**Белорусский государственный технологический университет**

**Факультет информационных технологий**

**Кафедра программной инженерии**

**Реферат**

По дисциплине «Основы алгоритмизации и программирования»

На тему «Отличия функции и макроса, указатели на массивы, умные указатели, использование указателей и ссылок в константных значениях, ссылки»

Выполнил:

Студент 1 курса 10 группы ПИ

Макаревич Кирилл Витальевич

Проверил: Белодед Николай Иванович

2024, Минск

**Оглавление**

[**Введение:** 3](#_Toc184333205)

[**Отличия функции и макроса:** 3](#_Toc184333206)

[***Макрос:*** 3](#_Toc184333207)

[***Функция:*** 4](#_Toc184333208)

[***Сравнение макросов и функций:*** 5](#_Toc184333209)

[**Указатели на массивы:** 6](#_Toc184333210)

[***Основные принципы работы:*** 7](#_Toc184333211)

[***Пример использования указателей на массивы:*** 7](#_Toc184333212)

[***Преимущества использования указателей на массивы:*** 7](#_Toc184333213)

[***Недостатки указателей на массивы:*** 7](#_Toc184333214)

[**Умные указатели:** 7](#_Toc184333215)

[***Типы умных указателей:*** 8](#_Toc184333216)

[***Сравнение обычных и умных указателей:*** 9](#_Toc184333217)

[**Использование указателей и ссылок в константных значениях:** 9](#_Toc184333218)

[***Константы и указатели:*** 9](#_Toc184333219)

[***Ссылки на константы:*** 11](#_Toc184333220)

[***Вывод в виде таблицы:*** 11](#_Toc184333221)

[**Cсылки:** 11](#_Toc184333222)

[**Заключение:** 12](#_Toc184333223)

# **Введение:**

*Цель:*

Целью данного реферата является подробное рассмотрение отличий между функциями и макросами в языке программирования C++, а также изучение их применения, преимуществ и недостатков. Дополнительно рассматриваются указатели и умные указатели, а также их использование в контексте динамической памяти и безопасного управления ею.

*Задачи:*

1. Рассмотреть определения и основные особенности функций и макросов.
2. Описать преимущества и недостатки макросов и функций, а также их основные различия.
3. Привести примеры кода, иллюстрирующие использование макросов и функций.
4. Рассмотреть указатели на массивы, их принципы работы и примеры использования.
5. Изучить умные указатели (unique\_ptr, shared\_ptr, weak\_ptr) и их отличие от обычных указателей.
6. Рассмотреть использование указателей и ссылок на константы в C++.

# **Отличия функции и макроса:**

## ***Макрос:***

Макросы и функции в языках программирования, таких как C++, выполняют схожие задачи — они помогают избежать повторения кода, упрощают его чтение и повторное использование. Однако они имеют кардинальные отличия, которые важно понимать для грамотного применения в различных ситуациях. Макрос — это директива препроцессора, которая заменяет определенные фрагменты кода их значением или выражением до этапа компиляции. Макросы объявляются с использованием #define и позволяют задавать простые константы или сложные выражения.

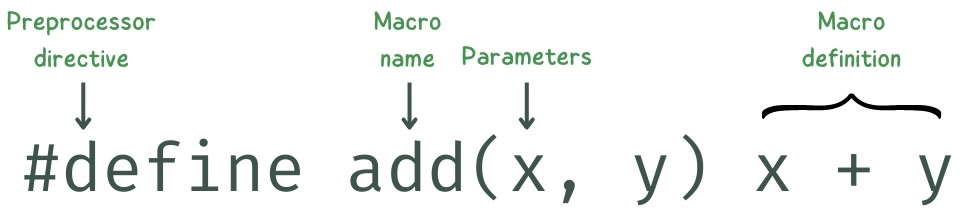
|  |
| --- |
| #define macro\_name(argument\_list) replacement\_code |

Мы можем использовать макросы для определения постоянного значения. Например, мы можем использовать следующий  макрос **PI** в любом месте программы.

|  |
| --- |
| #define PI 3.14159265 |

Мы также можем давать аргументы макросам, делая их похожими на функции. Например, следующий макрос умножает два целых числа.

|  |
| --- |
| #define MULTI(x, y) ((x) \* (y)) |



Препроцессор удаляет все вхождения макроса (и все другие директивы препроцессора) и заменяет его определением макроса, как есть. При обработке макросы заменяются их определением напрямую в исходном коде. Это делает их эффективными, но может вызывать неожиданные результаты при неправильном использовании.

Исходный код:

|  |
| --- |
| #define NUMBER 10  int main() {  int a = NUMBER, b = NUMBER \* NUMBER;  return 0;  } |

Код после обработки:

|  |
| --- |
| int main() {  int a = 10, b = 10 \* 10;  return 0;  } |

Псевдокод:

НАЧАЛО

СОЗДАТЬ целое число a

СОЗДАТЬ целое число b

ПРИСВОИТЬ a значение 10

ПРИСВОИТЬ b значение 10 \* 10

ВЕРНУТЬ 0

КОНЕЦ

В этом коде после предварительной обработки мы видим, что компилятор заменяет все вхождения макроса  NUMBER его определением  10 .

**Преимущества макросов:**

* Макросы сокращают дублирование кода и экономят наше время и усилия.
* Макросы повышают читабельность, определяя простой макрос для сложной задачи.
* Макросы могут включать или отключать определенные блоки кода и поэтому могут быть полезны для отладки.
* Как правило, макросы работают быстрее функций.

**Недостатки макросов:**

* Макросы иногда могут затруднить отладку кода, если ошибка возникает внутри макроса.
* Макросы не выполняют проверку типов и поэтому иногда могут приводить к неожиданным результатам.
* Макросы увеличивают размер кода после предварительной обработки.
* Макросы могут привести к неожиданным результатам, если их использовать неосторожно.

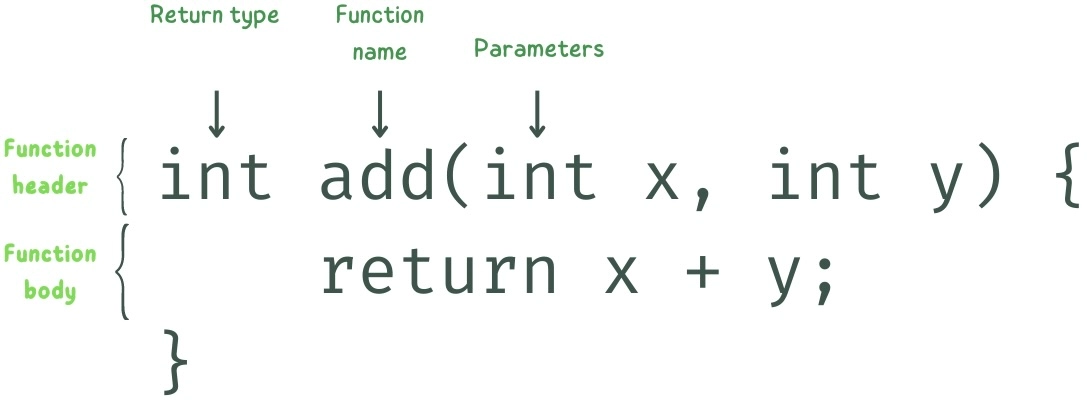
## ***Функция:***

**Функция** — это именованный блок кода, который выполняет определенную задачу. Она может принимать аргументы, возвращать значения и компилируется в машинный код на этапе компиляции. Синтаксис определения функции следующий:

|  |
| --- |
| return\_type function\_name(parameter list) {  // function body  } |

Кроме того, функция не обязательно должна возвращать значение. Например, следующая функция складывает два целых числа.

|  |
| --- |
| int add(int x, int y) {  return x + y;  } |



Компилятор транслирует функции в машинный код и включает их в исполняемую программу, что означает, что функции компилируются и связываются во время компиляции. Давайте рассмотрим пример использования функций.

|  |
| --- |
| #include <iostream>  using namespace std;  int add(int x, int y) {  return x + y;  }  int main() {  cout << "Sum of 1 and 3 = " << add(1, 3) << '\n';  }  Результат программы: |

Псевдокод:

НАЧАЛО

СОЗДАТЬ функцию add с параметрами x и y

ВЕРНУТЬ результат сложения x и y

КОНЕЦ функции add

ВЫВЕСТИ "Sum of 1 and 3 = "

ВЫЗВАТЬ функцию add с параметрами 1 и 3

ВЫВЕСТИ результат выполнения функции

КОНЕЦ

**Преимущества функций:**

* Читаемость и логическая структуризация кода.
* Возможность многократного использования кода.
* Проверка типов данных, что предотвращает ошибки.
* Поддержка перегрузки для работы с разными типами данных.

**Недостатки функций:**

* Отладка рекурсивной функции может оказаться сложной задачей.
* Вызовы функций имеют накладные расходы, что делает их более медленными.

## ***Сравнение макросов и функций:***

|  |  |  |
| --- | --- | --- |
| **Критерий** | **Макрос** | **Функция** |
| **Проверка типов** | Не выполняется. Возможны ошибки при передаче некорректных аргументов. | Выполняется на этапе компиляции. |
| **Отладка** | Сложна из-за прямой подстановки значений. | Проще благодаря структурированному коду. |
| **Производительность** | Работают быстрее за счет исключения вызовов. | Медленнее из-за вызовов функций. |
| **Перегрузка** | Невозможна. | Поддерживается. |
| **Время обработки** | На этапе предварительной обработки. | На этапе компиляции. |
| **Возвращаемые значения** | Не поддерживаются. | Поддерживаются. |
| **Длина кода** | Увеличивается после обработки. | Остается неизменной. |
| **Длина** | Обычно одна строка. Поддерживаются многострочные макросы с использованием символа \. | Может охватывать любое количество строк (без ограничений). |
| **Побочные эффекты** | Возможны неожиданные побочные эффекты. | Как правило, отсутствуют. |
| **Использование** | Полезны для коротких повторяющихся участков кода. | Полезны для крупных участков кода. |

**Разница на примере:**

|  |
| --- |
| #include <iostream>  using namespace std;  int MUL\_FUNC(int x, int y) {  return x \* y;  }  #define MUL\_MACRO(x, y) x \* y  int main() {  cout << "MUL\_FUNC(2 + 3, 4 + 5): " << MUL\_FUNC(2 + 3, 4 + 5) << '\n';  cout << "MUL\_MACRO(2 + 3, 4 + 5): " << MUL\_MACRO(2 + 3, 4 + 5) << '\n';  return 0;  }  Результат программы: |

# **Указатели на массивы:**

Указатель на массив — это переменная, которая хранит адрес первого элемента массива. Благодаря этому, указатели могут использоваться для работы с массивами через адресации памяти. Используя указатели, мы можем получать доступ к элементам массива, манипулировать ими и передавать массивы в функции без необходимости копирования данных.

## ***Основные принципы работы:***

1. Адресация массива

Когда массив создаётся в памяти, его имя (arr) указывает на адрес первого элемента. Например, для массива int arr[5] имя arr эквивалентно указателю int\* на arr[0].

2. Арифметика указателей

Указатели поддерживают арифметические операции. Например, выражение ptr + i позволяет обратиться к i-му элементу массива, а \*(ptr + i) возвращает значение этого элемента.

3. Передача массивов в функции

Указатели позволяют передавать массивы в функции как ссылку, избегая дорогостоящего копирования. Это особенно важно для работы с большими массивами.

## ***Пример использования указателей на массивы:***

|  |
| --- |
| #include <iostream>  using namespace std;  int main() {  int arr[5] = { 1, 2, 3, 4, 5 };  int\* ptr = arr;  for (int i = 0; i < 5; i++) {  cout << "arr[" << i << "] = " << \*(ptr + i) << endl;  }  return 0;  }  Результат программы: |

Псевдокод:

НАЧАЛО

СОЗДАТЬ массив arr размером 5 с элементами {1, 2, 3, 4, 5}

СОЗДАТЬ указатель ptr, указывающий на первый элемент массива arr

ДЛЯ каждого индекса i от 0 до 4

ВЫВЕСТИ значение элемента массива arr[i], используя указатель ptr

КОНЕЦ

КОНЕЦ

## ***Преимущества использования указателей на массивы:***

1. Эффективность памяти: Указатели передают адреса массива, избегая копирования данных.
2. Гибкость: Указатели позволяют работать с элементами массивов через арифметику адресов.
3. Простота передачи: Массивы можно передавать в функции без дополнительных накладных расходов.
4. Доступ к многомерным массивам: Указатели упрощают работу с многомерными массивами.

## ***Недостатки указателей на массивы:***

1. Сложность отладки: Ошибки, связанные с указателями, могут быть сложны для обнаружения и исправления.
2. Отсутствие проверки границ: При использовании указателей программист несёт ответственность за проверку диапазонов массива.
3. Потенциальные ошибки: Неправильное использование указателей может привести к повреждению данных или утечке памяти.

# **Умные указатели:**

Умные указатели (smart pointers) являются одной из важных возможностей современного C++ для управления динамической памятью. Они были введены в стандарт C++. Умные указатели — это классы-обертки над обычными указателями, которые обеспечивают безопасное управление динамической памятью. Они автоматически освобождают занимаемую память, когда указатель становится ненужным, используя механизмы подсчета ссылок и автоматического удаления. Это помогает избежать типичных ошибок работы с памятью, таких как "утечки" или двойное освобождение.

## ***Типы умных указателей:***

1. unique\_ptr
2. shared\_ptr
3. weak\_ptr

Каждый из них имеет свои особенности и используется в зависимости от конкретной ситуации.

1. unique\_ptr

Умный указатель unique\_ptr обеспечивает эксклюзивное владение ресурсом. Это означает, что указатель владеет объектом, и его нельзя копировать. Когда объект становится ненужным, память автоматически освобождается.

|  |
| --- |
| #include <iostream>  #include <memory>  using namespace std;  int main() {  auto ptr = make\_unique<int>(10); // Создаём уникальный указатель на число 10  cout << "Value: " << \*ptr << endl;  return 0;  } |

Псевдокод:

НАЧАЛО

СОЗДАТЬ уникальный указатель ptr на целое число со значением 10

ВЫВЕСТИ "Value: " + значение, на которое указывает ptr

КОНЕЦ

Особенности:

* Нельзя копировать, но можно передавать владение через move.
* Подходит для ситуаций, где объект имеет одного владельца.

2. shared\_ptr

shared\_ptr обеспечивает разделяемое владение ресурсом. Несколько указателей могут владеть одним объектом. Освобождение памяти происходит автоматически, когда последний владелец (shared\_ptr) удаляется.

|  |
| --- |
| #include <iostream>  #include <memory>  using namespace std;  int main() {  shared\_ptr<int> ptr1 = make\_shared<int>(20);  shared\_ptr<int> ptr2 = ptr1;  cout << "Value: " << \*ptr1 << endl;  cout << "Reference count: " << ptr1.use\_count() << endl;  return 0;  } |

Псевдокод:

НАЧАЛО

СОЗДАТЬ shared\_ptr ptr1, указывающий на число 20

СОЗДАТЬ shared\_ptr ptr2, который делит владение с ptr1

ВЫВЕСТИ значение, на которое указывает ptr1

ВЫВЕСТИ количество ссылок (использований) ptr1

КОНЕЦ

Особенности:

* Подходит для ситуаций, где ресурс используется несколькими объектами.
* Использует механизм подсчета ссылок (reference counting).

3. weak\_ptr

weak\_ptr — это умный указатель, который не участвует в управлении временем жизни объекта. Он используется совместно с shared\_ptr для предотвращения циклических ссылок.

|  |
| --- |
| #include <iostream>  #include <memory>  int main() {  std::shared\_ptr<int> sharedPtr = std::make\_shared<int>(30);  std::weak\_ptr<int> weakPtr = sharedPtr;  if (auto temp = weakPtr.lock()) { // Проверяем, доступен ли объект  std::cout << "Value: " << \*temp << std::endl;  }  else {  std::cout << "Resource is no longer available." << std::endl;  }  return 0;  } |

Псевдокод:

НАЧАЛО

СОЗДАТЬ shared\_ptr sharedPtr, указывающий на число 30

СОЗДАТЬ weak\_ptr weakPtr, который ссылается на sharedPtr

ЕСЛИ можно заблокировать weakPtr (проверить доступность объекта)

ВЫВЕСТИ значение, на которое указывает объект

ИНАЧЕ

ВЫВЕСТИ "Ресурс больше не доступен"

КОНЕЦ

Особенности:

* Не увеличивает счетчик ссылок.
* Используется для создания "слабых" ссылок на объект.

## ***Сравнение обычных и умных указателей:***

|  |  |  |
| --- | --- | --- |
| **Критерий** | **Обычные указатели** | **Умные указатели** |
| **Освобождение памяти** | Ручное управление (delete). | Автоматическое управление. |
| **Устойчивость к ошибкам** | Подвержены утечкам памяти. | Устраняют утечки памяти. |
| **Сложность использования** | Просты, но требуют тщательного управления. | Более сложные, но безопасные. |
| **Совместное использование** | Требует ручного управления ссылками. | Поддерживается с помощью std::shared\_ptr. |

# **Использование указателей и ссылок в константных значениях:**

## ***Константы и указатели:***

1. Указатели на константы

Указатели могут указывать как на переменные, так и на константы. Чтобы определить указатель на константу, он тоже должен объявляться с ключевым словом const:

|  |
| --- |
| #include <iostream>  int main()  {  const int a{ 10 };  const int\* pa{ &a };  std::cout << "address=" << pa << "\tvalue=" << \*pa << std::endl;  } |

Псевдокод:

НАЧАЛО

СОЗДАТЬ константную переменную a, присвоить ей значение 10

СОЗДАТЬ указатель pa, который ссылается на a (pa указывает на константное значение)

ВЫВЕСТИ адрес, на который указывает pa, и значение по этому адресу

КОНЕЦ

Здесь указатель pa указывает на константу a. Поэтому даже если мы захотим изменить значение по адресу, который хранится в указателе, мы не сможем это сделать.

2. Константный указатель

От указателей на константы надо отличать константные указатели. Они не могут изменять адрес, который в них хранится, но могут изменять значение по этому адресу.

|  |
| --- |
| #include <iostream>  int main()  {  int a{ 10 };  int\* const pa{ &a };  std::cout << "value=" << \*pa << std::endl; // value = 10  \*pa = 22; // меняем значение  std::cout << "value=" << \*pa << std::endl; // value = 22  int b{ 45 };  // pa = &b; так нельзя сделать  } |

Псевдокод:

НАЧАЛО

СОЗДАТЬ переменную a и присвоить ей значение 10

СОЗДАТЬ константный указатель pa, который указывает на a

ВЫВЕСТИ значение, на которое указывает pa (значение a)

ИЗМЕНИТЬ значение по адресу, на который указывает pa (поменять значение a на 22)

ВЫВЕСТИ новое значение, на которое указывает pa (новое значение a)

СОЗДАТЬ переменную b и присвоить ей значение 45

ПЫТАТЬСЯ присвоить указателю pa адрес переменной b (это невозможно, так как pa — константный указатель)

КОНЕЦ

3. Константный указатель на константу

И объединение обоих предыдущих случаев - константный указатель на константу, который не позволяет менять ни хранимый в нем адрес, ни значение по этому адресу:

|  |
| --- |
| int main()  {  int a{ 10 };  const int\* const pa{ &a };  //\*pa = 22; так сделать нельзя  int b{ 45 };  // pa = &b; так сделать нельзя  } |

Псевдокод:

НАЧАЛО

СОЗДАТЬ переменную a и присвоить ей значение 10

СОЗДАТЬ константный указатель на константу pa, который указывает на a

СОЗДАТЬ переменную b и присвоить ей значение 45

КОНЕЦ

## ***Ссылки на константы:***

Ссылка на константу (const) позволяет привязаться к значению таким образом, чтобы запретить его изменение через эту ссылку. Это полезно, когда требуется передать данные по ссылке для избежания копирования, но нужно защитить данные от изменения.

|  |
| --- |
| #include <iostream>  using namespace std;  int main() {  int value{ 10 };  const int& ref{ value }; // Ссылка на константу  cout << "Значение: " << ref << '\n';  // ref = 20; // Ошибка! Нельзя изменять значение через ссылку  value = 20; // Прямое изменение допустимо  return 0;  } |

Псевдокод:

НАЧАЛО

СОЗДАТЬ переменную value и присвоить ей значение 10

СОЗДАТЬ ссылку на константу ref, которая ссылается на value

ВЫВЕСТИ значение, на которое указывает ссылка ref (значение будет 10)

value = 20; // Это допустимо

КОНЕЦ

## ***Вывод в виде таблицы:***

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| const int\* ptr | Указатель на константные данные (данные неизменяемы через указатель). |
| int\* const ptr | Константный указатель на изменяемые данные (указатель неизменяем, данные изменяемы). |
| const int\* const ptr | Константный указатель на константные данные (и указатель, и данные неизменяемы). |
| const int& ref | Ссылка на константные данные (данные неизменяемы через ссылку). |

# **Cсылки:**

Ссылки в C++ представляют собой альтернативный способ работы с данными, позволяя ссылаться на существующую переменную без создания новой копии. Ссылка является псевдонимом переменной: любое действие с ней отразится на оригинальном объекте.

**Основные особенности ссылок:**

1. Объявление:  
Ссылка объявляется с использованием символа &.

2. Обязательная инициализация:  
Ссылки должны быть инициализированы при создании.

3. Неизменяемость объекта:  
После привязки ссылка всегда будет указывать на один и тот же объект. Ее нельзя переназначить.

4. Ссылки работают как "прозрачный доступ":  
Операции через ссылку идентичны операциям с самой переменной.

**Виды ссылок**

1. **Ссылки на переменные:** Используются для упрощения синтаксиса и прямого доступа к объектам.

|  |
| --- |
| #include <iostream>  using namespace std;  int main() {  int number{ 5 };  int& refNumber{ number };  return 0;  } |

Псевдокод:

НАЧАЛО

СОЗДАЕМ переменную number и присваиваем ей значение 5

СОЗДАЕМ ссылку refNumber, которая ссылается на переменную number

КОНЕЦ

2. **Ссылки на константы**: Рассказал про них выше в реферате.

3. **R-value ссылки (C++11):** Позволяют работать с временными объектами. Используются для перемещения данных (move semantics) и оптимизации.

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  // Функция для временных объектов  void process(string&& temp) {  cout << "Обработан временный объект: " << temp << '\n';  }  // Функция для постоянных объектов  void process(const string& ref) {  cout << "Обработана постоянная строка: " << ref << '\n';  }  int main() {  setlocale(LC\_ALL, "ru");  string str = "Привет";  process(str); // Передается постоянная строка  process("Временный объект"); // Передается временный объект  return 0;  } |

Псевдокод:

НАЧАЛО

СОЗДАТЬ строку str с значением "Привет"

ВЫЗВАТЬ функцию process с передачей постоянного объекта str:

ВЫВЕСТИ "Обработана постоянная строка: Привет"

ВЫЗВАТЬ функцию process с передачей временного объекта "Временный объект":

ВЫВЕСТИ "Обработан временный объект: Временный объект"

КОНЕЦ

Ссылки — это удобный и безопасный способ работы с данными, позволяющий писать более читаемый и эффективный код.

# **Заключение:**

Изучение различий между функциями и макросами, а также их применения, показывает, что выбор между этими инструментами зависит от конкретной ситуации и требований к коду. Макросы эффективны в случаях, когда нужно повысить производительность и уменьшить количество кода, однако их использование требует осторожности из-за возможных ошибок, связанных с отсутствием проверки типов и неожиданными побочными эффектами. Функции, в свою очередь, более безопасны и удобны для работы с типами данных, обеспечивая проверку типов и поддержку перегрузки.

Указатели, включая умные указатели, играют важную роль в работе с динамической памятью, предлагая гибкость и безопасность, особенно при использовании в современных C++ приложениях. Умные указатели значительно снижают вероятность ошибок, таких как утечки памяти или двойное освобождение памяти. Константные указатели и ссылки обеспечивают дополнительные гарантии безопасности, предотвращая модификацию данных, на которые они ссылаются, что особенно полезно при разработке стабильных и надежных программ.