1) Функция в языке программирования C++ — это независимый участок кода, который выполняет определенную задачу.

- Определение функции:

тип\_возвращаемого\_значения имя\_функции(параметры) {

// код функции

return значение;

}

Пример:

#include <iostream>

using namespace std;

int add(int a, int b) {

return a + b;

}

int main() {

int result = add(5, 3);

cout << "Результат сложения: " << result << endl;

return 0;

}

2) Способы передачи параметров в функции в C++: по значению, по указателю, по ссылке.

3) Передача параметров в функцию по значению означает, что значения переменных копируются в параметры функции.

Пример:

void square(int x) {

x = x \* x;

}

int main() {

int num = 5;

square(num);

cout << "Число: " << num << endl; // Выводит 5, так как num не изменилось

return 0;

}

4) Передача параметров в функцию по ссылке позволяет изменять значения переданных переменных в функции.

Пример:

void square(int &x) {

x = x \* x;

}

int main() {

int num = 5;

square(num);

cout << "Число: " << num << endl; // Выводит 25, так как num изменилось

return 0;

}

5) Передача массивов в функцию в C++ может быть выполнена двумя способами: с помощью указателя на первый элемент массива или с помощью ссылки на массив. При передаче массива как параметра функции, его размер также может быть передан явно.

Пример передачи массива с использованием указателя:

#include <iostream>

using namespace std;

void printArray(int arr[], int size) {

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

cout << arr[i] << " ";

}

cout << endl;

}

int main() {

int numbers[] = {1, 2, 3, 4, 5};

int size = 5;

printArray(numbers, size);

return 0;

}

Работа с двумерными массивами происходит похожим образом, но требуется указать количество строк и столбцов.

6) Параметры по умолчанию в C++ позволяют задать значения по умолчанию для параметров функций. Если при вызове функции значения для этих параметров не переданы, будет использовано значение по умолчанию.

Пример:

#include <iostream>

using namespace std;

void greet(string name = "Гость") {

cout << "Привет, " << name << "!" << endl;

}

int main() {

greet(); // Выводит: Привет, Гость!

greet("Джон"); // Выводит: Привет, Джон!

return 0;

}

7) Механизм перегрузки функций в C++ позволяет создавать несколько функций с одинаковым именем, но разными параметрами. Компилятор на основе переданных параметров вызывает соответствующую функцию.

Пример:

#include <iostream>

using namespace std;

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

int main() {

cout << add(5, 3) << endl; // Выводит 8

cout << add(2.5, 3.5) << endl; // Выводит 6.0

return 0;

}

8) Шаблоны функций в C++ позволяют создавать обобщенные функции, которые могут работать с различными типами данных, не указывая их явно.

Пример:

#include <iostream>

using namespace std;

template <typename T>

T max(T a, T b) {

return (a > b) ? a : b;

}

int main() {

cout << max(5, 3) << endl; // Выводит 5

cout << max(2.5, 3.5) << endl; // Выводит 3.5

return 0;

}

9) Заголовочные файлы в C++ используются для подключения объявлений функций, классов и переменных, которые используются в программе. Заголовочные файлы имеют расширение .h и содержат прототипы функций и объявления классов.

Пример использования заголовочных файлов:

// Файл: operations.h

#ifndef OPERATIONS\_H

#define OPERATIONS\_H

int add(int a, int b);

int subtract(int a, int b);

#endif

// Файл: main.cpp

#include <iostream>

#include "operations.h"

int main() {

int result1 = add(5, 3);

int result2 = subtract(7, 2);

std::cout << "Результат сложения: " << result1 << std::endl;

std::cout << "Результат вычитания: " << result2 << std::endl;

return 0;

}

При компиляции программы все исходные файлы с заголовочными файлами объединяются и компилируются вместе.

10) Класс памяти auto в C++ указывает на автоматическую продолжительность жизни переменной, что означает, что переменная будет создаваться при входе в блок кода, где она объявлена, и уничтожаться при выходе из этого блока кода. Переменные типа auto являются локальными для блока, в котором они объявлены, и не инициализируются по умолчанию.

Пример:

void func() {

auto int x = 5;

// x существует только в пределах этой функции и инициализируется при входе в функцию

}

11) Класс памяти register в C++ используется для объявления переменной, которая будет храниться в регистре процессора вместо оперативной памяти. Однако ключевое слово register является рекомендацией для компилятора, и компилятор может проигнорировать его, если считает, что использование регистра неэффективно или невозможно. Переменные типа register обычно используются для часто используемых данных, чтобы повысить производительность программы.

Пример:

int main() {

register int x = 10;

// Переменная x может быть помещена в регистр процессора для более быстрого доступа

return 0;

}

12) Класс памяти static в C++ используется для создания переменных, которые сохраняют своё значение между вызовами функций или видимости блока. Статические переменные инициализируются только один раз, и их значение сохраняется до конца программы. Они также имеют область видимости, ограниченную блоком, в котором они объявлены.

Пример:

void func() {

static int count = 0;

count++;

std::cout << "Вызов функции: " << count << std::endl;

}

int main() {

func(); // Выводит: Вызов функции: 1

func(); // Выводит: Вызов функции: 2

return 0;

}

13) Класс памяти extern в C++ используется для объявления переменных или функций, которые могут быть определены в другом файле. Переменные или функции, объявленные как extern, могут использоваться в других файлах без необходимости повторного определения. Переменные extern объявляются без их фактического определения, и определение их должно быть представлено в другом файле.

Пример:

// Файл: extern\_example.cpp

extern int globalVar; // Объявление переменной globalVar из другого файла

// Файл: main.cpp

#include <iostream>

extern int globalVar = 10; // Определение переменной globalVar

int main() {

std::cout << "Глобальная переменная: " << globalVar << std::endl;

return 0;

}

Это позволяет создавать разделяемые данные между различными модулями программы.

14) Работа с файлами в C++ предполагает использование потоков для ввода и вывода данных в файлы. Для работы с файлами в C++ используются классы std::ifstream, std::ofstream и std::fstream из стандартной библиотеки <fstream>. std::ifstream используется для чтения данных из файла, std::ofstream для записи данных в файл, а std::fstream для чтения и записи данных.

Пример чтения данных из файла:

#include <iostream>

#include <fstream>

int main() {

std::ifstream file("example.txt");

if (file.is\_open()) {

std::string line;

while (std::getline(file, line)) {

std::cout << line << std::endl;

}

file.close();

} else {

std::cout << "Не удалось открыть файл." << std::endl;

}

return 0;

}

15) Заголовочные файлы в C++ содержат объявления функций, классов, переменных и других объектов, которые можно использовать в программе. Заголовочные файлы обычно имеют расширение .h или .hpp, и их подключают в программу с помощью директивы #include. Заголовочные файлы используются для разделения кода на модули, упрощения работы с большими проектами и уменьшения времени компиляции.

Пример заголовочного файла:

// Файл: example.h

#ifndef EXAMPLE\_H

#define EXAMPLE\_H

void sayHello();

#endif

16) Процесс компиляции состоит из следующих этапов:

1)лексический анализ;

2) синтаксический анализ;

3) семантический анализ;

4)оптимизация;

5) генерация кода.

Лексический анализ. Последовательность символов исходного файла преобразуется в последовательность лексем. Лексема – это элементарная составляющая языка, несущая смысловую нагрузку, например: имя, ключевое слово, символ операции, разделитель.

Синтаксический анализ. В процессе синтаксического анализа из лексем собирают выражения, а из выражений – операторы. Последовательность терминальных символов преобразуется в нетерминалы.

Семантический анализ – проверка смысловой правильности синтаксических конструкций. Например, если в выражении используется переменная, то она должна быть предварительно объявлена и проинициализирована

Оптимизация – упрощение кода с сохранением его смысла и удаление лишних конструкций.

Генерация кода. На этом этапе происходит замена операторов языка высокого уровня инструкциями ассемблера, а затем – последовательностью машинных команд. Результат преобразования исходного текста программы записывается в виде двоичного файла, имеющего расширение \*.obj, который называют объектным модулем.17) Режимы компиляции в C++ включают два основных режима: режим сборки и режим разработки. Режим сборки предназначен для создания исполняемого файла из исходного кода программы при помощи компилятора. Режим разработки предназначен для удобной работы с исходным кодом, включая отладку, тестирование и исправление ошибок.

Для компиляции программ на C++ используются различные компиляторы, такие как GCC, Clang, Visual C++ и другие. Компиляторы могут принимать различные аргументы и флаги для управления процессом сборки. Например, для компиляции файла main.cpp в исполняемый файл main.exe с помощью компилятора GCC можно использовать следующую команду:

g++ -o main.exe main.cpp

Совместная компиляция. При совместной компиляции на вход компилятору подаётся один исходный файл, содержащий в себе все исходные модули, подключённые через директиву #include. Далее полученный obj-файл обрабатывается компоновщиком, который, в свою очередь, собирает exe-файл. Модули с исходным кодом, подключённые с помощью директивы #include в модуль, содержащий main функцию, из проекта необходимо исключить (remove). Иначе возникнет ошибка, связанная с переопределением.

Раздельная компиляция. Все модули компилируются независимо друг от друга, генерируется столько obj-файлов, сколько было модулей с исходным кодом. Во время компоновки все obj-файлы собираются в один exe-файл. Для успешной компоновки необходимо, чтобы во всех компонуемых файлах содержались объявление всех используемых функций (как пользовательских, так и стандартных) и определение внешних переменных (объявленных с модификатором extern). В противном случае возникнет ошибка компоновки. Для реализации вышесказанного необходимо подключение через директиву #include заголовочных файлов во все модули, где планируется вызов объявленных в них функций.

1) Алгоритм быстрой сортировки (Quick Sort) - это эффективный алгоритм сортировки данных. Он работает по принципу разделяй и властвуй. Алгоритм быстрой сортировки в общем виде выглядит следующим образом:

- Выбирается опорный элемент из массива.

- Массив разбивается на две подмассива: один содержит элементы меньше опорного, другой содержит элементы больше опорного.

- Рекурсивно применяется алгоритм к каждому из двух подмассивов.

2) Рекурсивный алгоритм добавления элемента в двоичное дерево:

struct Node {

int data;

Node \*left, \*right;

};

Node\* insert(Node\* root, int value) {

if (root == nullptr) {

Node\* newNode = new Node;

newNode->data = value;

newNode->left = newNode->right = nullptr;

return newNode;

} else {

if (value < root->data) {

root->left = insert(root->left, value);

} else if (value > root->data) {

root->right = insert(root->right, value);

}

return root;

}

}

3) Рекурсивный алгоритм LCR обхода двоичного дерева (ин-ордер обход):

void LCR\_Traversal(Node\* root) {

if (root != nullptr) {

LCR\_Traversal(root->left);

cout << root->data << " ";

LCR\_Traversal(root->right);

}

}

4) Рекурсивный алгоритм CLR обхода двоичного дерева (pre-order обход):

void CLR\_Traversal(Node\* root) {

if (root != nullptr) {

cout << root->data << " ";

CLR\_Traversal(root->left);

CLR\_Traversal(root->right);

}

}

5) Рекурсивный алгоритм LRC обхода двоичного дерева (пост-ордер обход):

void LRC\_Traversal(Node\* root) {

if (root != nullptr) {

LRC\_Traversal(root->left);

LRC\_Traversal(root->right);

cout << root->data << " ";

}

}

6) Алгоритм добавления элемента в линейную динамическую структуру, организованную по принципу стека:

// Предполагается, что стек реализован с помощью std::vector

void push(std::vector<int>& stack, int value) {

stack.push\_back(value);

}

7) Алгоритм добавления элемента в линейную динамическую структуру, организованную по принципу очереди:

// Предполагается, что очередь реализована с помощью std::queue

void enqueue(std::queue<int>& queue, int value) {

queue.push(value);

}

8) Алгоритм извлечения элемента из линейной динамической структуры, организованной по принципу стека:

void pop(std::vector<int>& stack) {

if (!stack.empty()) {

stack.pop\_back();

}

}

9) Алгоритм извлечения элемента из линейной динамической структуры, организованной по принципу очереди:

void dequeue(std::queue<int>& queue) {

if (!queue.empty()) {

queue.pop();

}

}

10) Алгоритм перебора элементов линейной динамической структуры, организованной по принципу очереди:

void traverseQueue(std::queue<int> queue) {

while (!queue.empty()) {

cout << queue.front() << " ";

queue.pop();

}

}

11) Алгоритм добавления элемента в конец двунаправленного линейного списка:

```cpp

void appendToEnd(Node\*\* head\_ref, int value) {

Node\* new\_node = new Node;

new\_node->data = value;

new\_node->next = nullptr;

if (\*head\_ref == nullptr) {

new\_node->prev = nullptr;

\*head\_ref = new\_node;

} else {

Node\* temp = \*head\_ref;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = new\_node;

new\_node->prev = temp;

}

}

```

12) Алгоритм добавления элемента в конец двунаправленного кольцевого линейного списка:

```cpp

void appendToEndCircular(Node\*\* head\_ref, int value) {

Node\* new\_node = new Node;

new\_node->data = value;

if (\*head\_ref == nullptr) {

\*head\_ref = new\_node;

new\_node->next = new\_node;

new\_node->prev = new\_node;

} else {

Node\* prev\_node = (\*head\_ref)->prev;

new\_node->next = \*head\_ref;

new\_node->prev = prev\_node;

prev\_node->next = new\_node;

(\*head\_ref)->prev = new\_node;

}

}

```

13) Алгоритм добавления элемента в начало двунаправленного линейного списка:

```cpp

void addToBegin(Node\*\* head\_ref, int value) {

Node\* new\_node = new Node;

new\_node->data = value;

new\_node->next = \*head\_ref;

new\_node->prev = nullptr;

if (\*head\_ref != nullptr) {

(\*head\_ref)->prev = new\_node;

}

\*head\_ref = new\_node;

}

```

14) Алгоритм добавления элемента в начало двунаправленного кольцевого линейного списка:

```cpp

void addToBeginCircular(Node\*\* head\_ref, int value) {

Node\* new\_node = new Node;

new\_node->data = value;

if (\*head\_ref == nullptr) {

\*head\_ref = new\_node;

new\_node->next = new\_node;

new\_node->prev = new\_node;

} else {

Node\* prev\_node = (\*head\_ref)->prev;

new\_node->next = \*head\_ref;

new\_node->prev = prev\_node;

prev\_node->next = new\_node;

(\*head\_ref)->prev = new\_node;

\*head\_ref = new\_node;

}

}

```

15) Алгоритм удаления элемента по ключу из однонаправленного линейного списка:

```cpp

void deleteByKey(Node\*\* head\_ref, int key) {

Node\* temp = \*head\_ref;

Node\* prev = nullptr;

if (temp != nullptr && temp->data == key) {

\*head\_ref = temp->next;

delete temp;

return;

}

while (temp != nullptr && temp->data != key) {

prev = temp;

temp = temp->next;

}

if (temp == nullptr) return;

prev->next = temp->next;

delete temp;

}

```

16) Алгоритм удаления элемента по ключу из двунаправленного линейного списка:

```cpp

void deleteByKeyTwoWay(Node\*\* head\_ref, int key) {

Node\* temp = \*head\_ref;

if (temp != nullptr && temp->data == key) {

\*head\_ref = temp->next;

(\*head\_ref)->prev = nullptr;

delete temp;

return;

}

while (temp != nullptr && temp->data != key) {

temp = temp->next;

}

if (temp == nullptr) return;

if (temp->next != nullptr) {

temp->next->prev = temp->prev;

}

temp->prev->next = temp->next;

delete temp;

}

```

17) Алгоритм вставки элемента в однонаправленный линейный список после элемента с указанным значением ключевого поля или в конец списка, если таких значений в списке нет:

```cpp

void insertAfter(Node\* prev\_node, int value) {

if (prev\_node == nullptr) {

cout << "Previous node cannot be null";

return;

}

Node\* new\_node = new Node;

new\_node->data = value;

new\_node->next = prev\_node->next;

prev\_node->next = new\_node;

}

```

18) Алгоритм вставки элемента в двунапраленный линейный список после элемента с указанным значением ключевого поля или в конец списка, если таких значений в списке нет:

```cpp

void insertAfterTwoWay(Node\*\* head\_ref, int key, int value) {

Node\* temp = \*head\_ref;

while (temp != nullptr && temp->data != key) {

temp = temp->next;

}

if (temp == nullptr) {

appendToEnd(head\_ref, value);

} else {

Node\* new\_node = new Node;

new\_node->data = value;

new\_node->next= temp->next;

new\_node->prev = temp;

if (temp->next != nullptr) {

temp->next->prev = new\_node;

}

temp->next = new\_node;

}

}

```

19) Алгоритм вставки элемента в однонаправленный линейный список перед элементом с указанным значением ключевого поля или в начало списка, если таких значений в списке нет:

```cpp

void insertBefore(Node\*\* head\_ref, int key, int value) {

if (\*head\_ref == nullptr) {

return;

}

if ((\*head\_ref)->data == key) {

addToBegin(head\_ref, value);

return;

}

Node\* current = \*head\_ref;

while (current->next != nullptr && current->next->data != key) {

current = current->next;

}

if (current->next == nullptr) {

return;

}

Node\* new\_node = new Node;

new\_node->data = value;

new\_node->next = current->next;

current->next = new\_node;

}

```

20) Алгоритм вставки элемента в двунаправленный линейный список перед элементом с указанным значением ключевого поля или в начало списка, если таких значений в списке нет:

```cpp

void insertBeforeTwoWay(Node\*\* head\_ref, int key, int value) {

if (\*head\_ref == nullptr) {

return;

}

if ((\*head\_ref)->data == key) {

addToBegin(head\_ref, value);

return;

}

Node\* current = \*head\_ref;

while (current->next != nullptr && current->next->data != key) {

current = current->next;

}

if (current->next == nullptr) {

return;

}

Node\* new\_node = new Node;

new\_node->data = value;

new\_node->next = current->next;

new\_node->prev = current;

if (current->next != nullptr) {

current->next->prev = new\_node;

}

current->next = new\_node;

}

```

21) Алгоритм поиска в двоичном дереве (возвращает адрес элемента):

```cpp

Node\* search(Node\* root, int key) {

if (root == nullptr || root->data == key) {

return root;

}

if (root->data < key) {

return search(root->right, key);

}

return search(root->left, key);

}

```