Operator Overloading

```
#include <vlib/vlib.h>
typedef struct {
  u8 red;
  u8 green;
  u8 blue;
} color_t;
static int
color_cmp (color_t *a, color_t *b) {
  // Compare color components lexicographically
  if (a->red != b->red) {
    return a->red > b->red ? 1 : -1;
  } else if (a->green != b->green) {
    return a->green > b->green ? 1 : -1;
  } else {
    return a->blue > b->blue ? 1 : -1;
  }
}
static uword
unimplemented (vlib_main_t * vm, vlib_node_runtime_t * node, vlib_frame_t * from) {
  clib_warning("Operator overloading not directly supported in VPP");
  return 0;
}
VLIB_API_REGISTRATION_HANDLER(vlibapi_color_cmp, {
  .type = REGISTRATION_TYPE_APP,
  .data = NULL,
  .handler = unimplemented,
});
```

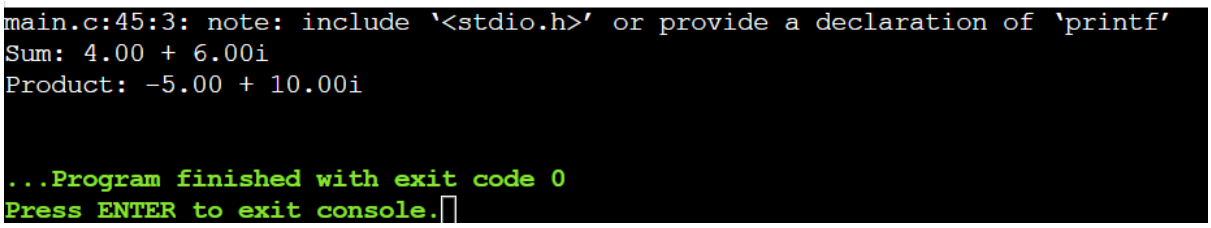
Output

```
main.cpp:9:10: fatal error: vlib/vlib.h: No such file or directory
    9 | #include <vlib/vlib.h>
      |          ^~~~~~~~~~~~~~
compilation terminated.
```

```
#include <math.h>

typedef struct {
  double real;
  double imag;
} complex_t;

complex_t *
complex_add(complex_t *a, complex_t *b) {
  complex_t *c = malloc(sizeof(complex_t));
  c->real = a->real + b->real;
  c->imag = a->imag + b->imag;
  return c;
}

complex_t *
complex_multiply(complex_t *a, complex_t *b) {
  complex_t *c = malloc(sizeof(complex_t));
  c->real = a->real * b->real - a->imag * b->imag;
  c->imag = a->real * b->imag + a->imag * b->real;
  return c;
}

void
complex_free(complex_t *c) {
  free(c);
}

// Usage Example
int main() {
  complex_t num1 = {1.0, 2.0};
  complex_t num2 = {3.0, 4.0};
  complex_t *result_add = complex_add(&num1, &num2);
```

```
  complex_t *result_mul = complex_multiply(&num1, &num2);
  printf("Sum: %.2f + %.2fi\n", result_add->real, result_add->imag);
  printf("Product: %.2f + %.2fi\n", result_mul->real, result_mul->imag);
  complex_free(result_add);
  complex_free(result_mul);
  return 0;
}
```

Output



Operator Overloading

1. What are the benefits and drawbacks of operator overloading?

Benefits:

1. **Improved Readability**:
    - **Intuitive Syntax**: Operator overloading can make code that uses user-defined types more intuitive
2. **Consistency with Built-in Types**:
    - **Uniform Interface**: Operator overloading allows user-defined types to be manipulated using the same syntax as built-in types. This provides a uniform interface and can reduce the learning curve for new users of the class.
3. **Enhanced Expressiveness**:
    - **Concise Code**: Code that uses operator overloading can be more concise. Complex operations can be expressed more clearly and succinctly, reducing boilerplate and improving maintainability.
4. **Encapsulation**:
    - **Abstract Operations**: Operator overloading allows you to encapsulate complex operations within a class, providing a clear abstraction layer and hiding the implementation details from the user.

Drawbacks:

1. **Potential for Misuse**:
    - **Unexpected Behavior**: If not used carefully, operator overloading can lead to code that is confusing and difficult to understand. Overloading operators in ways that deviate from their conventional meanings can result in unexpected behavior.
2. **Maintenance Challenges**:
    - **Code Complexity**: Overloaded operators can make the code more complex and harder to maintain, especially if the overloaded operators perform non-trivial operations. Developers maintaining the code need to understand the custom behavior of the operators.
3. **Debugging Difficulties**:
    - **Hidden Logic**: The logic behind overloaded operators can be hidden, making it harder to debug issues. Developers may need to dig into the implementation of the overloaded operators to understand the cause of a bug.
4. **Performance Overhead**:
    - **Indirect Costs**: Operator overloading can introduce performance overhead, especially if the overloaded operators involve complex computations or additional function calls.
5. **Limited to Specific Languages**:
    - **Language Support**: Not all programming languages support operator overloading. This means that code relying heavily on this feature may not be portable to other languages that do not support it.

2. Can you overload the assignment operator (=) in C++? If so, how would you ensure proper behavior?

Yes, you can overload the assignment operator (=) in C++. Overloading the assignment operator is common practice when dealing with user-defined types, especially if the class manages resources like dynamic memory, file handles, or other system resources that need proper management.

To ensure proper behavior when overloading the assignment operator, you should follow these guidelines:

1. **Check for self-assignment**: Ensure the operator handles the case where an object is assigned to itself.
2. **Release any resources**: Clean up any resources the object currently holds before assigning new values.
3. **Copy resources from the source object**: Properly copy the resources from the right-hand side (RHS) object.
4. **Return *this**: Return the current object to allow for chained assignment operations.

Aspect	Member function Overloading	Non-Member (Friend)Function Overloading
Definition Location	Inside the class	Outside the class
Syntax	ReturnType operator+(const ClassName& other) const	friend ReturnType operator+(const ClassName& 1hs, const ClassName& rhs)
Left Operand Type	Must be an instance of the class	Can be any type
Right Operand Type	Can be any type	Can be any type
Access to Private Members	Direct access without friend	Requires friend to access private members
Encapsulation	Better encapsulation, all logic inside the class	Less encapsulation, logic outside the class

Flexibility	Less flexible, left operand must be class instance	More flexible, both operands can be different types
Symmetry	Asymmetric, implicity receives this pointer	Symmetric, both operands treated equally
Usage	obj1+obj2 calls obj.operator +(obj2)	obj1+obj2 calls operator+(obj1, obj2)
Common use cases	When the left operand is always an instance of the class	When the operation involves different types or symmetry is desired

5. Is it possible to overload the comparison operators (==, !=, <, >, <=, >=) for custom classes? If so, what considerations should be taken into account?

Yes, it is possible to overload comparison operators (==, !=, <, >, <=, >=) for custom classes in C++. Overloading these operators allows you to define custom behavior for comparing objects of your classes, making your code more intuitive and expressive.

Considerations for Overloading Comparison Operators

1. **Consistency**:
    - Ensure that the overloaded operators maintain logical consistency. For instance, if a < b is true, then b > a should also be true. Similarly, if a == b is true, then a != b should be false.
2. **Symmetry and Transitivity**:
    - Symmetry: a == b should imply b == a.
    - Transitivity: If a == b and b == c, then a == c.
3. **Efficiency**:
    - Implement the operators efficiently, especially if they are frequently used. Avoid unnecessary computations and prefer inline functions if appropriate.
4. **Avoid Redundancy**:
    - Implement the primary comparison operators (== and <) and derive the others from them to avoid redundancy and maintain consistency.
5. **Return Type**:
    - The comparison operators should return a boolean value (bool).

8. Discuss the potential ambiguity that could arise when overloading the subscript operator ([]) for a class. How can this ambiguity be resolved?

Overloading the subscript operator [] in C++ allows objects of a class to be indexed in a manner similar to arrays or standard containers. While this can enhance usability and provide a more intuitive interface for accessing elements, it can also introduce potential ambiguities

Potential Ambiguity

1. **Const vs. Non-const Overloads**:
    - If both const and non-const versions of the subscript operator are provided, the compiler must choose the correct one based on the context. This can be ambiguous if the usage context isn't clear.
2. **Multiple Indices**:
    - If a class conceptually supports multi-dimensional indexing, overloading the subscript operator to handle multiple indices can be confusing. This is typically resolved by defining a single subscript operator and overloading additional operators (such as the function call operator ()) for multiple indices.
3. **Return Types**:
    - The subscript operator usually returns a reference to the element. However, if the return type is not correctly chosen, it can lead to unexpected behavior or inefficient code, especially if returning by value instead of by reference.

9. Can operator overloading be used to implement the concept of immutability (unchanging state) for a class? Explain your answer.

Operator overloading can play a role in implementing the concept of immutability in a class, but it is not the sole mechanism. Immutability means that once an instance of a class is created, its state cannot be changed. To achieve this, you typically use several techniques together, with operator overloading being one of them.

Techniques to Implement Immutability

1. **Const Member Functions**:
    - Ensure that all member functions, including operator overloads, do not modify the state of the object. This is done by declaring them as const.
2. **Private or Deleted Mutators**:
    - Make sure that no member functions (including setters) that can modify the state are accessible. These functions can be private or deleted.
3. **Const Data Members**:
    - Use const for data members to ensure their values cannot be modified after the object is constructed.

4. **Returning New Objects**:
    - When implementing operations that might traditionally modify the object (like addition or concatenation), return new objects with the result instead of modifying the current object.

10. When overloading operators, what are some best practices to ensure code clarity and maintainability?

When overloading operators in C++, following best practices ensures code clarity, maintainability, and avoids potential pitfalls.

1. Use Operators Consistently

- **Follow Standard Conventions**: Stick to expected behavior of operators. For example, + should denote addition, == should denote equality comparison, etc.
- **Avoid Surprises**: Ensure that overloaded operators behave similarly to built-in types where possible. Users should not be surprised by unexpected behavior.

2. Implement All Related Operators

- **Provide Complete Set**: When overloading binary operators (+, -, *, /, etc.), ensure to overload related operators (+=, -=, *=, /=, etc.) for consistency.
- **Logical Operators**: Overload &&, ||, ! together for logical operations if applicable.

3. Prefer Member Functions for Binary Operators

- **Member vs. Non-member**: Prefer defining binary operators as member functions for consistency and access to private members, unless a non-member function is more appropriate (e.g., symmetry with different types).

4. Use const Correctness

- **Const Member Functions**: Declare member functions that do not modify the object's state as const.
- **Const Parameters**: Use const references for parameters whenever possible to avoid unnecessary copies and indicate immutability.

5. Return Types and Efficiency

- **Return Types**: Return by reference where appropriate to avoid unnecessary copying. Return by value when creating new objects or when returning primitives.
- **Efficiency**: Ensure that overloaded operators are efficient, especially for commonly used operations.

6. Document Intent and Edge Cases

- **Document Assumptions**: Clearly document the behavior of custom operators, especially if deviating from standard usage.

- **Handle Edge Cases**: Consider edge cases (like empty containers, null pointers, etc.) and ensure operators handle them gracefully.

7. Consider Global Function Approach

- **Friend Functions**: Use friend functions for operators that require access to private members but cannot be implemented as member functions (e.g., asymmetric operators).

Function Overloading

1. What is the core concept behind function overloading?

Function overloading is a core concept in programming languages like C++ and Java, where multiple functions can have the same name but differ in the type or number of parameters they accept. The core concept behind function overloading revolves around providing multiple implementations of a function with the same name but different parameter lists, allowing flexibility and convenience in function calls.

Key Aspects of Function Overloading:

1. **Same Function Name**:
    - Functions that perform similar tasks or operations can be given the same name.
2. **Different Parameter Lists**:
    - Overloaded functions must differ in the types of their parameters, their number, or both. This allows the compiler to distinguish between different versions of the function based on the arguments passed during a function call.
3. **Compile-time Resolution**:
    - The appropriate function to execute is determined at compile-time based on the arguments provided. This is also known as static or early binding.

2. How does the compiler differentiate between overloaded functions with the same name?

The compiler differentiates between overloaded functions with the same name primarily based on the number and types of parameters they accept.

Function Signature

1. **Parameter Types and Number**:
    - When you overload a function, each version must have a unique function signature, which includes the types and the number of parameters.

2. Compile-Time Resolution

- **Static Binding**: The determination of which overloaded function to call is done by the compiler at compile-time based on the arguments provided during the function call.

3. Considerations

- **Ambiguity**: If two overloaded functions have the same parameter types and number, the compiler cannot resolve which function to call, leading to a compilation error.
- **Implicit Conversions**: The compiler can perform implicit conversions (like int to double) to match function parameters if an exact match isn't found but an implicit conversion exists.

3. Can functions with different return types be overloaded? Explain your reasoning.

No, functions with different return types cannot be overloaded in C++. Function overloading is based solely on the function signature, which includes the function name and its parameter list. The return type alone does not form part of the function signature for the purpose of overloading.

Reasoning:

1. **Function Signature for Overloading**:
    - In C++, function overloading is resolved at compile-time based on the function name and its parameter types. This means that two functions with the same name and parameter types, but different return types, would result in a compilation error because the compiler cannot differentiate between them based on the return type alone.
2. **Return Type Ambiguity**:
    - If the compiler allowed functions with the same name and parameter list but different return types, it would lead to ambiguity.
3. **Overload Resolution**:

- Function overloading is meant to provide multiple implementations of the same function name for different parameter types or numbers. The return type does not participate in overload resolution because it does not affect how functions are called or distinguished based on arguments.

4. **Compile-Time Error:**

- The code above would result in a compilation error because the compiler cannot determine which add function to call based on the return type.

4.Design a function printValue that can handle different data types (e.g., int, double, std::string) by overloading it with appropriate parameter lists.

To design a function printValue that can handle different data types (int, double, std::string), we can use function overloading. Each overload of printValue will take a parameter of a specific type and print it accordingly. Here's how you can implement it:

```
#include <iostream>

#include <string>

Using namespace std;

// Overloaded function to print an integer

void printValue(int value) {

   cout << "Integer value: " << value << std::endl;

}

// Overloaded function to print a double

void printValue(double value) {

   cout << "Double value: " << value << std::endl;

}

// Overloaded function to print a string

void printValue(const std::string& value) {

   cout << "String value: " << value << std::endl;

}

int main() {

   int intValue = 5;

   double doubleValue = 3.14;

   string stringValue = "Hello, World!";

   // Calling different overloads of printValue

   printValue(intValue);
```

```
    printValue(doubleValue);

    printValue(stringValue);

    return 0;

}
```

5. Discuss the advantages and disadvantages of using default arguments in overloaded functions.

Advantages:

1. **Simplifies Function Calls**:
    - Default arguments allow you to define a single function signature that can handle multiple scenarios without requiring explicit arguments for each call. This simplifies function calls and reduces the need for overloaded versions with different parameter lists.
2. **Reduces Code Redundancy**:
    - Instead of defining multiple overloaded functions with similar functionality but different parameters, default arguments consolidate logic into a single function definition. This reduces code redundancy and makes the codebase more maintainable.
3. **Enhances Readability**:
    - Functions with default arguments are often more readable because they provide a clear indication of what parameters are optional and their default values. This improves code comprehension, especially when the function is used frequently.
4. **Maintains Backward Compatibility**:
    - When adding new parameters to existing functions, default arguments allow you to maintain backward compatibility with existing code that may not explicitly provide values for the new parameters. This helps in evolving APIs without breaking existing functionality.

Disadvantages:

1. **Ambiguity in Function Overloading**:
    - Default arguments can sometimes lead to ambiguity when combined with overloaded functions that differ only in default values. If multiple overloaded functions have similar signatures but differ in default arguments, the compiler may not be able to resolve which function to call in certain contexts.
2. **Hidden Dependencies**:
    - Default arguments can introduce hidden dependencies in functions. Developers relying on default values may not be aware of potential changes in behavior if defaults are modified in future versions of the function.
3. **Debugging Complexity**:

- Debugging functions with default arguments can be more complex. When unexpected behavior occurs, determining which set of default arguments were used (or overridden) during the function call may require careful inspection.

4. **Maintainability Challenges**:
   - Overuse of default arguments can make code less maintainable over time. When multiple parameters have defaults, understanding the full behavior of the function and its potential variations becomes more challenging.

6. In the context of function overloading, explain the concept of argument promotion and implicit type conversion.

In the context of function overloading, argument promotion and implicit type conversion play crucial roles in determining which overloaded function is called when a function is invoked with arguments of different types.

Argument Promotion:

Argument promotion refers to the automatic conversion of a function argument to a wider or more general data type if the function does not have an exact match for the provided argument type. This process typically occurs during function resolution in C++ and other languages with similar type promotion rules.

Implicit Type Conversion:

Implicit type conversion, also known as type coercion, refers to the automatic conversion of a data type to another data type without the need for explicit instructions from the programmer. This is done by the compiler to make compatible types match in expressions or function calls.

How They Affect Function Overloading:

1. **Argument Promotion**:
   - When selecting which overloaded function to call, the compiler may promote arguments to a wider type if an exact match is not found.
   - For example, if a function is overloaded to accept both int and double parameters, and you pass an int to a function expecting a double, the int will be promoted to a double.
2. **Implicit Type Conversion**:
   - If an exact match for the argument type is not found among the overloaded functions, the compiler may attempt to perform implicit type conversions to match the argument to one of the available function overloads.
   - For instance, if you have overloaded functions that accept int and double, and you pass a float, the float may be implicitly converted to either int or double, depending on the context and the conversions defined.

7.When might it be a better idea to use separate functions with descriptive names instead of overloading a single function?

Using separate functions with descriptive names instead of overloading a single function can be advantageous in several scenarios where clarity, maintainability, and readability of the code are paramount:

1. Distinct Functionality:

When the functions perform fundamentally different tasks or operations, even if they take similar types of arguments, using separate function names makes the code more self-documenting. This approach avoids confusion about the purpose of each function.

2. Different Algorithms:

If the functions employ different algorithms or computational approaches to achieve similar results, it's clearer to have separate function names. This prevents ambiguity and aids in understanding the underlying logic of each function.

3. Function Contracts:

When functions have significantly different preconditions or postconditions, using separate names clarifies the contract for each function. This helps users of the functions understand what is expected as inputs and what can be guaranteed as outputs.

4. Readability and Maintainability:

Separate function names enhance code readability by making it clear which function is being called at each invocation point. This aids in maintaining the codebase over time, as developers can quickly grasp the purpose and behavior of each function.

5. Avoiding Ambiguity and Confusion:

Overloading can lead to ambiguity or confusion if the function behavior is not clearly distinguished by the parameters alone. Separate function names eliminate this risk by explicitly naming each function based on its distinct purpose.

When to Consider Function Overloading Instead:

- **Homogeneous Operations**: When functions perform very similar operations on different types or configurations of data (like printValue for int, double, string), overloading can enhance code conciseness and maintainability.
- **Logical Variants**: When functions perform logically similar tasks with minor variations (like add(int, int) and add(double, double)), overloading can streamline the interface.

8.Can function overloading be used to achieve polymorphism (the ability to treat objects of different derived classes in a similar way)? Explain.

Function overloading in C++ does not directly achieve polymorphism in the same way that virtual functions and inheritance do. Polymorphism, particularly achieved through inheritance and virtual functions (runtime polymorphism), allows objects of different derived classes to be treated as objects of their base class, facilitating dynamic dispatch and runtime method resolution.

Function Overloading vs Polymorphism:

1. **Static Binding**:
    - Function overloading (also known as static polymorphism or compile-time polymorphism) is resolved at compile-time based on the function name and parameter types. The decision on which overloaded function to call is made by the compiler based on the arguments passed during the function call.
    - This does not change at runtime based on the actual type of the object being referenced or pointed to.
2. **Inheritance and Virtual Functions**:
    - Polymorphism achieved through inheritance and virtual functions (dynamic polymorphism or runtime polymorphism) allows different derived classes to override base class methods. Calls to these methods are resolved at runtime based on the actual type of the object pointed to or referenced, facilitating behavior that can vary based on the object's runtime type.

9.Describe a scenario where overloading a function with a variable number of arguments (varargs) could be beneficial.

Overloading a function with a variable number of arguments (varargs) can be beneficial in scenarios where the function needs to handle different numbers or types of arguments in a flexible and convenient manner. Here's a scenario where overloading a varargs function could be advantageous:

Logging Utility

Consider a logging utility that needs to log messages of varying complexity and data types

```
#include <iostream>

#include <sstream>

// Function to log messages with variable arguments using std::cout

void log(const char* message) {

   std::cout << "Log: " << message << std::endl;

}
```

```
// Overloaded function to log messages with variable arguments using std::cout

template<typename T, typename... Args>

void log(const T& firstArg, const Args&... args) {

   std::ostringstream oss;

   oss << firstArg;

   ((oss << ", " << args), ...); // C++17 fold expression to append each argument

   std::cout << "Log: " << oss.str() << std::endl;

}

int main() {

   log("Logging started.");  // Calls log(const char* message)

   log("Error", 404);       // Calls log(const T& firstArg, const Args&... args)

   log("User", "John", "logged in.");  // Calls log(const T& firstArg, const Args&... args)

   return 0;

}
```

Explanation:

- **Single Argument Version**: The log(const char* message) function is specialized to handle simple log messages specified as a C-style string.
- **Varargs Version**: The overloaded log(const T& firstArg, const Args&... args) function uses a variadic template parameter pack (Args...) to accept an arbitrary number of arguments of any type (T). It constructs a concatenated log message using std::ostringstream and prints it to std::cout.
- **Benefits**:
    - **Flexibility**: The varargs version of log() allows logging of messages with different numbers and types of arguments in a single function call, providing flexibility without needing to define multiple overloaded functions for specific argument combinations.
    - **Convenience**: Developers can use the same function name (log) for logging messages with varying levels of detail or complexity, simplifying the interface and enhancing code readability.
    - **Code Maintenance**: Centralizing logging functionality into overloaded varargs functions reduces redundancy and promotes code maintenance by encapsulating common logging logic.

Use Cases:

- **Debugging and Tracing**: Logging function calls with variable argument details during debugging sessions.
- **Event Logging**: Capturing and logging events with varying data associated with them, such as timestamps, user identifiers, or error codes.
- **Diagnostic Output**: Generating informative diagnostic messages with dynamic content based on the current state of the program or application.

10. Compare and contrast function overloading with virtual functions in C++ inheritance. Which approach is more suitable for specific use cases?

Function overloading and virtual functions in C++ inheritance serve different purposes and are suitable for different use cases based on the requirements of the program design. Here's a comparison and contrast between the two approaches:

Function Overloading:

1. **Purpose**:
    - **Static Polymorphism**: Function overloading allows multiple functions with the same name but different parameter lists to coexist within the same scope.
    - Resolved at compile-time based on the function name and the types and number of arguments.
2. **Usage**:
    - Useful when you want to provide multiple implementations of a function for different types or numbers of arguments.
    - Provides compile-time polymorphism where the compiler determines which function to call based on the static type of the arguments.
3. **Example**:

```
// Function overloading example

void print(int num);

void print(double num);

void print(const std::string& str);
```

1. **Advantages**:
    - Simplicity: Easy to understand and implement.
    - Efficiency: Resolved at compile-time, no runtime overhead.
2. **Disadvantages**:
    - Lack of runtime flexibility: Cannot dynamically change behavior based on the runtime type of objects.
    - Limited to the scope where the functions are defined.

Virtual Functions in C++ Inheritance:

1. **Purpose**:
   - **Dynamic Polymorphism**: Virtual functions allow a base class pointer or reference to invoke derived class methods at runtime, based on the actual object type being pointed to or referenced.
   - Resolved at runtime using vtables and virtual dispatch.
2. **Usage**:
   - Ideal for designing hierarchical class structures where base classes define interfaces and derived classes provide specific implementations.
   - Provides runtime flexibility to change behavior based on the runtime type of objects.
3. **Example**:

```
// Virtual function example

class Shape {

public:

   virtual void draw() const {

      // Base implementation

   }

   virtual ~Shape() {}

};

class Circle : public Shape {

public:

   void draw() const override {

      // Derived implementation

   }

};
```

1. **Advantages**:
   - Polymorphic behavior: Enables treating objects of derived classes through base class pointers or references, facilitating code extensibility and flexibility.
   - Runtime polymorphism: Allows dynamic method resolution based on object types at runtime.
2. **Disadvantages**:

- Overhead: Slight performance overhead due to virtual function dispatch mechanism.
- Complexity: Requires understanding of inheritance and virtual mechanisms, which may introduce complexity in design and implementation.

Suitability for Specific Use Cases:

- **Function Overloading**:
  - **Suitable Use Cases**: When you have a small number of related functions with different argument types or numbers, especially when the behavior is determined statically at compile-time.
  - **Example Use**: Utility functions like printing, basic arithmetic operations on different types.
- **Virtual Functions in Inheritance**:
  - **Suitable Use Cases**: When you need to model hierarchies of related classes where objects of derived classes need to be treated uniformly through base class pointers or references.
  - **Example Use**: Shape hierarchy with different drawing methods for each shape, where the drawing behavior depends on the specific type of shape object.

Programs

1.Complex Numbers (C++) - Define a class Complex to represent complex numbers with member variables for real and imaginary parts. Overload the +, -, and * operators for complex number addition, subtraction, and multiplication.

```
#include <iostream>

using namespace std;

class Complex {

private:

   double real;

   double imag;

public:

   // Constructor

   complex(double r = 0, double i = 0) : real(r), imag(i) {}


   // Overload the + operator

   Complex operator+(const Complex& other) const {

      return Complex(real + other.real, imag + other.imag);

   }

   // Overload the - operator

   Complex operator-(const Complex& other) const {

      return Complex(real - other.real, imag - other.imag);

   }

   // Overload the * operator
```

```
    Complex operator*(const Complex& other) const {
        return Complex(
            real * other.real - imag * other.imag,
            real * other.imag + imag * other.real
        );
    }
    // Friend function to overload << for printing
    friend ostream& operator<<(ostream& out, const Complex& c) {
        out << c.real << " + " << c.imag << "i";
        return out;
    }
};
int main() {
    Complex c1(3.0, 2.0);
    Complex c2(1.5, 4.5);
    Complex sum = c1 + c2;
    Complex diff = c1 - c2;
    Complex prod = c1 * c2;
    cout << "c1: " << c1 << std::endl;
    cout << "c2: " << c2 << std::endl;
    cout << "Sum: " << sum << std::endl;
    cout << "Difference: " << diff << std::endl;
    cout << "Product: " << prod << std::endl;
    return 0;
```

}

2. Point2D (Java) - Create a class Point2D with x and y coordinates. Overload the + operator to return a new Point2D object representing the sum of two points.

```
#include <iostream>

using namespace std;

class Point2D {

private:

int x

int y;

public:

Point2D(int xCoord = 0, int yCoord = 0) : x(xCoord), y(yCoord) {}

Point2D operator+(const Point2D& other) const {

return Point2D(x + other.x, y + other.y);  }  // Overloading + operator for addition of two points

void display() const {

cout << "(" << x << ", " << y << ")" << endl; } // Display function to print the coordinates of the point

};

int main() {

Point2D p1(2, 3);

Point2D p2(1, -1);

Point2D sum = p1 + p2;

cout << "Point 1: ";

p1.display();
```

```
cout << "Point 2: ";

p2.display();

cout << "Sum of Points: ";

sum.display();

return 0;

}
```

3. **Time (C++) - Design a class Time to store hours, minutes, and seconds. Overload the + operator to add two Time objects and return a new Time object with the combined duration.**

```
#include <iostream>

using namespace std;

class Time {

private:

   int hours;

   int minutes;

   int seconds;

public:

   Time(int h = 0, int m = 0, int s = 0) : hours(h), minutes(m), seconds(s) {}

   Time operator+(const Time& other) const {

      int totalSeconds = seconds + other.seconds;  // Overloading + operator

      int carryMinutes = totalSeconds / 60;

      totalSeconds %= 60;

      int totalMinutes = minutes + other.minutes + carryMinutes;

      int carryHours = totalMinutes / 60;

      totalMinutes %= 60;
```

```
        int totalHours = hours + other.hours + carryHours;

        return Time(totalHours, totalMinutes, totalSeconds);

    }

    void display() const {

        cout << hours << " hours, " << minutes << " minutes, " << seconds << " seconds"
<< endl;

    } };

int main() {

    Time t1(2, 30, 45);

    Time t2(1, 15, 20);

    Time sum = t1 + t2;

    cout << "Time 1: ";

    t1.display();

    cout << "Time 2: ";

    t2.display();

    cout << "Sum of Times: ";

    sum.display();

    return 0;

}
```

4. **Date (C#) - Implement a class Date with year, month, and day. Overload the comparison operators (== and !=) to compare two Date objects.**

```
#include <iostream>

using namespace std;

class Date {

private:

   int year;

   int month;

   int day;

public:

   Date(int y, int m, int d) : year(y), month(m), day(d) {}

   bool operator ==(const Date& other) {

      return year == other.year && month == other.month && day == other.day;

   }

   bool operator !=(const Date& other) {

      return !(*this == other);

   }

};


int main() {

   Date date1(2024, 6, 27);

   Date date2(2024, 6, 27);

   Date date3(2024, 6, 28);

   cout << boolalpha; // Print bools as true/false

   cout << "Date 1 == Date 2: " << (date1 == date2) << endl; // Output: true
```

```
    cout << "Date 1 != Date 3: " << (date1 != date3) << endl; // Output: true

    return 0;

}
```

5. **String Equality (C++) - Overload the equality operator (==) for a custom String class to compare string contents (not just memory addresses).**

```
#include <iostream>

#include <cstring>

using namespace std;

class MyString {

private:

    char* str;

public:

    MyString(const char* s) {

        str = new char[strlen(s) + 1];

        strcpy(str, s);

    }

    bool operator==(const MyString& other) {

        return strcmp(str, other.str) == 0;

    }


    // Destructor to free dynamically allocated memory

    ~MyString() {

        delete[] str;

    }

};
```

```
int main() {
    MyString str1("Hello");
    MyString str2("Hello");
    MyString str3("World");
    cout << boolalpha;
    cout << "str1 == str2: " << (str1 == str2) << endl; // Output: true
    cout << "str1 == str3: " << (str1 == str3) << endl; // Output: false
    return 0;
}
```

Function Overloading

1.Area Calculation (Java) - Create a function calculateArea that can handle different shapes (e.g., rectangle, circle) by overloading it with parameters like width, height, or radius.

```
#include <iostream>
using namespace std;
const double PI = 3.14159;
// Function to calculate area of a rectangle
double calculateArea(double width, double height) {
    return width * height;
}

// Function to calculate area of a circle
double calculateArea(double radius) {
```

```
    return PI * radius * radius;
}
int main() {
    cout << "Area of rectangle (5x3): " << calculateArea(5.0, 3.0) << endl;
    cout << "Area of circle (radius 2): " << calculateArea(2.0) << endl;
    return 0;
}
```

2.Unit Conversion (Python) - Design a function convert that takes a value and a unit (e.g., meters, feet, Celsius, Fahrenheit) and converts it to another unit using appropriate conversion factors.

```
#include <iostream>
using namespace std;
// Function to convert units
double convert(double value, const string& from_unit, const string& to_unit) {
    if (from_unit == "meters" && to_unit == "feet") {
        return value * 3.281;  // Conversion factor from meters to feet
    } else if (from_unit == "feet" && to_unit == "meters") {
        return value / 3.281;  // Conversion factor from feet to meters
    } else if (from_unit == "Celsius" && to_unit == "Fahrenheit") {
        return (value * 9/5) + 32;  // Celsius to Fahrenheit
    } else if (from_unit == "Fahrenheit" && to_unit == "Celsius") {
        return (value - 32) * 5/9;  // Fahrenheit to Celsius
    } else {
        cout << "Unsupported conversion" << endl;
        return 0.0;
```

```
    }

}

int main() {

    cout << "10 meters to feet: " << convert(10, "meters", "feet") << endl;  // Output: 32.81
feet

    cout << "32 Fahrenheit to Celsius: " << convert(32, "Fahrenheit", "Celsius") << endl;  //
Output: 0 Celsius

    return 0;

}
```

3.Statistics (C++) - Implement functions average, minimum, and maximum that can take an array of integers or doubles as input, depending on the function call.

```
#include <iostream>

#include <algorithm>

using namespace std;

// Function to calculate average of an array of integers

double average(int arr[], int size) {

    int sum = 0;

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

        sum += arr[i];

    }

    return static_cast<double>(sum) / size;

}

// Function to find minimum value in an array of doubles

double minimum(double arr[], int size) {

    double min_val = arr[0];

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

        if (arr[i] < min_val) {

            min_val = arr[i];
```

```
        }
    }
    return min_val;
}
// Function to find maximum value in an array of integers
int maximum(int arr[], int size) {
    int max_val = arr[0];
    for (int i = 1; i < size; ++i) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }
    return max_val;
}
int main() {
    int intArr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    double doubleArr[] = {3.5, 1.2, 4.7, 0.9, 5.4};
    cout << "Average of integers: " << average(intArr, 9) << endl;
    cout << "Minimum of doubles: " << minimum(doubleArr, 5) << endl;
    cout << "Maximum of integers: " << maximum(intArr, 9) << endl;
    return 0;
}
```

4. String Formatting (C#) - Write overloaded functions formatString that can take a format string and different data types (e.g., int, double, string) to create formatted output strings.

```
#include <iostream>
#include <sstream>
using namespace std;
// Function to format string with integer input
string formatString(const string& format, int number) {
```

```
    stringstream ss;
    ss << "Formatted output: " << format << " " << number;
    return ss.str();
}
// Function to format string with double input
string formatString(const string& format, double number) {
    stringstream ss;
    ss << "Formatted output: " << format << " " << number;
    return ss.str();
}
// Function to format string with string input
string formatString(const string& format, const string& text) {
    stringstream ss;
    ss << "Formatted output: " << format << " " << text;
    return ss.str();
}
int main() {
    cout << formatString("Number:", 42) << endl;
    cout << formatString("Value:", 3.14159) << endl;
    cout << formatString("Hello,", "World!") << endl;
    return 0;
}
```

5. Math Functions (Python) - Create overloaded functions factorial and power that can handle integer and floating-point input for calculating factorials and raising a number to a power.

```
#include <iostream>
using namespace std;
```

```
// Function to calculate factorial of an integer
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}
// Function to calculate power of a number (base ^ exponent)
double power(double base, int exponent) {
    double result = 1.0;
    for (int i = 0; i < exponent; ++i) {
        result *= base;
    }
    return result;
}
// Overloaded function to calculate power for floating-point base
double power(double base, double exponent) {
    return pow(base, exponent);
}
int main() {
    cout << "Factorial of 5: " << factorial(5) << endl;  // Output: 120
    cout << "2 ^ 3: " << power(2, 3) << endl;  // Output: 8
    cout << "2.5 ^ 2.0: " << power(2.5, 2.0) << endl;  // Output: 6.25
    return 0;
}
```