**Programming assignment**

**Chapter 14 keyword**

**Operator**

In mathematics, computer programming, and other related subjects, the word "operator" often refers to a symbol or function that is used to carry out a certain operation or computation. There a different type such as arithmetic operators like addition and subtraction, comparison operators like "less than" and "greater than," and logical operators like "AND" and "OR." Several different types of operators, such as arithmetic operators, bitwise operators, and relational operators, are used to manipulate data and execute computations in computer programming. Operators are crucial in the rapid completion of complicated calculations and activities.

**Float**

"Float" is a data type used to represent floating-point values with a single precision declared as follows (float variable name = value;). Float variables are frequently used when less accuracy than a double is required since they may store real values with decimal points. a primitive data type that can represent single precision floating point integers with an accuracy of six to seven significant digits. With 14 to 15 significant digits of accuracy, double precisions are represented by the double type. Float variables are widely used in graphics and game development, where speed and memory efficiency are vital because the level of precision required is not as high as it is in scientific or financial applications.

**Int**

In computer programming, "int" is a data type used to represent integer values. An integer is a whole number, meaning that it does not have a decimal point or fraction. In C++, an int occupies 4 bytes of memory (on most platforms) and can store values ranging from approximately -2,147,483,648 to 2,147,483,647. Integers are commonly used in programming for counting, indexing, and looping, among other tasks. They are useful for representing quantities that are always whole, such as the number of items in a list or the number of iterations in a loop. To declare an int variable in C++, the syntax is: "int variable name = value;".

**Delete**

If a class object contains a destructor, it is called before the memory is deallocated when delete is used to deallocate the object's memory. When an object is erased, the value of an operand supplied to the delete operator that is a changeable value becomes undefined. When memory is dynamically allocated on the heap using the "new" keyword, one is in charge of deallocating the memory when you are done with it. The memory can be deallocated and made available for other purposes by using the "delete" keyword. It is used in conjunction with the pointer to the memory that was allocated to delete a variable or array that was dynamically allocated.

**Programming exercise 14.10 (convert decimals to fractions)**

#include <iostream>

#include <string>

#include <cmath>

#include <boost/multiprecision/cpp\_int.hpp>

#include <boost/rational.hpp>

using namespace std;

using namespace boost::multiprecision;

//prompt user to enter a decimal number

int main() {

string decimal;

cout << "Enter a decimal number: ";

cin >> decimal;

// Extract integer and fractional parts

size\_t dot\_pos = decimal.find(".");

cpp\_int int\_part = cpp\_int(decimal.substr(0, dot\_pos));

cpp\_int frac\_part = cpp\_int(decimal.substr(dot\_pos + 1));

// Compute denominator and numerator

cpp\_int denominator = pow(cpp\_int(10), decimal.length() - dot\_pos - 1);

cpp\_int numerator = int\_part \* denominator + frac\_part;

// Create rational number from numerator and denominator

boost::rational<cpp\_int> r(numerator, denominator);

// printing the rational number as a fraction using the ‘r’ object instance

cout << "The rational representation of " << decimal << " is " << r << endl;

return 0;

}

**Chapter 15 keyword**

**Bool**

The name "bool" stands for "boolean," a basic data type in programming that symbolizes a logical value. As the only two possible truth values for a logical expression, "True" or "False" are employed to indicate them. Programmers frequently use boolean values to simplify decision-making and control program flow. Without them, it is difficult to create conditional statements, loops, and other control structures.

**Override**

The "override" keyword is used to designate when a virtual function in a derived class is intended to replace a virtual function with the same name and signature in the base class. Virtual functions in derived classes that are marked as overrides must have the same function signature as the base class function they are intended to replace. This requirement is indicated by the override keyword. This might help identify any problems that might arise if the function signature is unintentionally changed and ensures that the derived class is properly overriding the base class function. When overriding virtual functions in a derived class, the override keyword is optional but encouraged. The compiler will produce an error if the function signature is incorrect, which can assist in identifying mistakes early in the development process.

**Final**

When a class, function, or virtual function cannot be further derivate from or overridden in a derived class, the keyword "final" is used to denote this. A class that is declared as final cannot be descended from and cannot be used as a base class for any subsequent classes. Similarly to this, a virtual function that is designated as final cannot be overridden in any class that derives from it. The final keyword may only be applied to virtual functions, classes, and other similar sorts of things. The final keyword can be used to guarantee the stability and accuracy of a program by forbidding additional modifications to the behavior of a class or function.

**Private**

When a class declaration uses the "private" keyword as an access specifier, it means that only member functions of the same class may access the members stated below it. A member variable or function is made private when it is specified to be accessible only by other member functions of the class. This makes it more difficult for outside code to directly alter the internal data of a class and helps to encapsulate the implementation details of a class.

**Programming exercise 15. 1 (the triangle class)**

#include <iostream>

#include <cmath>

#include "GeometricObject.h"

class Triangle : public GeometricObject {

private:

double side1, side2, side3;

public:

// Default constructor

Triangle() : side1(1.0), side2(1.0), side3(1.0) {}

// Constructor with specified sides

Triangle(double s1, double s2, double s3) : side1(s1), side2(s2), side3(s3) {}

// Accessor functions for sides

double getSide1() const { return side1; }

double getSide2() const { return side2; }

double getSide3() const { return side3; }

// Function to calculate the area of the triangle using Heron's formula

double getArea() const {

double s = (side1 + side2 + side3) / 2.0;

return std::sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

// Function to calculate the perimeter of the triangle

double getPerimeter() const {

return side1 + side2 + side3;

}

};

We then use the triangle class created above together with geometric class to implement a program.

#include <iostream>

#include "Triangle.h"

int main() {

double side1, side2, side3;

std::string color;

int filled;

// Prompt user to enter triangle sides, color and filled status

std::cout << "Enter the three sides of the triangle: ";

std::cin >> side1 >> side2 >> side3;

std::cout << "Enter a color for the triangle: ";

std::cin >> color;

std::cout << "Is the triangle filled? (1 for yes, 2 for no): ";

std::cin >> filled;

// Create a Triangle object with the user-specified sides

Triangle triangle(side1, side2, side3);

// Set the color and filled properties of the triangle object

triangle.setColor(color);

triangle.setFilled(filled == 1);

// Display the area, perimeter, color, and filled status of the triangle

std::cout << "Area of the triangle is " << triangle.getArea() << std::endl;

std::cout << "Perimeter of the triangle is " << triangle.getPerimeter() << std::endl;

std::cout << "Color of the triangle is " << triangle.getColor() << std::endl;

std::cout << "The triangle is " << (triangle.isFilled() ? "filled" : "not filled") << std::endl;

return 0;

}

**UML Diagram**

|  |
| --- |
| GeometricObject |
|  |
| -color: std::string  -filled: bool |
|  |
| +GeometricObject()  +GeometricObject(std::string, bool)  +getColor(): std::string  +setColor(std::string): void  +isFilled(): bool  +setFilled(bool): void  +getArea(): double  +getPerimeter(): double |
|  |

|  |
| --- |
| Triangle |
|  |
| -side1: double  -side2: double  -side3: double |
|  |
| +Triangle()  +Triangle(double, double, double)  +getSide1(): double  +getSide2(): double  +getSide3(): double  +getArea(): double  +getPerimeter(): double |

**Chapter 16 keywords**

**Catch**

The catch keyword captures and handles errors or unusual occurrences in exception handling. The catch block is run when a try block throws an exception, enabling the program to correct the problem and carry on with its work. The syntax for the catch keyword calls for a block of code to be surrounded in braces before the kind of exception to catch is specified in parentheses following the keyword. To handle different exception types, different catch blocks can be used, with the most specific kind specified first. If there isn't a catch block that matches the thrown exception, the program will terminate early.

**Throw**

Using the "throw" keyword, an error or other unusual event that occurs while a program is executing can be explicitly tossed. When an exception is thrown, it is caught by the nearest enclosing try-catch block that can handle it. Any expression that evaluates to an exception object may be used as the exception to be thrown, according to the syntax for the throw keyword. To provide exception management in C++, the "throw" keyword is frequently used in conjunction with the "try" and "catch" keywords. Try-catch blocks let programs handle failures gracefully so that they don't result in a program crash.

**New**

The "new" keyword is used to facilitate dynamic memory allocation for objects. Memory is allocated from the heap rather than the stack when an object is formed with the "new" keyword, enabling more flexible memory management. After stating the kind of object to be formed, the new keyword is followed by parentheses that may or may not include optional parameters for the object's constructor. When an object is created with the "new" keyword, it remains in memory until it is explicitly destroyed using the "delete" keyword. Applications can dynamically allocate memory using "new" as needed rather to rely on a fixed amount of memory that is placed on the stack.

**Try**

In exception handling, the "try" keyword is used to designate a block of code where exceptions could be raised. The "try" block's goal is to catch and handle any exceptions that are thrown while the code inside the block is being executed. The block of code that could throw exceptions must be enclosed in a set of curly braces after the try keyword to use the syntax for it. When implementing exception handling the "try" and "catch" keywords are frequently combined. Try-catch blocks let programs handle failures gracefully so that they don't result in a program crash.

**Programming exercise 16.8 (modify StackOfIntergers class)**

#include <iostream>

#include <exception>

class EmptyStackException : public std::exception {

public:

virtual const char\* what() const throw() {

return "Stack is empty!";

}

};

class StackOfIntegers {

public:

StackOfIntegers() : size\_(0) {}

bool empty() const { return size\_ == 0; }

int peek() const {

if (empty()) throw EmptyStackException();

return elements\_[size\_ - 1];

}

void push(int value) {

if (size\_ == 100) throw std::out\_of\_range("Stack is full!");

elements\_[size\_++] = value;

}

int pop() {

if (empty()) throw EmptyStackException();

return elements\_[--size\_];

}

int getSize() const { return size\_; }

private:

int elements\_[100];

int size\_;

};

int main() {

StackOfIntegers stack;

try {

stack.push(1);

stack.push(2);

std::cout << stack.pop() << std::endl; // output: 2

std::cout << stack.pop() << std::endl; // output: 1

std::cout << stack.pop() << std::endl; // throws EmptyStackException

}

catch (const std::exception& e) {

std::cerr << e.what() << std::endl; // output: Stack is empty!

}

return 0;

}

**Chapter 17 keyword**

**Return**

This is a function can be terminated and a value returned to the program or function that called it using the return keyword. The execution of a function is instantly stopped when the return keyword is used in it, and control is then sent back to the program or function that called it. When a certain condition is satisfied, it is common practice to stop a function early using the return statement.

**Switch**

Depending on the value of a variable or expression, an application can run multiple code blocks using a switch statement, which is a control flow statement. The term "switch" precedes a variable or expression that is surrounded in parentheses, and then a series of case statements that compare the variable or expression to a series of constant values make up the syntax of the statement. If a match is discovered, the code block that follows the case statement is run. The default case in the switch statement is also executed if any other case statements fail to match the variable or expression. Code readability and maintainability may both improve thanks to the switch statement's ability to simplify complex decision-making logic. It only works with certain kinds of data.

**If**

The "if" keyword is a conditional statement that is employed to determine if a certain condition is true or false. True IF condition allows the programmer to execute one set of instructions; if the condition is false, it allows the programmer to execute another set of instructions. The word "if" precedes the usage of the keyword in a straightforward syntax, which is then followed by parentheses carrying the condition that has to be assessed. If the condition is satisfied, the code included in the curly brackets after the "if" expression will be run. The code inside the curly brackets will not be executed if the condition evaluates to false; instead, execution will go on to the code that follows the closing curly brace. Programmers may build complicated, sophisticated systems that can respond to changing circumstances by using the "if" statement, which is a strong technique in programming.

**Namespace**

A namespace is a feature that enables programmers to categorize their code logically to prevent naming conflicts. It serves as a sort of container for a collection of identifiers that fall under a certain umbrella. The moment a namespace is formed, all stated identifiers are added to that namespace. As a result, there won't be any naming conflicts when two identifiers with the same name coexist in distinct namespaces. The "namespace" keyword, the namespace name, and the curly brackets that surround the namespace's identifiers make up the syntax for utilizing namespaces in C++. When accessing an identifier inside of a namespace, identifier should be preceded with the name of the namespace and the scope resolution operator (::). Large codebases may be organized using the namespace keyword, making it possible to distinguish between different revisions of the same code. Due to their ability to avoid naming conflicts and guarantee that each identifier is uniquely identifiable inside its own namespace, they are especially crucial for creating libraries and frameworks where numerous developers may be working on the same codebase.

**Programming exercise 17.1 (compute factorials**

#include <iostream>

// Return the factorial for a specified index

long long factorial(int n) {

if (n < 0) {

std::cerr << "Error: Factorial is not defined for negative integers." << std::endl;

exit(1);

}

long long result = 1;

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

result \*= i;

}

return result;

}

int main() {

// Prompt the user to enter an integer

std::cout << "Enter a non-negative integer: ";

int n;

std::cin >> n;

// Display factorial

std::cout << "Factorial of " << n << " is " << factorial(n) << std::endl;

return 0;

}

**Chapter 18 keywords**

**Double**

A variable that may hold floating-point integers with double precision is a "double". It is a flexible data type that may be used to represent any numerical value, including decimal values, in the compiler. There are two distinct categories of double data types: whole numbers and fractional numbers with values. A double 1.8e+308 uses 8 bytes of memory, has a precision of 15 to 16 significant digits, and can store values between 2.23e-308. Double variables are advantageous when showing figures that require greater accuracy, such as in financial or scientific calculations involving small or large values.

**Const**

The keyword "const" is short for "constant" and is used in programming to define a variable or object that cannot be modified after it has been declared. In other words, once a value is assigned to a "const" variable, it cannot be changed during the program's execution. This makes "const" variables useful for defining values that should remain constant throughout the program, such as mathematical constants, fixed configuration settings, or unchanging data. Using "const" variables can also help prevent unintended changes to the program's state, which can lead to bugs and errors. The "const" keyword is commonly used in programming languages such as C++, Java, and JavaScript.

**Else**

When the condition stated in the if statement is false, the else keyword in C++ is used to run a block of code. If the else block is missing, the computer will skip the if statement and move on to the next line of code. The else block is optional. To build nested conditional statements, the else block can also include its own if statement in addition to any number of other statements. It's crucial to keep in mind that the otherwise block is connected to the nearest if statement that does not already have an else block connected to it.

**Long**

When declaring integer variables that need a wider range of values than the basic integer data type can accommodate, the term "long" is used. There are three alternative methods to use the keyword "long": "long", "long int", or "long long". The integer data type "long long" may store values that are even greater than those that can be stored in the "long" data type. Most systems generally use 64 bits. It's crucial to remember that a variable declared with the "long" keyword could use more memory than a regular integer variable. Additionally, "long" variables may require more time to complete arithmetic operations than regular integer variables do.

**Programming exercise 18.1 (maximum consecutive increasingly ordered substring)**

#include <iostream>

#include <string>

using namespace std;

int main() {

string s;

cout << "Enter a string: ";

cin >> s;

string max\_substring;

string current\_substring;

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

if (i == 0 || s[i] > s[i-1]) {

current\_substring += s[i];

} else {

if (current\_substring.size() > max\_substring.size()) {

max\_substring = current\_substring;

}

current\_substring = s[i];

}

}

if (current\_substring.size() > max\_substring.size()) {

max\_substring = current\_substring;

}

cout << "Maximum consecutive increasingly ordered substring: " << max\_substring << endl;

return 0;

}

The above code example has a complexity of (O)n. n being the length of the input string. These is because the program iterates over the string once and performs consecutive constant-time operations for each character.

**Chapter 19**

**For**

The "for" keyword is used to build a loop that runs a piece of code continuously until a certain condition is fulfilled. The initialization statement, the condition statement, and the iteration statement make up the "for" loop's particular syntax. The loop counter variable is initialized using the initialization statement, which is run just once at the start of the loop. At the start of each iteration of the loop, the condition statement is examined, and if it evaluates to true, the block of code contained within the loop is performed. The loop counter variable is updated using the iteration statement, which is run at the conclusion of each loop iteration.

**While**

While a certain condition is satisfied, a loop that continually runs a block of code is created using the "while" keyword. The "while" loop has a more straightforward syntax than the "for" loop. As long as the condition evaluates to true, the while loop constantly runs the block of code inside it. Each time the loop iterates, the condition is checked at the start, and the loop is stopped if it evaluates to false. In C++, it is frequently used for iterating through data structures or carrying out a series of iterations up until a certain condition is satisfied. The loop condition must finally become false for the loop to stop, failing to do so will cause an infinite loop.

**Break**

Using the "break" keyword, you may end a loop or switch expression. The program will instantly leave a loop or switch statement that contains a break statement and continue running the code after the loop or switch. When a certain condition is satisfied, a break statement in a loop is frequently used to end the loop early. For instance, rather than continuing to iterate through the remaining items if you are searching for a certain value within an array, you can use a break statement to end the loop once the value is located. A break statement is used in a switch statement to stop the flow of control to the next case statement. If there were no break statements, the program would always run the code in the following case statement, whether or not the case condition was satisfied.

**Char**

When defining a variable that can store one character, the "char" keyword is used. Any letter, number, or symbol that may be represented in the ASCII or Unicode character set is considered to be a character. A char variable is kept in a single byte of memory and may contain any value between -128 and 127 or 0 and 255, depending on whether it is signed or unsigned. A char variable is declared by prefixing the variable name with the "char" keyword.

**Programming exercise 19.1 (Generic bubble sort)**

template<typename d>

void bubbleSort(T arr[], int n) {

for(int i = 0; i < n - 1; i++) {

for(int j = 0; j < n - i - 1; j++) {

if(arr[j] > arr[j+1]) {

std::swap(arr[j], arr[j+1]);

}

}

}

}

The above function takes an array of n elements of any data type d and sorts the array using bubble algorithm in ascending order.

**Chapter 20 keywords**

**Typename**

When a dependent name in a template is a type, the "typename" keyword is used to indicate such. The "typename" keyword is used to tell the compiler that a dependent name is a type rather than a value. A dependent name is a name that depends on a template parameter. It must be used if the name in the template definition qualifies and dependent on template argument. It is not allowed in a base class list unless as a template argument to a template class

**Class**

A class is a template, blueprint or contract that predefines the objects data fields and functions. It is a set of instructions for making objects or instances of the class. You can define member variables, member functions, and access specifiers when defining a class, which controls how the class members can be accessed. The data components that make up the class are represented by the member variables, and the operations or behaviors that may be applied to the data are represented by the member functions.

**This**

The term "this" refers to the object that is presently being picked. It is a reserved term that may be used to refer to the object that a member function of a class is currently working on. A class-defined member function operates on the object that calls it. You can make references to that object within the member function by using the keyword "this." The "this" keyword allows you to return a reference to the object itself as well as access the object's member variables and member functions. The "this" keyword can only be used inside of member functions; it is ineffective outside of classes. In addition, the term "this" should only be used to distinguish between a member variable or member function and a local variable or function parameter that shares the same name.

**Nullpntr**

Null pointers, or pointers that do not point to any objects or functions, are denoted by the term "nullptr". It was made available in to take the place of the NULL macro, which was used to represent a null pointer in earlier iterations of the language. To declare a pointer variable to be null, the nullptr keyword can be used as a literal. Only pointers of the same type may be assigned to it since it is type-safe. Errors are less likely to occur when using the NULL, which is just a value of zero that may be supplied to any pointer type. Only pointers may be used with the nullptr keyword; all other types are incompatible. Additionally, as it is more clear and safer, the nullptr keyword ought to be used in new code instead of the NULL.

**Programming exercise 20.9 (implement stack using inheritance)**

#include <iostream>

template <typename T>

class LinkedList {

private:

struct Node {

T data;

Node\* next;

Node(T data) : data(data), next(nullptr) {}

};

Node\* head;

int length;

public:

LinkedList() : head(nullptr), length(0) {}

void push\_front(T data) {

Node\* new\_node = new Node(data);

new\_node->next = head;

head = new\_node;

length++;

}

T pop\_front() {

if (head == nullptr) {

throw std::out\_of\_range("LinkedList is empty");

}

Node\* old\_head = head;

T data = old\_head->data;

head = old\_head->next;

delete old\_head;

length--;

return data;

}

int size() const {

return length;

}

bool is\_empty() const {

return length == 0;

}

~LinkedList() {

Node\* current = head;

while (current != nullptr) {

Node\* next = current->next;

delete current;

current = next;

}

}

};

template <typename T>

class Stack : public LinkedList<T> {

public:

void push(T data) {

this->push\_front(data);

}

T pop() {

return this->pop\_front();

}

int size() const {

return this->LinkedList<T>::size();

}

bool is\_empty() const {

return this->LinkedList<T>::is\_empty();

}

};

int main() {

Stack<int> s;

s.push(1);

s.push(2);

s.push(3);

while (!s.is\_empty()) {

std::cout << s.pop() << " ";

}

std::cout << std::endl;

return 0;

}