1. **Обобщенное программирование в C++** - это парадигма, позволяющая писать код, который работает с различными типами данных. Основные преимущества обобщенного программирования включают: уменьшение дублирования и возможность писать более универсальные и гибкие алгоритмы.

2. **Основная идея шаблонных функций в C++** заключается в создании функций, которые могут работать с любого типа данных, не определяя их заранее. Преимущества включают: уменьшение дублирования кода, возможность писать более общий и универсальный код, а также улучшение читаемости.

3. **Создание шаблонного класса в C++** выглядит следующим образом:

template<typename T\_key, typename T\_data>

class MyBinaryTree<T\_key, T\_data>::Node

{

private:

T\_key key;

T\_data data;

Node\* left = nullptr;

Node\* right = nullptr;

Node(const T\_key& key, const T\_data& value)

: key(key), data(value) {}

~Node()

{

if(left)

{

delete left;

left = nullptr;

}

if(right)

{

delete right;

right = nullptr;

}

}

};

int main()

{

MyBinaryTree<int, int> tree;

tree.push(1, 7);

tree.push(2, 3);

tree.push(-5, 7);

tree.push(56, 2);

return 0;

}

4. **Структура данных стек** - это коллекция элементов, работающая по принципу "последний пришел - первый вышел" (LIFO). Основные операции:

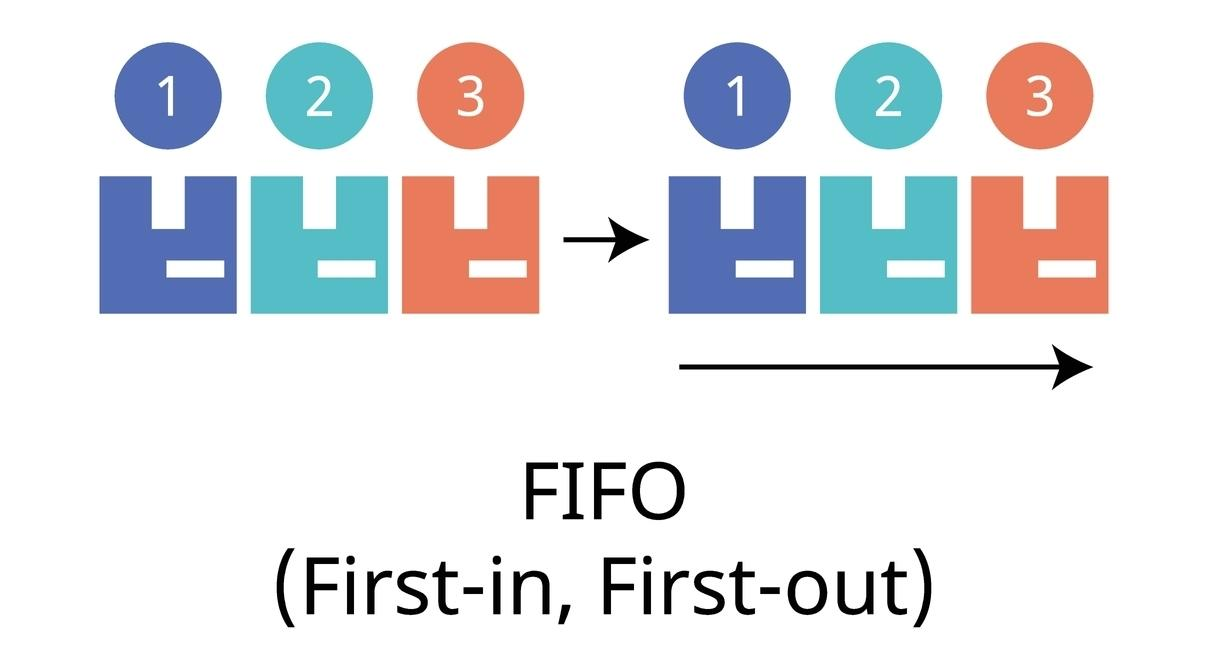
push - добавление элемента на вершину стека,

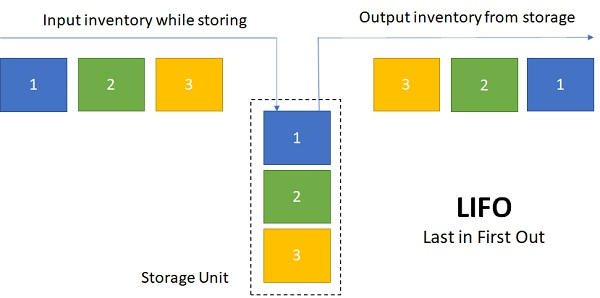
pop - удаление элемента с вершины стека,

top - получение значения элемента на вершине стека без его удаления,

isEmpty - проверка, пуст ли стек.

5. **Очередь** и **стек** отличаются принципами работы: очередь работает по принципу "первый пришел - первый вышел" (FIFO), а стек - LIFO. Пример ситуации, когда очередь предпочтительнее: обработка запросов на сервере, где запросы должны обслуживаться в порядке их поступления.





6. Операции, которые можно выполнять над **двусторонней очередью (деком) в C++**:

push\_front и push\_back - добавление элемента в начало или конец,

pop\_front и pop\_back - удаление элемента из начала или конца,

front и back - получение значения элемента в начале или конце,

isEmpty - проверка, пуст ли деки.

7. **Принцип "DRY" (Don't Repeat Yourself)** ставит цель избегать дублирования кода, что способствует его поддерживаемости и уменьшает вероятность ошибок. Использование шаблонов в C++ позволяет реализовать этот принцип, позволяя писать обобщённые функции и классы, которые могут работать с разными типами данных без дублирования кода.

8. **Концепции (concepts) в C++20** позволяют ограничивать набор типов, с которыми может работать шаблон, что помогает избежать ошибок времени компиляции и делает код более понятным. Преимущества: улучшают читаемость кода, упрощают диагностику ошибок, но могут усложнить реализацию. Ограничения могут заключаться в совместимости с старыми версиями C++.

#include <iostream>

#include <concepts>

// Определение концепции "Addable", которая требует наличия операции сложения

template<typename T>

concept Addable = requires(T a, T b) {

{ a + b } -> std::same\_as<T>;

};

template<Addable T>

T add(T a, T b) {

return a + b;

}

int main() {

std::cout << add(2, 3) << std::endl;

std::cout << add(2.5, 3.5) << std::endl;

// std::cout << add("Hello, ", "world!"); // Ошибка компиляции, строки не поддерживают операцию сложения

return 0;

}

10. **Использование обобщенного программирования и шаблонов** в больших проектах улучшает читаемость и поддерживаемость кода, позволяя создать универсальные решения, которые можно повторно использовать, избегая дублирования. Это упрощает модификацию и рефакторинг кода.

11. Специализация шаблонов требуется, когда необходимо реализовать особое поведение для конкретных типов данных. Специализированный шаблон создается следующим образом:

template <>

class Stack<bool> {

private:

std::vector<bool> elements; // Специфическая реализация для bool

public:

// Специфические методы

};

12. **std::any** - это контейнер, который может содержать любое значение, тогда как **std::variant** - это тип, который может принимать значения из заранее определенного набора типов. Они помогают в обобщенном программировании, позволяя работать с различными типами данных без необходимости использования явных указателей или базовых классов.

int main() {

std::any value;

value = 10;

std::cout << "Значение: " << std::any\_cast<int>(value) << std::endl;

value = std::string("Hello, world!");

std::cout << "Значение: " << std::any\_cast<std::string>(value) << std::endl;

}

int main() {

// Определяем std::variant, который может быть либо int, либо std::string

std::variant<int, std::string> var;

var = 42;

std::cout << "Значение: " << std::get<int>(var) << std::endl;

var = std::string("Привет, мир!");

std::cout << "Значение: " << std::get<std::string>(var) << std::endl;

}

13. **Шаблоны по умолчанию для типовых параметров** позволяют задать значение параметра, если оно не указано. Пример:

template <typename T = int>

class Example {

T value;

};

Здесь *Example* по умолчанию будет работать с типом *int*, если не указано иное.

14. В C++ ключевые слова typename и class могут использоваться в похожих контекстах, например, при определении шаблонов, и иногда они могут быть взаимозаменяемыми. Однако между ними есть некоторые различия в использовании и контексте.

### Основные различия

1. **Использование в шаблонах**:
   * typename:
     + typename используется для указания на параметр шаблона типа. Это указывает компилятору, что следующее имя является типом. Например:

* template<typename T>

void func(T arg) {

// ...

}

* class:
* class также может использоваться для указания на параметр шаблона типа, и в этом контексте он считается синонимом typename. Например:

* + template<class T>

void func(T arg) {

// ...

}

В контексте определения параметров шаблона typename и class эквивалентны:

* template<typename T> // или template<class T>

void func(T arg);

* **Использование в контексте зависимых имен**:
* Здесь typename необходимо, чтобы указать, что имя, предшествующее которому указано как зависимое, является типом. Это важно для правильного разрешения имен в шаблонах и в других контекстах. Например:

* + template<typename T>

void func(typename T::type arg) {

// ...

}

В этом примере typename указывает, что T::type является типом.

* + Если бы вы использовали class в этом контексте, был бы произведен вывод компилятора о синтаксической ошибке, так как class не подходит для этого использования.