Object-Oriented Programming

Lecture 7: Operator Overloading

Overloaded Operators

- It's a feature that allows you to redefine the behavior of built-in operators for user-defined types (like classes).
- You redefine operators by defining special functions called operator functions.
- This enables you to use familiar operators with custom objects, making code more intuitive and expressive.

lect08_01.cpp
lect08_02.cpp

Assignment Operator (=)

- The assignment operator performs memberwise assignment for most classes. This means that each data member of the "source" object (on the right side of the =) is copied to the corresponding data member of the "target" object (on the left side of the =).
- For classes containing pointer members, memberwise assignment can lead to shallow copying, the pointer members in both the source and target objects point to the same memory location. This can cause problems like double deletion (when both objects are destroyed) and unintended data sharing. To prevent these issues, the assignment operator should be explicitly overloaded to perform a deep copy or to handle the pointer members appropriately.

lect08_03.cpp

Address-of Operator (&)

- The address-of operator is used to obtain the memory address of an object. Although not commonly overloaded, it is possible to do so. Overloading this operator can change its default behavior, but this is rarely needed and can lead to confusion, so it's typically avoided in practice.

Comma Operator (,)

- The comma operator evaluates the expression on its left, then evaluates the expression on its right, and returns the result of the right-hand expression. This operator can also be overloaded to define custom behaviors when using the comma in expressions involving class objects. However, overloading the comma operator is uncommon and generally discouraged because it can lead to code that's difficult to understand and maintain.

Operators That Cannot Be Overloaded

- Scope Resolution Operator (::): This operator is used to define the scope of a function or a variable, indicating the class or namespace to which the member belongs.
- Member Access or Dot Operator (.): This operator is used to access a member of a structure or an object.
- Member Pointer Access Operator (.*): This operator is used in conjunction with pointers to classes to access members and member functions of an object.
- Ternary Conditional Operator (?:): This operator is used as a shorthand for if-else statements to choose between two expressions based on the evaluation of a condition.
- Sizeof Operator: Returns the size of a variable or a data type.
- typeid Operator: Returns the type of an object.
- static_cast, dynamic_cast, const_cast, reinterpret_cast: Type conversion.

static_cast

- Basic Type Conversions, conversions between numeric types, such as from int to float or double to int.

```
int i = 10;
float f = static_cast<float>(i); // Convert int to float
double d = 3.14;
int j = static_cast<int>(d); // Convert double to int
```

- Pointer Conversions (Upcasting): Safely convert a pointer of a derived class to a pointer of its base class.

```
class Base { /*... */ };
class Derived: public Base { /*... */ };
Derived* derivedPtr = new Derived();
Base* basePtr = static_cast<Base*>(derivedPtr); // Upcast - generally safe
```

- Pointer Conversions (Downcasting): Convert a base class pointer to a derived class pointer. This is **potentially risky** if the base class pointer doesn't actually point to a Derived object.

dynamic_cast

- Safe Downcasting with Runtime Checks: Use dynamic_cast to convert pointers or references in a class hierarchy. It performs a runtime check to ensure the conversion is valid. If the conversion is not valid, it returns nullptr for pointers or throws an exception (std::bad_cast) for references.
- Requires RTTI: dynamic_cast relies on Run-Time Type Information (RTTI), which might add some overhead. You can disable RTTI if you don't need it.

const_cast

- Removing const or volatile: Use const_cast to remove the const or volatile qualifiers from a variable. This should be used with extreme caution, as modifying a variable that was originally declared const can lead to undefined behavior.

```
const int x = 10;
int* ptr = const_cast<int*>(&x); // Remove const-ness from x (use with care!)
*ptr = 20; // Potentially dangerous!
```

reinterpret_cast

- Low-Level Reinterpretation
- It converts any pointer type to any other pointer type, even unrelated ones. It can also cast pointers to integers and vice versa. This is generally used for very specific scenarios and can be unsafe if not used carefully.

```
int i = 42;
int* ptr = &i;
char* charPtr = reinterpret_cast<char*>(ptr); // Treat the integer as a sequence of
bytes

// Example: Accessing individual bytes of a float
float f = 3.14f;
unsigned char* bytePtr = reinterpret_cast<unsigned char*>(&f);
for (int i = 0; i < sizeof(float); ++i) {
   std::cout << static_cast<int>(bytePtr[i]) << " ";
}
```

Rules and Restrictions on Operator Overloading

1. **Precedence Unchanged**: Operator precedence remains unchanged. Use parentheses to control the evaluation order.
2. **Associativity Unchanged**: An operator's associativity (left-to-right or right-to-left) remains as per its default. (Left-associative: 2 - 3 + 4 Right-associative: a = b = 5)
3. **Arity Unchanged**: The arity of an operator, or the number of operands it takes, cannot be modified through overloading.

   - Unary operators (taking one operand -, !) remain unary, and

   - Binary operators (taking two operands +, -, >, <, etc.) remain binary even after being overloaded.

   - Operators like &, *, +, and -, which have both unary and binary forms, can have each form overloaded separately.

Rules and Restrictions on Operator Overloading

4. **No New Operators**: If you need such functionality, you must use a regular function instead.
5. **Fundamental Types Unchanged**: Overloaded operators cannot alter the behavior of operators for fundamental (built-in) types.
6. **Related Operators Overloaded Separately**: Operators that are related, such as + and +=, do not automatically share overloading definitions. Each must be overloaded individually if custom behavior is desired for both.
7. **Certain Operators Must Be Member Functions**: When overloading the function call operator (), the subscript operator [], the arrow operator ->, or any assignment operators, the overloaded operator function must be declared as a member of the class.

Rules and Restrictions on Operator Overloading

7. **Certain Operators Must Be Member Functions**: When overloading the function call operator (), the subscript operator [], the arrow operator ->, or any assignment operators.

```
class Example {
public:
    int value;

    Example(int v) : value(v) {}

    // Overloading []
    int& operator[](int index) {
        static int arr[10] = {0}; // Example internal array
        return arr[index]; // Simplified example
    }

    // Overloading =
    Example& operator=(const Example& other) {
        value = other.value;
        return *this;
    }
};
```

Overloading Unary Operators

- Allows to redefine the way unary operators work when applied to objects of a custom class. Unary operators are those that operate on a single operand. Common unary operators include:
  Increment (++)
  Decrement (--)
  Logical NOT (!)
  Unary minus (-)
  Address-of (&)
  Dereference (*)
- A member function that does not take any arguments (other than the object itself on which the operator is called)
- Returns a value that is typically either a modified object or a property of the object.

lect08_07.cpp

```cpp
#include <iostream>
#include <vector>
#include <memory>

class MyArray {
private:
    std::vector<int> data;

public:
    // Constructor to initialize the array with a given size and default value
    MyArray(int size, int defaultValue = 0) : data(size, defaultValue) {}

    // Overload the subscript operator []
    int& operator[](size_t index) {
        if (index >= data.size()) {
            throw std::out_of_range("Index out of range");
        }
        return data[index];
    }

    // Overload the function call operator ()
    int operator()(size_t index) const {
        if (index >= data.size()) {
            throw std::out_of_range("Index out of range");
        }
        return data[index];
    }

    // Additional functionality for demonstration
    void print() const {
        for (auto val : data) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
    }
};

// Example structure for demonstrating the arrow operator ->
struct MyStruct {
    void print() const { std::cout << "MyStruct::print() called" << std::endl; }
};

// Overload the arrow operator ->
MyStruct* operator->() const {
    static MyStruct example;
    return &example;
}

// Overload the copy assignment operator =
MyArray& operator=(const MyArray& other) {
    if (this != &other) {
        data = other.data;
    }
    return *this;
}

int main() {
    MyArray arr(5, 10); // Create an array of size 5, initialized to 10

    // Demonstrate subscript operator []
    std::cout << "Element at index 2: " << arr[2] << std::endl;

    // Demonstrate function call operator ()
    std::cout << "Element at index 2 using operator(): " << arr(2) << std::endl;

    // Demonstrate arrow operator ->
    arr->print(); // Calls MyStruct::print()

    // Demonstrate assignment operator =
    MyArray arr2(5);
    arr2 = arr; // Copy data from arr to arr2
    std::cout << "arr2 after assignment: ";
    arr2.print();

    return 0;
}
```

lect08_04.cpp

```cpp
#include <iostream>

class Complex {
public:
    double real, imag;
    Complex(double real = 0.0, double imag = 0.0) : real(real), imag(imag) {}

    // Member function for operator overloading
    // Overloads the + operator to add two complex numbers.
    // It takes a const reference to another Complex object and returns a new Complex object representing the sum.
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }
    // Friend function for output formatting (optional)
    // that overloads the << operator to allow printing of Complex objects to output streams
    friend std::ostream& operator<<(std::ostream& os, const Complex& c) {
        os << "(" << c.real << ", " << c.imag << ")";
        return os;
    }
};

int main() {
    Complex c1(2, 3);
    Complex c2(4, 5);
    Complex c3 = c1 + c2;  // Using overloaded + operator
    std::cout << "c1: " << c1 << std::endl;  // Printing using friend function
    std::cout << "c2: " << c2 << std::endl;
    std::cout << "c3: " << c3 << std::endl;
    return 0;
}
```

```
c1: (2, 3)
c2: (4, 5)
c3: (6, 8)
```

lect08_05.cpp

Overloading the Binary Stream Insertion and Stream Extraction Operators

- Friend Function:
  - Declared as a friend of the PhoneNumber class, allowing access to private or protected members of the class.
  - Used to output the contents of a PhoneNumber object to an output stream (e.g., std::cout).
- Return Type (std::ostream&):
  - Returns a reference to an std::ostream object, enabling chaining of multiple stream insertions.
- Function Name (operator<<):
  - Overloads the << operator for the PhoneNumber class to customize its output formatting
- Parameters:
  - std::ostream& output: A reference to the output stream where data will be written.
  - const PhoneNumber& number: A constant reference to the PhoneNumber object whose data will be formatted and inserted into the stream.

lect08_06.cpp

Case Study: A Date Class

friend std::ostream& operator<<(std::ostream&, const Date&);

- friend: indicates that the function being declared, although not a member of the Date class, has access to the private and protected members of the class.
- std::ostream&: This specifies the return type of the function, which is a reference to an std::ostream object. Returning a reference to the stream allows the function to support chaining of stream insertions.
- operator<<: This part signifies that the function is an overload of the << operator, which is typically used for output operations in C++. Overloading this operator for custom types like Date allows these types to be used with standard C++ output streams.
- (std::ostream&, const Date&): These are the parameters of the function. The first parameter is a reference to an std::ostream object, which is the destination stream for the output (e.g., std::cout). The second parameter is a reference to a constant Date object, which is the source of the data being output. The const qualifier ensures that the Date object is not modified by the function.

lect08_08.cpp

Dynamic Memory Management

- Manual control of memory allocation and deallocation during the runtime of a program.
- Unlike automatic or static memory allocation, which is managed by the compiler, dynamic memory is allocated from the heap, and it's the programmer's responsibility to explicitly allocate and free this memory.
- In C++, dynamic memory management is primarily done using the new and delete operators, as well as their array forms new[] and delete[].

Dynamic Memory Management

- Allocation single object
  int* ptr = new int; // Allocate memory for a single int
- Allocation array of objects
  int* arrayPtr = new int[10]; // Allocate memory for an array of 10 ints
- Initilization
  int* ptr = new int(42); // Allocate an int, initialized to 42
- Deallocation
  delete ptr; // Free memory for a single int
  delete[] arrayPtr; // Free memory for an array of 10 ints

Dynamic Memory Management

- **Flexibility**:
  - Allows programs to request memory at runtime based.
  - Useful for data structures whose size cannot be determined beforehand.
- **Efficient Memory Usage**:
  - Allocate only as much memory as needed during runtime, it can lead to more efficient use of memory resources.
- **Control**: Programmers have direct control over how and when memory is allocated and deallocated. This level of control can optimize performance and memory usage, particularly in systems where resource management is critical.
- **Scalability**: Applications can scale more effectively because they can allocate more memory space as needed. This allows for the creation of data-driven applications that can handle varying amounts of data.

Dynamic Memory Management

- **Data Lifetime**: With dynamic memory, data can persist beyond the scope in which it was created. This allows for the creation of complex data structures that need to maintain state or data across different parts of a program or throughout the program's lifecycle.
- **Allocation of Large Objects**: It's often impractical to allocate very large objects on the stack due to size limitations; dynamic memory allocation allows for the creation of large objects on the heap.
- **Data Structures Libraries**: It enables the development of advanced library data structures and algorithms, such as those found in the Standard Template Library (STL), which can handle data of arbitrary size or quantity.

Dynamic Memory Management

- Match Allocation and Deallocation
- Avoid Memory Leaks: Ensure that you deallocate all dynamically allocated memory when it is no longer needed. Memory leaks occur when allocated memory is not freed.
- Handle Exceptions: The new operator can throw an exception (typically std::bad_alloc) if memory cannot be allocated
- Use Smart Pointers: Modern C++ encourages the use of smart pointers (std::unique_ptr, std::shared_ptr, std::weak_ptr) which automatically manage dynamic memory, helping to avoid leaks and dangling pointers.
- RAII (Resource Acquisition Is Initialization): It is a design pattern that binds the lifecycle of resources (like dynamically allocated memory) to the lifetime of an object, ensuring proper cleanup (like calling delete) when the object goes out of scope.

Homework

- Random generate maze 15 x 15 in text mode
- Maze must have one entry one exit
- Display generated maze and solution path
- Submit code on Feb 16th

```
###############.#
#.......................#
#.################
#..............#.......#
########.#.#####
#.....................#
#.#######.#######
#.#.....................#
#.################
```

Q & A