Определение и объявление функций
Функция — это именованный блок кода, который выполняет определенную задачу. Объявление функции (прототип) сообщает компилятору о существовании функции, ее имени, возвращаемом типе и параметрах. Определение функции содержит фактическую реализацию кода.

```
// Объявление функции (прототип)
int sum(int a, int b);
// Определение функции
int sum(int a, int b) {
return a + b;
}
int main() {
int result = sum(5, 3); // Вызов функции
return 0;
}
```

Параметры функций
Параметры — это переменные, которые принимают значения аргументов, передаваемых в функцию. Они определяют интерфейс функции и тип данных, с которым она работает.

```
// Функция с параметрами разных типов
void printData(int number, double value, const char* text) {
std::cout << "Number: " << number
<< ", Value: " << value
<< ", Text: " << text << std::endl;
}
int main() {
printData(10, 3.14, "Hello"); // Передача аргументов
return 0;
}
```

Передача аргументов по ссылке и по значению

При передаче по значению создается копия аргумента, изменения не влияют на оригинал. При передаче по ссылке функция работает непосредственно с оригинальной переменной, что позволяет изменять ее значение и избежать копирования.

```
// Передача по значению (копирование)
void incrementByValue(int x) {
x++; // Изменяется копия
}
// Передача по ссылке (работа с оригиналом)
void incrementByReference(int &x) {
x++; // Изменяется оригинал
}
int main() {
int a = 5;
incrementByValue(a); // a останется 5
incrementByReference(a); // a станет 6
return 0;
}
```

Константные параметры
Ключевое слово const гарантирует, что параметр не будет изменен внутри функции. Это обеспечивает безопасность данных и явно показывает намерения программиста.

```
// Константная ссылка - эффективно и безопасно
void printLargeObject(const std::string &str) {
// str.push_back('!'); // Ошибка: нельзя изменить константный параметр
std::cout << str << std::endl;
}
// Константный указатель
void processArray(const int* arr, int size) {
// arr[0] = 10; // Ошибка: данные защищены от изменений
for (int i = 0; i < size; i++) {
std::cout << arr[i] << " ";
}
}
int main() {
std::string text = "Large text content";
printLargeObject(text);
int numbers[] = {1, 2, 3};
processArray(numbers, 3);
}
```

Указатели и массивы в параметрах
В C++ массивы передаются в функции через указатели на первый элемент. Это означает, что функция получает адрес начала массива, а не копию всего массива. Такая передача эффективна по памяти, но позволяет функции изменять оригинальный массив.

```
#include <iostream>
// Способ 1: Явный указатель
void processArray1(int* arr, int size) {
for (int i = 0; i < size; i++) {
arr[i] *= 2; // Изменяем оригинальный массив
}
}
// Способ 2: Синтаксис массива (компилятор преобразует в указатель)
void processArray2(int arr[], int size) {
for (int i = 0; i < size; i++) {
arr[i] += 5;
}
}
// Способ 3: Массив с явным размером (размер только для документации)
void processArray3(int arr[5]) {
for (int i = 0; i < 5; i++) {
arr[i] -= 1;
}
// Многомерные массивы
void process2DArray(int matrix[][3], int rows) {
for (int i = 0; i < rows; i++) {
for (int j = 0; j < 3; j++) {
matrix[i][j] *= 10;
}
```

```
}
int main() {
int numbers[] = {1, 2, 3, 4, 5};
int size = sizeof(numbers) / sizeof(numbers[0]);
processArray1(numbers, size);
processArray2(numbers, size);
processArray3(numbers);
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
process2DArray(matrix, 2);
return 0;
}
```

Возвращение указателей и ссылок

Функции могут возвращать указатели и ссылки, что позволяет создавать гибкие интерфейсы. Однако при этом возникает ответственность за управление временем жизни объектов. Возвращать ссылки/указатели на локальные переменные нельзя - они уничтожаются при выходе из функции.

```
#include <iostream>
#include <vector>
// ОПАСНО: возврат указателя на локальную переменную
int* badFunction() {
int x = 10; // Локальная переменная
return &x; // ОШИБКА: x уничтожится после return!
}
// БЕЗОПАСНО: возврат указателя на статическую переменную
int* getStaticValue() {
static int counter = 0; // Статическая переменная
counter++;
return &counter; // Корректно - переменная существует всегда
}
// БЕЗОПАСНО: возврат указателя на динамическую память
int* createArray(int size) {
int* arr = new int[size];
for (int i = 0; i < size; i++) {
arr[i] = i * i;
}
return arr; // Память существует до явного удаления
}
// Возврат ссылки на существующий объект
int& getElement(std::vector<int>& vec, int index) {
return vec[index]; // Корректно - вектор существует вне функции
}
// Возврат ссылки на элемент массива
int& maxElement(int* arr, int size) {
int maxIndex = 0;
for (int i = 1; i < size; i++) {
if (arr[i] > arr[maxIndex]) {
maxIndex = i;
}
}
return arr[maxIndex]; // Возвращаем ссылку на элемент массива
}
int main() {
```

```
// Пример 1: Динамический массив
int* dynamicArray = createArray(5);
for (int i = 0; i < 5; i++) {
std::cout << dynamicArray[i] << " ";
}
delete[] dynamicArray; // Не забываем освободить память!
// Пример 2: Работа со ссылками
std::vector<int> numbers = {10, 20, 30, 40, 50};


int& ref = getElement(numbers, 2);
ref = 100; // Изменяет numbers[2]
std::cout << "\nnumbers[2] = " << numbers[2] << std::endl;
// Пример 3: Поиск максимального элемента
int arr[] = {5, 2, 8, 1, 9};
int& maxRef = maxElement(arr, 5);
maxRef = 1000; // Изменяем максимальный элемент
std::cout << "Modified array: ";
for (int val : arr) {
std::cout << val << " ";
}
return 0;
```

Важные правила:

1 Никогда не возвращайте указатели/ссылки на локальные переменные
2 Динамическую память нужно явно освобождать (delete/delete[])
3 Убедитесь, что объект существует дольше, чем ссылка/указатель на него
4 Используйте умные указатели для автоматического управления памятью

Объявление и определение функций

Объявление функции (прототип) сообщает компилятору о существовании функции, ее имени, возвращаемом типе и параметрах. Определение содержит фактическую реализацию. Функция может быть объявлена несколько раз, но определена только один раз.

```
// Объявление (прототип)
int calculate(int a, int b);
// Еще одно объявление (допустимо)
int calculate(int x, int y);
// Определение (реализация)
int calculate(int a, int b) {
return a * b;
```

Передача параметров по значению

При передаче по значению создается копия аргумента. Изменения параметра внутри функции не влияют на оригинальную переменную. Может быть неэффективно для больших объектов.

```
void modifyValue(int x) {
x = 100; // Изменяется копия
cout << "Inside function: " << x << endl;
}
int main() {
int num = 5;
```

```
modifyValue(num);
cout << "After function: " << num << endl; // Осталось 5
return 0;
}
```

Передача по ссылке
Передача по ссылке позволяет избежать копирования больших объектов, изменять оригинальные аргументы и экономит память. Параметр становится псевдонимом оригинальной переменной.

```
void modifyReference(int &x) {
x = 100; // Изменяется оригинал
}
void largeObjectPass(const vector<int> &vec) {
// Эффективная передача без копирования
cout << "Size: " << vec.size() << endl;
}
```

Константные ссылки
Константные ссылки используются для передачи больших объектов без копирования с гарантией, что функция не изменит переданный объект. Позволяют передавать временные объекты.

```
void processData(const string &str) {
// str нельзя изменить
cout << "Processing: " << str << endl;
}
void printVector(const vector<int> &vec) {
for (int val : vec) {
cout << val << " ";
}
}
```

Массивы как параметры
Массивы передаются по указателю на первый элемент. Информация о размере теряется, поэтому размер обычно передается отдельно. Не создается копия массива.

```
void processArray(int arr[], int size) {
for (int i = 0; i < size; i++) {
arr[i] *= 2; // Изменяет оригинальный массив
}
}
void arrayWithPointer(int *arr, int size) {
// Эквивалентно предыдущему
}
```

Ссылки на массивы
Ссылки на массивы сохраняют информацию о размере. Синтаксис требует круглых скобок вокруг имени ссылки и размера массива.

```
void processArrayRef(int (&arr)[5]) {
// Знает размер массива
```

```
for (int i = 0; i < 5; i++) {
cout << arr[i] << " ";
}
}
```

Возвращение ссылок
Возвращение ссылки безопасно когда объект существует после вызова функции: статические переменные, переданные параметры, глобальные переменные. Опасно для локальных переменных.

```
int& getStatic() {
static int value = 42;
return value; // Безопасно
}
int& getElement(int arr[], int index) {
return arr[index]; // Безопасно если массив существует
}
```

Параметры по умолчанию
Параметры по умолчанию должны быть указаны в объявлении функции и могут быть только у последних параметров. Не могут быть переопределены в определении.

```
// Объявление с параметрами по умолчанию
void draw(int x, int y, int color = 255, int size = 1);
// Определение
void draw(int x, int y, int color, int size) {
// Реализация
}
```

Перегрузка функций
Перегрузка происходит по количеству и типам параметров. Возвращаемый тип и модификаторы const для параметров не учитываются при перегрузке.

```
void print(int value) {
cout << "Integer: " << value << endl;
}
void print(double value) {
cout << "Double: " << value << endl;
}
void print(const string &text) {
cout << "String: " << text << endl;
}
```

Указатели на функции
Указатели на функции объявляются с указанием типа возвращаемого значения и типов параметров. Синтаксис требует круглых скобок вокруг имени указателя.

```
double calculate(int x); // Объявление функции
// Указатель на функцию
double (*funcPtr)(int) = calculate;
// Использование
double result = funcPtr(5);
```

Рекурсивные функции

Рекурсивные функции требуют условия завершения, каждый вызов должен приближать к базовому случаю. Могут потреблять много памяти стека.

```
int factorial(int n) {
if (n <= 1) return 1; // Базовый случай
return n * factorial(n - 1); // Рекурсивный вызов
}
```

Встраиваемые функции

Inline-функции - рекомендация компилятору вставить код функции вместо вызова. Полезны для небольших функций, но компилятор может проигнорировать спецификатор.

```
inline int square(int x) {
return x * x;
// Компилятор может преобразовать:
// int result = square(5); → int result = 5 * 5;
```

Параметры-массивы и указатели

Параметры массивов и указателей эквивалентны в объявлениях функций. Компилятор преобразует массив в указатель на первый элемент.

```
void func(int arr[]); // Эквивалентно
void func(int *arr); // этому
void func(int arr[10]); // Размер игнорируется
```

Возвращение указателей

Опасно возвращать указатели на локальные переменные или освобожденную память. Безопасно возвращать указатели на статические, глобальные переменные или динамическую память.

```
int* createArray(int size) {
int* arr = new int[size]; // Безопасно
return arr;
int* dangerous() {
int local = 5;
return &local; // ОПАСНО!
```

Строки как параметры

C-строки лучше передавать как const char*, std::string - по константной ссылке. Для модификации строк нужны неконстантные параметры.

```
void processCString(const char* str) {
cout << "C-string: " << str << endl;
}
void processStdString(const string &str) {
cout << "std::string: " << str << endl;
}
```

Параметры константных указателей

const int* ptr означает указатель на константные данные - данные нельзя изменить через указатель, но сам указатель можно изменить.

```
void readOnly(const int* data) {
// *data = 10; // Ошибка - данные константные
```

```
cout << *data << endl; // Можно читать
}
```

Передача структур
Большие структуры эффективно передавать по константной ссылке (если не нужно изменять) или по ссылке (если нужно изменять). Избегать передачи по значению.

```
struct LargeData {
int array[1000];
double values[500];
};
void processData(const LargeData &data) {
// Эффективное чтение
}
void modifyData(LargeData &data) {
// Изменение оригинала
}
```

Возвращение по значению
Возвращение по значению предпочтительно для небольших объектов и когда нужна копия. Для больших объектов может быть неэффективно.

```
Point createPoint(int x, int y) {
return Point{x, y}; // Подходит для небольших объектов
}
vector<int> createVector() {
return vector<int>{1, 2, 3}; // Может быть неэффективно
}
```

Параметры шаблонных функций
Шаблонные функции могут работать с разными типами, типы параметров выводятся из аргументов. Компилятор генерирует код для каждого используемого типа.

```
template<typename T>
T max(T a, T b) {
return (a > b) ? a : b;
}
// Автоматическое определение типов
int i = max(5, 10);
double d = max(3.14, 2.71);
```

Указатели на функции-члены
Указатели на функции-члены имеют другой синтаксис и требуют объект для вызова. Не взаимозаменяемы с указателями на обычные функции.

```
class MyClass {
public:
void method(int x) { cout << x << endl; }
};
// Указатель на метод
void (MyClass::*methodPtr)(int) = &MyClass::method;
```

```
// Вызов
MyClass obj;
(obj.*methodPtr)(42);
```