**Белорусский государственный технологический университет**

**Факультет информационных технологий**

**Кафедра программной инженерии**

**Реферат**

По дисциплине «Основы алгоритмизации и программирования»

На тему «Различие сложных типов данных в программировании»

Выполнил:

Студент 1 курса 10 группы ПИ

Макаревич Кирилл Витальевич

Проверил: Белодед Николай Иванович

2024, Минск

**Оглавление**

**Введение3**

**Различия сложных типов данных4**

Эволюция языков программирования6

**Принцип фон Неймана7**

**Отличие раннего и позднего связывания 8**

**Конструкции try/catch10**

**Вывод по темам 12**

**Введение**

После того, как преподавателем была одобрена тема на реферат “Различие сложных типов данных ”, я сразу принялся искать материал и писать реферат на эту тему. Рассмотрю вопросы, которые были упомянуты в оглавлении.

**Различия сложных типов данных**

**Сложные типы данных**- это типы данных, которые состоят из других, более простых типов данных. В отличие от **простых типов данных** (например: int, char, float), которые могут хранить только одно значение (например, одно число или одну букву), сложные типы могут хранить много разных значений одновременно.

В C++ есть разные способы хранить данные, которые называются сложными типами данных. Вот самые важные из них:

**1. Массивы (Arrays)**

* **Что это**: Массив — это набор данных одного типа (например, набор чисел или букв).
* **Как работает**: Представьте, что это коробка с ячейками, где каждая ячейка хранит одно значение. Все значения — одного типа (например, все числа или все символы).
* **Пример**: Если у вас есть 5 чисел, ты можешь хранить их в массиве.

**Пример кода**:

int numbers[5] = { 1, 2, 3, 4, 5 }; // Массив из 5 чисел

**2. Структуры (Structures)**

* **Что это:** Структура позволяет хранить разные типы данных вместе.
* **Как работает:** Представьте, что структура — это как ящик с разными отделениями, и в каждом отделении лежит что-то своё.Например, вы можете создать структуру, которая хранит информацию о человеке: имя (текст) и возраст (число).

**Пример кода:**

struct Person {

std::string name; // Имя (текст)

int age; // Возраст (число)

};

**3. Классы (Classes)**

* **Что это**: Классы — это как структуры, но у них больше возможностей. Они могут не только хранить данные, но и выполнять действия (функции).
* **Как работает**: Класс — это как «чертёж» для создания объектов. Представьте, что вы делаете чертёж машины: ты можешь описать её характеристики (скорость, цвет) и действия (например, ехать или тормозить).

**Пример кода**:

class Car {

public:

int speed; // переменная Скорость

void drive() { // переменная Действия "ехать"

// код для движения

}

};

**4. Контейнеры (Containers)**

* **Что это**: Контейнеры — это особые структуры для хранения множества элементов. Например, вектор — это контейнер, который похож на массив, но его размер можно изменять.
* **Как работает**: Вектор можно сравнить с эластичной коробкой. Ты можешь добавлять или удалять элементы, и он автоматически меняет свой размер.

**Пример кода**:

std::vector<int> numbers = { 1, 2, 3 }; // Вектор из чисел

numbers.push\_back(4); // Добавляем число 4 в конец

**5. Перечисления (Enums)**

* **Что это**: Перечисление — это набор именованных значений. Например, цвета светофора: красный, жёлтый, зелёный.
* **Как работает**: Перечисления удобно использовать, когда есть фиксированное количество вариантов. Например, у светофора только три цвета.

**Пример кода:**

enum TrafficLight { Red, Yellow, Green };

**6. Указатели и ссылки (Pointers and References)**

* **Что это:** Указатели — это переменные, которые хранят адреса других переменных. Ссылки — это как "ярлыки" для других переменных.
* **Как работает:** Представь, что указатель — это карта с указанием, где находится нужная вещь (адрес), а ссылка — это короткий путь к этой вещи. С помощью указателей и ссылок можно управлять тем, где данные хранятся в памяти компьютера.

**Пример кода:**

int x = 5; // инициализация переменной x

int\* ptr = &x; // Указатель на переменную x

int& ref = x; // Ссылка на переменную x

**7. Объединения (Unions)**

* **Что это**: Объединение — это переменная, которая может хранить разные типы данных, но только одно значение за раз.
* **Как работает**: Представь, что у тебя есть один ящик, в который ты можешь положить или число, или строку, но не оба сразу. Ты можешь использовать только одно из них в данный момент.

**Пример кода:**

union Data { // Объявление объединения Data

int i; // Поле для хранения целого числа

float f; // Поле для хранения числа с плавающей точкой

char str[20]; // Поле для хранения строки (массив из 20 символов)

};

**Эволюция языков программирования**:

**Первый этап: Машинные языки**

* Описание: Ориентированы на использование в конкретной ЭВМ, сложны в освоении, требуют хорошего знания архитектуры ЭВМ.
* Особенности:
  + Использование двоичных кодов (0 и 1).
  + Полная зависимость от аппаратного обеспечения.
  + Очень трудоемкие и сложные для понимания человеком.

**Второй этап: Ассемблеры, Макроассемблеры**

* Описание: Более удобны для использования, но по-прежнему машинно-зависимы.
* Особенности:
  + Используют мнемоники вместо двоичных кодов, что делает код более читаемым.
  + Требуют глубоких знаний архитектуры компьютера.
  + Позволяют создавать программы быстрее, чем на машинных языках, но всё ещё привязаны к конкретным типам процессоров.

**Третий этап: Языки высокого уровня**

* Описание: Мобильные, человеко-ориентированные, проще в освоении.
* Особенности:
  + Абстрагируются от конкретного оборудования.
  + Программы стали переносимыми между разными компьютерами.
  + Примеры: FORTRAN, COBOL, Pascal.
  + Позволяют писать программы, используя структуры данных и операторы, более понятные для людей.

**Четвертый этап: Непроцедурные, объектно-ориентированные языки, языки запросов, параллельные**

* Описание: Ориентированы на непрофессионального пользователя и на ЭВМ с параллельной архитектурой.
* Особенности:
  + Включают ООП (объектно-ориентированное программирование), где код строится вокруг объектов и их взаимодействий (например, C++, Java).
  + Развиваются непроцедурные языки, где программист описывает не процесс, а результат (например, SQL для работы с базами данных).
  + Появляются параллельные языки для работы на многопроцессорных системах.

**Пятый этап: Языки искусственного интеллекта, экспертных систем, баз знаний, естественные языки**

* Описание: Ориентированы на повышение интеллектуального уровня ЭВМ и интерфейса с языками.
* Особенности:
  + Языки для работы с искусственным интеллектом и машинным обучением (например, Lisp, Prolog, Python).
  + Развитие языков для создания экспертных систем и баз знаний.
  + Появление интерфейсов с использованием естественных языков, которые приближают взаимодействие с компьютером к человеческой речи.

Этот процесс эволюции отражает стремление сделать программирование более доступным для человека и более мощным с точки зрения возможностей машин.

**Принцип фон Неймана**

**Принцип фон Неймана** — это фундаментальная архитектура компьютеров, предложенная математиком Джоном фон Нейманом в 1945 году. Этот принцип заложил основу для большинства современных компьютеров и описывает, как система обработки данных должна быть организована.

**Основные компоненты принципа фон Неймана:**

1. **Единая память для данных и программ:**
   * В компьютере фон Неймана используется одна и та же память для хранения как данных, так и программ. Это упрощает управление памятью, так как программы могут модифицировать данные и наоборот.
2. **Последовательное выполнение инструкций:**
   * Программа состоит из инструкций, которые выполняются последовательно, по одной за раз. Программа загружается в память, а процессор выполняет инструкции одну за другой, начиная с первой.
3. **Центральный процессор (ЦП):**
   * Центральный процессор (CPU) выполняет все вычисления. Он состоит из двух основных компонентов:
     + Устройство управления (Control Unit) — отвечает за интерпретацию и выполнение команд.
     + Арифметико-логическое устройство (ALU) — выполняет математические и логические операции.
4. **Использование адресуемой памяти:**
   * Память организована как набор адресуемых ячеек, в которых хранятся данные и инструкции. Каждая ячейка памяти имеет уникальный адрес.
5. **Последовательность: ввод — обработка — вывод:**
   * Входные данные поступают через устройства ввода (клавиатура, сенсоры и т.д.), затем проходят обработку в процессоре, а результат выводится через устройства вывода (дисплей, принтер и т.д.).

**Схема компьютера по принципу фон Неймана:**

* *Ввод → Память ↔ ЦП (процессор) → Вывод*

**Преимущества принципа фон Неймана:**

* Гибкость: программы можно хранить в той же памяти, что и данные, что позволяет создавать универсальные компьютеры, способные выполнять разные задачи.
* Простота архитектуры: общая схема компьютера проста и позволяет эффективно обрабатывать данные.

**Недостатки:**

* Узкое место (бутылочное горлышко) фон Неймана: процессор и память разделены, и скорость передачи данных между ними ограничена, что замедляет работу системы, особенно при большом объеме данных.

**Отличие раннего и позднего связывания**

Когда программа выполняется, она должна понимать, какую именно функцию или метод нужно вызвать. В зависимости от того, когда программа решает, какую функцию использовать, различают раннее и позднее связывание.

**1**. **Раннее связывание (Early Binding)**

* **Что это такое?**  
  Раннее связывание происходит тогда, когда программа уже во время компиляции (до запуска программы) знает, какую функцию или метод ей нужно вызвать. Все решения принимаются заранее.
* **Когда используется?**  
  Это используется для функций, которые не меняются, и их вызов ясен сразу. Например, если у тебя есть простая функция, которая всегда делает одно и то же.
* **Преимущества:**
  + Работает быстрее, потому что программа не тратит время на выбор функции во время выполнения.
* **Недостатки:**
  + Меньше гибкости — сложно изменять поведение программы в зависимости от ситуации.

**Пример кода:**

class Animal { // Объявление базового класса Animal

public:

void speak() { // Обычный (не виртуальный) метод speak()

std::cout << "Animal speaks" << std::endl; // Вывод "Animal speaks" на экран

}

};

Animal animal; // Создание объекта класса Animal

animal.speak(); // Вызов метода speak() у объекта класса Animal.

// Программа уже на этапе компиляции знает, что нужно вызвать speak() именно у Animal

**2. Позднее связывание (Late Binding)**

* **Что это такое?**  
  Позднее связывание — это когда программа решает, какую функцию или метод вызывать уже во время работы программы. Это важно, когда объекты могут вести себя по-разному в зависимости от ситуации.
* **Когда используется?**Это используется, когда у тебя есть несколько похожих объектов, но каждый из них должен выполнять свои собственные действия. Например, у тебя может быть общий класс "Животное", но разные животные (собака, кошка) будут "говорить" по-разному.
* **Преимущества:**
  + Даёт большую гибкость — можно менять поведение программы во время её работы.
* **Недостатки:**
  + Работает немного медленнее, так как программа решает, какую функцию вызывать уже во время выполнения.

**Пример кода:**

class Animal { // Определение базового класса Animal

public:

virtual void speak() { // Виртуальный метод speak(), который можно будет переопределить в наследниках

std::cout << "Animal speaks" << std::endl; // Вывод "Animal speaks" на экран по умолчанию

}

};

class Dog : public Animal { // Определение класса Dog, который наследуется от Animal

public:

void speak() override { // Переопределение виртуального метода speak() для класса Dog

std::cout << "Dog barks" << std::endl; // Вывод "Dog barks" на экран при вызове метода у объекта Dog

}

};

Animal\* animal = new Dog(); // Создание указателя на объект Dog, но с типом указателя Animal (полиморфизм)

animal->speak(); // Вызов метода speak() у объекта, на который указывает указатель animal

// Программа решает, что нужно вызвать переопределённый метод speak() у Dog

**Основные различия:**

* **Раннее связывание:**
  + Происходит до запуска программы.
  + Быстрее, потому что всё известно заранее.
  + Подходит для простых случаев, когда нет необходимости менять поведение программы.
* **Позднее связывание:**
  + Происходит во время работы программы.
  + Даёт больше гибкости, так как можно выбирать функции или методы в зависимости от ситуации.
  + Полезно, когда ты работаешь с полиморфизмом (разные объекты ведут себя по-разному).

**Конструкции try/catch**

Конструкции try/catch в C++ помогают программе **обрабатывать ошибки**, которые могут возникнуть во время её выполнения. Они нужны для того, чтобы программа **не завершалась с ошибкой**, а справлялась с проблемами, продолжая работать.

**Как это работает:**

1. **Блок try**:
   * Внутри блока try пишется код, который **может вызвать ошибку**.
   * Программа будет "следить" за этим кодом, и если произойдет ошибка, она перейдёт к блоку catch.
2. **Блок catch**:
   * Здесь программа **реагирует на ошибку**. Ты можешь указать, что делать, если возникла проблема: вывести сообщение или исправить её.

**Пример кода:**

#include <iostream> // Подключение библиотеки для ввода-вывода

int main() { // Начало функции main

setlocale(LC\_ALL, "ru"); // поддержка русского языка

try { // Начало блока try, где будет проверяться код на ошибки

int a = 10; // Инициализация переменной a значением 10

int b = 0; // Инициализация переменной b значением 0

if (b == 0) { // Проверка, если b равно 0

throw "Деление на ноль"; // Генерируем исключение, так как нельзя делить на ноль

}

int c = a / b; // Пытаемся выполнить деление a на b (но это не выполнится из-за throw выше)

std::cout << "Результат: " << c << std::endl; // Выводим результат деления (этот код не выполнится)

}

catch (const char\* e) { // Ловим исключение, если оно возникло

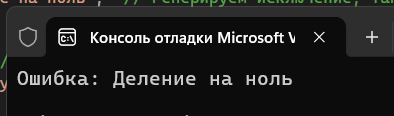
std::cout << "Ошибка: " << e << std::endl; // Выводим сообщение об ошибке (в данном случае "Деление на ноль")

}

return 0; // Завершаем программу

}

**Результат кода:**

****

**Что происходит в коде:**

1. В блоке try программа пытается разделить число a на b.
2. Но поскольку b = 0, программа не может выполнить деление на ноль. Это вызывает ошибку.
3. С помощью throw программа выбрасывает исключение с текстом "Деление на ноль".
4. Программа переходит в блок catch, где эта ошибка "ловится" и выводится сообщение "Ошибка: Деление на ноль".

**Главное:**

* **try** — проверяет код, в котором может возникнуть ошибка.
* **throw** — сообщает о проблеме, когда ошибка найдена.
* **catch** — ловит ошибку и решает, что делать дальше.

**Вывод по темам**

* 1. **Различия сложных типов данных**

Сложные типы данных, такие как массивы, структуры, классы и контейнеры STL, играют важную роль в программировании. Каждый тип данных имеет свои особенности и предназначение. Понимание различий между ними помогает разработчикам выбирать подходящие структуры для решения конкретных задач, обеспечивая эффективность и удобство работы с данными.

* 1. **Эволюция языков программирования**

Языки программирования прошли значительную эволюцию, начиная с машинного кода и ассемблера, до высокоуровневых языков, которые упрощают процесс разработки. Развитие языков привело к созданию объектно-ориентированных и функциональных подходов, позволяющих разработчикам писать более сложные и гибкие программы, что значительно улучшает взаимодействие между человеком и компьютером.

* 1. **Принцип фон Неймана**

Принцип фон Неймана является основополагающим в архитектуре современных компьютеров. Он описывает, как данные и инструкции хранятся в памяти и обрабатываются центральным процессором. Этот принцип позволяет создавать универсальные и многофункциональные вычислительные системы, которые легки в программировании и использовании.

* 1. **Отличие раннего и позднего связывания**

Раннее и позднее связывание определяют, как программа выбирает, какую функцию или метод использовать. Раннее связывание происходит до выполнения программы и обеспечивает большую скорость, но меньшую гибкость. Позднее связывание, наоборот, позволяет динамически выбирать методы во время выполнения, что повышает гибкость, но может замедлять выполнение программы.

* 1. **Конструкции try/catch**

Конструкции try/catch обеспечивают обработку ошибок в C++. Они позволяют программе не завершаться с ошибкой, а реагировать на возникшие проблемы, тем самым улучшая стабильность и надежность программного кода. Это важно для создания пользовательских приложений, где необходимо управлять исключительными ситуациями и информировать пользователя о произошедших ошибках.