**Технология программирования**

**1. Односвязные и двусвязные списки. Очереди и стеки.**

**Связные списки**

**Связный список** — динамическая структура данных, состоящая из узлов, каждый из которых содержит данные и указатель(и) на следующий (и предыдущий) узел.

Односвязный список:

struct Node {  
 int data; // Данные узла  
 Node\* next; // Указатель на следующий узел  
};

Особенности: - Каждый узел знает только о следующем - Добавление/удаление в начало списка — O(1) - Для доступа к произвольному элементу нужен последовательный обход - Требует меньше памяти, чем двусвязный список

Двусвязный список:

struct Node {  
 int data; // Данные узла  
 Node\* next; // Указатель на следующий узел  
 Node\* prev; // Указатель на предыдущий узел  
};

Особенности: - Каждый узел знает о следующем и предыдущем узлах - Возможность двунаправленного перемещения по списку - Добавление/удаление в начало и конец — O(1) - Более эффективное удаление узлов - Требует больше памяти из-за дополнительного указателя

Стек (LIFO - Last In, First Out)

Абстрактная структура данных, в которой элементы добавляются и удаляются с одного конца.

Основные операции: - **push** — добавление элемента на вершину стека - **pop** — удаление элемента с вершины стека - **peek/top** — просмотр элемента на вершине без удаления - **isEmpty** — проверка, пуст ли стек

Применения: - Отслеживание вызовов функций - Проверка сбалансированности скобок - Обратная польская запись - Алгоритмы обхода в глубину (DFS) - Отмена операций (Undo)

Очередь (FIFO - First In, First Out)

Абстрактная структура данных, в которой элементы добавляются в конец и удаляются из начала.

Основные операции: - **enqueue** — добавление элемента в конец очереди - **dequeue** — удаление элемента из начала очереди - **peek/front** — просмотр первого элемента без удаления - **isEmpty** — проверка, пуста ли очередь

Применения: - Буферизация данных - Очереди задач в операционных системах - Обработка запросов (принтеры, веб-серверы) - Алгоритмы обхода в ширину (BFS) - Моделирование систем с очередями

**2. Определение класса. Создание и уничтожение объектов класса. Компоненты класса. Конструкторы и деструкторы. Правила преобразования указателей. Способы реализации инкапсуляции.**

**Определение класса**

**Класс** — это пользовательский тип данных, объединяющий данные и функции для работы с ними.

class ClassName {  
private:  
 // Закрытые члены  
protected:  
 // Защищенные члены  
public:  
 // Открытые члены  
 ClassName(); // Конструктор  
 ~ClassName(); // Деструктор  
};

Создание и уничтожение объектов

Создание:

ClassName obj; // В стеке (автоматически)  
ClassName\* ptr = new ClassName; // В куче (динамически)

Уничтожение:

delete ptr; // Для объектов в куче  
// Объекты в стеке уничтожаются автоматически при выходе из области видимости

Компоненты класса:

**Поля (данные-члены)** — переменные, определяющие состояние объекта

**Методы (функции-члены)** — функции, определяющие поведение объекта

**Конструкторы** — специальные методы для инициализации объектов

**Деструктор** — специальный метод для освобождения ресурсов

**Статические члены** — общие для всех объектов класса

Конструкторы и деструкторы:

Конструкторы:

class Person {  
public:  
 Person() { } // Конструктор по умолчанию  
 Person(string n) : name(n) { } // Параметризованный конструктор  
 Person(const Person& p) { } // Конструктор копирования  
private:  
 string name;  
};

Деструктор:

class Person {  
public:  
 ~Person() {  
 // Освобождение ресурсов  
 }  
};

Правила преобразования указателей:

От производного класса к базовому (восходящее) — безопасно, неявно

От базового класса к производному (нисходящее) — требует явного преобразования:

dynamic\_cast — с проверкой типа в runtime

static\_cast — без проверки типа

reinterpret\_cast — “грубое” преобразование

Инкапсуляция:

Механизм, объединяющий данные и методы, работающие с ними, и ограничивающий доступ к деталям реализации.

Способы реализации: - **Модификаторы доступа**: private, protected, public - **Геттеры и сеттеры**: методы для контролируемого доступа к данным - **Дружественные функции и классы**: для предоставления доступа избранным - **PIMPL** (Pointer to Implementation): скрытие реализации за указателем

**3. Наследование классов. Базовый и производный классы. Правила доступа к элементам производного класса. Иерархия классов.**

**Наследование**

**Наследование** — механизм, позволяющий создавать новые классы на основе существующих, расширяя их функциональность.

class Base {  
 // Базовый класс (суперкласс, родительский класс)  
};  
  
class Derived : public Base {  
 // Производный класс (подкласс, дочерний класс)  
};

Базовый и производный классы

**Базовый класс** — класс, от которого наследуются другие классы

**Производный класс** — класс, наследующийся от базового класса

Производный класс: - Наследует все члены базового класса (с учетом доступа) - Может добавлять новые члены - Может переопределять методы базового класса - Может расширять или специализировать функциональность

Правила доступа:

Доступность членов базового класса в производном зависит от спецификатора наследования:

**public наследование**:

public→public: остаются публичными

protected→protected: остаются защищенными

private→недоступно: остаются недоступными

**protected наследование**:

public→protected: становятся защищенными

protected→protected: остаются защищенными

private→недоступно: остаются недоступными

**private наследование**:

public→private: становятся приватными

protected→private: становятся приватными

private→недоступно: остаются недоступными

Иерархия классов:

**Простая иерархия**: один предок - несколько потомков (дерево)

**Многоуровневая иерархия**: цепочка наследования (A→B→C)

**Иерархия с множественным наследованием**: несколько предков (граф)

**Ромбовидная иерархия**: проблема “ромба” при множественном наследовании

**4. Одиночное и множественное наследование классов. Особенности доступа при множественном наследовании.**

**Одиночное наследование**

Класс наследуется от одного базового класса, формируя простую иерархию.

class Derived : public Base {  
 // Наследование от одного класса  
};

Преимущества: - Простота и ясность иерархии - Отсутствие конфликтов имен - Эффективная реализация в большинстве языков

Множественное наследование

Класс наследуется от двух или более базовых классов, получая функциональность всех предков.

class Derived : public Base1, public Base2 {  
 // Наследование от нескольких классов  
};

Преимущества: - Возможность комбинировать функциональность разных классов - Естественное моделирование сложных отношений

Недостатки: - Усложнение иерархии классов - Возможные конфликты имен - Проблема ромбовидного наследования

Особенности множественного наследования:

1. Конфликт имен

Если два базовых класса содержат члены с одинаковыми именами, возникает неоднозначность:

derived.Base1::member; // Разрешение конфликта через указание класса

2. Порядок вызова конструкторов и деструкторов

Конструкторы базовых классов вызываются в порядке объявления в списке наследования

Деструкторы вызываются в обратном порядке

3. Проблема ромбовидного наследования

A  
 / \  
 B C  
 \ /  
 D

Решение — виртуальное наследование:

class B : virtual public A { };  
class C : virtual public A { };  
class D : public B, public C { };

**5. Виртуальные базовые классы. Виртуальные функции.**

**Виртуальные базовые классы**

Виртуальное наследование решает проблему ромбовидного наследования, гарантируя единственный экземпляр базового класса в иерархии.

class Base { };  
class Derived1 : virtual public Base { };  
class Derived2 : virtual public Base { };  
class Final : public Derived1, public Derived2 { }; // Только один экземпляр Base

Особенности: - Предотвращение дублирования экземпляра базового класса - Возможность однозначного доступа к членам виртуального базового класса - Конструктор виртуального базового класса вызывается самым последним производным классом

Виртуальные функции

Виртуальные функции — механизм, позволяющий переопределять функции в производных классах для реализации полиморфизма.

class Base {  
public:  
 virtual void func() { } // Виртуальная функция  
 virtual ~Base() { } // Виртуальный деструктор  
};  
  
class Derived : public Base {  
public:  
 void func() override { } // Переопределение  
};

Особенности: - Поздняя (динамическая) привязка вызовов - Определение конкретной реализации во время выполнения - Таблица виртуальных функций (vtable) для эффективной реализации - Ключевое слово override для явного указания переопределения (C++11) - Необходимость виртуального деструктора для корректного удаления объектов

Применение: - Полиморфизм — возможность использовать объекты разных классов через указатель на базовый класс - Абстрактные классы и интерфейсы - Реализация паттернов проектирования (Стратегия, Наблюдатель и др.)

**6. Шаблоны классов и функций. Правила отождествления параметров шаблона.**

**Шаблоны функций**

Шаблоны функций позволяют создавать функции, работающие с разными типами данных без дублирования кода.

template <typename T>  
T max(T a, T b) {  
 return (a > b) ? a : b;  
}

Применение: - Обобщенные алгоритмы, не зависящие от типа данных - Типобезопасные контейнеры и функции - Метапрограммирование

Шаблоны классов

Шаблоны классов позволяют создавать параметризованные типы данных.

template <typename T>  
class Stack {  
private:  
 T\* data;  
 int size;  
 int capacity;  
public:  
 void push(T value);  
 T pop();  
 T top() const;  
 bool isEmpty() const;  
};

Преимущества: - Возможность создания универсальных контейнеров (vector, list, map) - Типобезопасность - Оптимизация для конкретных типов - Возможность специализации для определенных типов

Правила отождествления параметров шаблона:

**Явное указание типа**:

max<int>(5, 10);

Тип параметра задается явно в угловых скобках.

**Вывод типа по аргументам**:

max(5, 10); // T выводится как int

Компилятор автоматически определяет тип по аргументам.

**Специализация шаблона**:

template <>  
char\* max<char\*>(char\* a, char\* b) {  
 return (strcmp(a, b) > 0) ? a : b;  
}

Специальная реализация для конкретного типа.

**Частичная специализация** (только для классов):

template <typename T>  
class Container { /\* общая реализация \*/ };  
  
template <typename T>  
class Container<T\*> { /\* реализация для указателей \*/ };

**Параметры по умолчанию**:

template <typename T = int>  
class Array { /\* ... \*/ };

Если тип не указан, используется тип по умолчанию.