigned to a pointer variable. - The compiler allows this assignment without issues.

4. Assignment Compatibility Between `const` and Non-`const` Types

Test Code:

Listing 4: Assigning `const int*` to `int*`

```c
#include <stdio.h>

int main() {
    const int ci = 10;
    int *pi;
    pi = &ci; // Assigning address of const int to int*
    *pi = 20; // Attempt to modify const value
    printf("ci = %d\n", ci);
    return 0;
}
```

Compilation and Output:
Compiled with `gcc -Wall`:

```
warning: initialization discards 'const' qualifier from pointer target type [-Wdiscarded-qu
```

Program Output:

```
ci = 20
```

Analysis:
- The compiler warns about discarding the `const` qualifier. - Despite the warning, the program compiles, and we can modify the `const` variable through the pointer, which leads to undefined behavior. - This demonstrates that the compiler's type system can be circumvented, leading to potential safety issues.

5. Implicit Casting Between `void*` and Other Pointer Types

Test Code:

Listing 5: Assigning `void*` to `int*`

```c
#include <stdio.h>

int main() {
    void *vp;
    int i = 42;
    vp = &i; // Assigning address of int to void*
    int *ip = vp; // Assigning void* to int* without cast
    printf("Value via int pointer: %d\n", *ip);
    return 0;
}
```

Compilation and Output:
No warnings or errors during compilation.
Program Output:

```
Value via int pointer: 42
```

Analysis:
- The compiler allows implicit conversion from `void*` to other pointer types without a cast. - This shows that `void*` is a generic pointer type in C.

6. Assigning a `double` Value to an `int` Variable

Test Code:

Listing 6: Assigning `double` to `int`

```c
#include <stdio.h>

int main() {
    double d = 9.99;
    int i = d; // Assigning double to int
    printf("Double value: %f\n", d);
    printf("Integer value after assignment: %d\n", i);
    return 0;
}
```

Compilation and Output:
Compiled with `gcc -Wall`:

```
warning: conversion from 'double' to 'int', possible loss of data [-Wconversion]
```

Program Output:

```
Double value: 9.990000
Integer value after assignment: 9
```

Analysis:
- Similar to assigning `float` to `int`, the compiler allows it with a warning. - The fractional part is truncated, indicating possible data loss.

7. Implicit Conversion from `char` to `int`

Test Code:

Listing 7: Using `char` in arithmetic expressions

```c
#include <stdio.h>

int main() {
    char c = 'A';
    int i = c + 1; // Implicit conversion of char to int
```

```
    printf("Character: %c\n", c);
    printf("Integer value after addition: %d\n", i);
    printf("Corresponding character: %c\n", i);
    return 0;
}
```

Compilation and Output:
No warnings or errors during compilation.
Program Output:

```
Character: A
Integer value after addition: 66
Corresponding character: B
```

Analysis:
- The `char` type is implicitly converted to `int` in arithmetic expressions. - This behavior is allowed without warnings, demonstrating type compatibility between `char` and `int`.

8. Assigning an `int` Value to a `char` Variable

Test Code:

Listing 8: Assigning `int` to `char`

```c
#include <stdio.h>

int main() {
    int i = 1000;
    char c = i; // Assigning int to char
    printf("Integer value: %d\n", i);
    printf("Character after assignment: %c\n", c);
    printf("Character ASCII code: %d\n", c);
    return 0;
}
```

Compilation and Output:
Compiled with `gcc -Wall`:

```
warning: overflow in conversion from 'int' to 'char' changes value from '1000' to '-24' [-W
```

Program Output:

```
Integer value: 1000
Character after assignment:
Character ASCII code: -24
```

Analysis:
- Assigning a large `int` value to a `char` causes overflow. - The compiler warns about the overflow and the change in value. - This shows that implicit conversion can lead to data corruption.

9. Comparing Pointers of Different Types

Test Code:

Listing 9: Comparing pointers of different types

```c
#include <stdio.h>

int main() {
    int i = 10;
    float f = 3.14f;
    int *pi = &i;
    float *pf = &f;
```

```
    if (pi == (int *)pf) {
        printf("Pointers are equal.\n");
    } else {
        printf("Pointers are not equal.\n");
    }
    return 0;
}
```

Compilation and Output:
Compiled with `gcc -Wall`:

```
warning: comparison between distinct pointer types 'int *' and 'float *' [-Wpointer-compare
```

Program Output:

```
Pointers are not equal.
```

Analysis:
- The compiler warns about comparing pointers of different types. - Casting is required to suppress the warning, indicating type incompatibility.

10. Implicit Conversion from `int` to `float` in Function Calls

Test Code:

Listing 10: Passing `int` to a function expecting `float`

```c
#include <stdio.h>

void printFloat(float f) {
    printf("Float value: %f\n", f);
}

int main() {
    int i = 42;
    printFloat(i); // Implicit conversion from int to float
    return 0;
}
```

Compilation and Output:
No warnings or errors during compilation.
Program Output:

```
Float value: 42.000000
```

Analysis:
- The compiler allows implicit conversion from `int` to `float` when passing arguments to functions. - This demonstrates compatibility in function calls.

Summary of Findings

- The C compiler allows various implicit conversions between basic data types, often with warnings about possible data loss or overflow. - Assigning between floating-point and integer types truncates the fractional part, with warnings about data loss. - Implicit conversions in arithmetic expressions and function calls are generally allowed without warnings. - Assigning pointers between different types may generate warnings, emphasizing the need for explicit casting. - Assigning `const` variables to non-`const` pointers can lead to undefined behavior, highlighting potential safety issues. - The compiler's type system is flexible but relies on the programmer to handle potential issues arising from implicit conversions.

Conclusion

The GCC C compiler enforces type compatibility rules with a balance between flexibility and safety. While it allows many implicit conversions to facilitate programming convenience, it provides warnings to alert the programmer of potential issues. Understanding these rules is crucial for writing safe and reliable C programs.

Problem 5: Write a simple program in C++ to investigate the safety of its enumeration types. Include at least 10 different operations on enumeration types to determine what incorrect or just silly things are legal. Now, write a C# program that does the same things and run it to determine how many of the incorrect or silly things are legal. Compare your results.

Introduction

In this problem, we explore the safety and type-checking of enumeration types in **C++** and **C#**. We will:

- Write C++ and C# programs performing various operations on enums.
- Include at least 10 different operations, including incorrect or nonsensical ones.
- Compare the behavior of both languages regarding enum safety.

C++ Enumeration Tests

Test Code in C++

Listing 11: C++ Enumeration Tests

```cpp
#include <iostream>

enum Color { Red, Green, Blue };
enum TrafficLight { Stop = Red, Go = Green, Caution = Yellow }; // Yellow undefined
enum class Direction { North, East, South, West };

int main() {
    Color c = Red;

    // 1. Assign integer to enum variable
    c = (Color)1;

    // 2. Assign out-of-range integer to enum variable
    c = (Color)10;

    // 3. Implicit conversion from enum to int
    int n = c;

    // 4. Increment enum variable
    c = (Color)(c + 1);

    // 5. Compare enum with integer
    if (c == 2) {
        std::cout << "c equals 2\n";
    }

    // 6. Use enum variable in switch case
    switch (c) {
        case Red:
            std::cout << "Red\n";
            break;
        case Green:
            std::cout << "Green\n";
            break;
        case Blue:
            std::cout << "Blue\n";
            break;
        default:
```

```
            std::cout << "Unknown Color\n";
    }

    // 7. Assign enum of one type to enum of another type
    Direction d = (Direction)c;

    // 8. Use enum class without scoping
    // Direction dir = North; // Error: 'North' is not in scope

    // 9. Increment enum class variable
    Direction dir = Direction::North;
    dir = (Direction)((int)dir + 1); // Needs casting

    // 10. Bitwise operations on enum
    int flags = Red | Blue;

    std::cout << "Program completed.\n";
    return 0;
}
```

Compilation and Observations

Compiled with `g++ -std=c++11 -Wall`.

Observations:

1. **Assign integer to enum variable**: Allowed with explicit cast. 2. **Assign out-of-range integer**: Allowed with explicit cast, but may lead to invalid enum value. 3. **Implicit conversion from enum to int**: Allowed without cast. 4. **Increment enum variable**: Allowed with casting. 5. **Compare enum with integer**: Allowed without cast. 6. **Switch statement with enum**: Works as expected. 7. **Assign enum of one type to another**: Allowed with casting, but unsafe. 8. **Use enum class without scoping**: Not allowed; causes compile-time error. 9. **Increment enum class variable**: Requires casting to int and back. 10. **Bitwise operations on enum**: Allowed, as underlying type is int.

Analysis

- **Type Safety**: C++ traditional enums are not strongly typed. - **Implicit Conversions**: Enums can be implicitly converted to integers. - **Invalid Values**: Enums can hold values outside their defined range when cast from integers. - **Enum Classes**: Provide better type safety but require explicit casting for arithmetic operations.

C# Enumeration Tests

Test Code in C#

Listing 12: C# Enumeration Tests

```
using System;

enum Color { Red, Green, Blue }
enum TrafficLight { Stop = Color.Red, Go = Color.Green, Caution = Color.Yellow } // Error: 'Color
    .Yellow' does not exist
enum Direction { North, East, South, West }

class Program
{
    static void Main()
    {
        Color c = Color.Red;

        // 1. Assign integer to enum variable
        c = (Color)1;

        // 2. Assign out-of-range integer to enum variable
```

```
        c = (Color)10;

        // 3. Implicit conversion from enum to int
        int n = (int)c;

        // 4. Increment enum variable
        c++; // Allowed

        // 5. Compare enum with integer
        if ((int)c == 2)
        {
            Console.WriteLine("c equals 2");
        }

        // 6. Use enum variable in switch case
        switch (c)
        {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown Color");
                break;
        }

        // 7. Assign enum of one type to enum of another type
        // Direction d = (Direction)c; // Error: Cannot convert type

        // 8. Use enum without scoping
        Direction dir = North; // Error: 'North' does not exist in the current context

        // 9. Increment enum variable
        dir = dir + 1; // Allowed with warning

        // 10. Bitwise operations on enum
        Color flags = Color.Red | Color.Blue;

        Console.WriteLine("Program completed.");
    }
}
```

Compilation and Observations

Compiled with `csc` (C# compiler).

Observations:

1. **Assign integer to enum variable**: Allowed with explicit cast. 2. **Assign out-of-range integer**: Allowed with explicit cast, but may result in invalid enum value. 3. **Implicit conversion from enum to int**: Requires explicit cast. 4. **Increment enum variable**: Allowed. 5. **Compare enum with integer**: Requires casting enum to int. 6. **Switch statement with enum**: Works as expected. 7. **Assign enum of one type to another**: Not allowed, even with casting. 8. **Use enum without scoping**: Not allowed; causes compile-time error. 9. **Increment enum variable**: Allowed but may generate a warning. 10. **Bitwise operations on enum**: Allowed if the enum has the '[Flags]' attribute or by casting.

Analysis

- **Type Safety**: C# enums are strongly typed. - **Implicit Conversions**: Enums do not implicitly convert to integers. - **Invalid Values**: Enums can hold invalid values when cast from integers. - **As-

signment Restrictions**: Cannot assign between different enum types, even with casting.

Comparison of Results

Operation	C++ Behavior	C# Behavior
Assign integer to enum variable	Allowed with cast	Allowed with cast
Assign out-of-range integer	Allowed with cast	Allowed with cast
Implicit conversion from enum to int	Allowed without cast	Requires explicit cast
Increment enum variable	Allowed with cast	Allowed
Compare enum with integer	Allowed without cast	Requires casting enum to int
Use enum in switch case	Works as expected	Works as expected
Assign enum of one type to another	Allowed with cast	Not allowed
Use enum without scoping	Not allowed for enum class	Not allowed
Increment enum class variable	Requires casting	N/A
Bitwise operations on enum	Allowed	Allowed with considerations

Overall Findings

- **Type Safety**: C# enforces stronger type safety with enums compared to C++. - **Implicit Conversions**: C++ allows more implicit conversions, which can lead to errors. - **Assignment Restrictions**: Assigning between different enum types is restricted in C#. - **Scoping**: Both languages require proper scoping for enum usage. - **Operations**: Incrementing enums is allowed in both, but may lead to invalid values.

Conclusion

Our investigation reveals that while both C++ and C# allow certain operations on enums, C# provides better type safety and stricter rules to prevent misuse. C++ offers more flexibility at the expense of potential type-related bugs. Programmers should be cautious when performing operations on enums, especially in C++, to avoid unintended behavior.