Модуль 6

- Константность типов
- Массивы
- C-style строки
- Указатели
- Оператор new, delete
- Ссылки
- Введение в std::array, std::vector
- Умные указатели

Константность типов

Константы - переменные, значения которых изменить нельзя.

Если не предполагается, что переменная будет изменяться **всегда!** Надо использовать **const**: в параметрах функции, локальные переменные и тд.

```
int foo(const int& f, const int& g)
{
   const int y{2};
   const int h{poo(5,6)};
}
```

const int = int const, но первый вариант лучше.

constexpr

Выполняется на этапе компиляции

```
constexpr int square(int x) {
    return x * x;
}
int main() {
    constexpr int result = square(4); // Значение известно на этапе компиляции
    return 0;
}
```

Указатели

Оператор адреса & позволяет узнать, какой адрес памяти присвоен определенной переменной.

```
int a = 7;
&a - адрес где размещена переменная
```

Оператор разыменования * позволяет получить значение по указанному адресу:

```
*&a - выводим значение ячейки памяти переменной a
```

Указатели

Указатель — это переменная, значением которой является адрес ячейки памяти

```
int *iPtr; // указатель на значение типа int
double *dPtr; // указатель на значение типа double

int value = 5;
int *ptr = &value; // инициализируем ptr адресом значения переменной
```

Указатели

Размер указателей

Размер указателя зависит от архитектуры, на которой скомпилирован исполняемый файл: 32-битный исполняемый файл использует 32-битные адреса памяти, 64-битный файл - 64-битные адреса.

Указатели

Польза указателей

Случай №1: **Массивы реализованы с помощью указателей**. Указатели могут использоваться для итерации по массиву.

Случай №2: **Они являются единственным способом динамического выделения памяти в C++**. Это, безусловно, самый распространенный вариант использования указателей.

Случай №3: **Они могут использоваться для передачи большого количества данных в функцию без копирования этих данных**.

Случай №4: **Они могут использоваться для передачи одной функции в качестве параметра другой функции**.

Случай №5: **Они используются для достижения полиморфизма при работе с наследованием**.

Указатели

Что не так с кодом?

```
int value = 45;
int *ptr = &value;
*ptr = &value;
```

Указатели

Нулевые указатели

Нулевое значение — это специальное значение, которое означает, что указатель ни на что не указывает. Указатель, содержащий значение null, называется **нулевым указателем**.

`int *ptr = nullptr;` – Инициализируйте указатели нулевым значением, если не собираетесь присваивать им другие значения. Перед использованием, проверяется является ли указатель нулевым.

Указатели

```
#include <iostream>

void doAnything(int *ptr)
{
    if (ptr)
        std::cout << "You passed in " << *ptr << '\n';
    else
        std::cout << "You passed in a null pointer\n";
}
int main()
{
    doAnything(nullptr); // теперь аргумент является точно нулевым указателем, а не целочисленным значением
    return 0;
}
```

Константные указатели

Константный указатель — это указатель, значение которого не может быть изменено после инициализации.

```
int value = 7;
int *const ptr = &value; - ptr всегда будет указывать на адрес value
```

НО

```
*ptr = 8; // ок, так как ptr указывает на тип данных (неконстантный int)
```

Константные указатели на константные значения

```
int value = 7;
const int *const ptr = &value;
```

Динамическое выделение памяти

Типы выделения памяти

Статическое выделение памяти выполняется для **статических** и **глобальных** переменных. Память выделяется один раз (при запуске программы) и сохраняется на протяжении работы всей программы.

Автоматическое выделение памяти выполняется для **параметров функции** и **локальных переменных**. Память выделяется при входе в блок, в котором находятся эти переменные, и удаляется при выходе из него.

Динамическое выделение памяти.

Динамическое выделение памяти

Размер переменной/**массива** должен быть известен во время компиляции.

И что делать?

Для динамического выделения памяти одной переменной используется оператор **new**:

```
int *ptr = new int;
*ptr1 = 8;
```

```
int *ptr1 = new int (7); // используем прямую инициализацию
int *ptr2 = new int { 8 }; // используем uniform-инициализацию
```

Для освобождения памяти используется оператора **delete**:

```
delete ptr; // возвращаем память, на которую указывал ptr, обратно в
операционную систему
ptr = nullptr;
```

Динамическое выделение памяти

Указатель, указывающий на освобожденную память, называется **висячим указателем**.

```
#include <iostream>
int main()
{
    int *ptr = new int;
    *ptr = 8;
    delete ptr; // ptr - висячий указатель
    std::cout << *ptr;
    delete ptr;
    return 0;
}
```

Динамическое выделение памяти

Оператор new

Бывает, что память может быть не выделена и в таком случае будет сгенерировано исключение.

Необходимо отлавливать исключения

!Но это крайне редкая ситуация и разработчики зачастую игнорируют данную рекомендацию.

Динамическое выделение памяти

Перед выделением памяти необходимо проверять на null.

```
if (!ptr)
    ptr = new int;
```

При удалении необходимости в проверке нету

```
delete ptr;
```

Динамическое выделение памяти

Утечка памяти

Динамически выделенная память не имеет области видимости, т.е. она остается выделенной до тех пор, пока не будет явно освобождена или пока ваша программа не завершит свое выполнение.

```
void doSomething()
{
    int *ptr = new int;
}
```

```
int *ptr = new int;
ptr = new int;
```

Массивы

Массив — совокупный тип данных, который позволяет получить доступ ко всем переменным одного и того же типа данных через использование одного идентификатора.

```
int data1;
int data2;
int data3;
// ...
int data30;
```

Можно заменить на

```
int data[30];
```

`[]`, чтобы сообщить компилятору, что это переменная массива

Фиксированный массив - Это массив, размер которого известен во время компиляции.

Массивы

Для работы с массивом используются индексы.
Индексы начинаются с нуля и всегда целые значения!

data[0],
data[1],
data[5],
data[x].

Структуры могут быть любого типа, даже пользовательского.

Массивы

Объявление массивов фиксированного размера

При объявлении массива фиксированного размера, его длина (между квадратными скобками) должна быть **константой типа compile-time.**

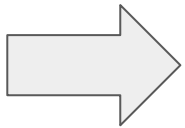
Массивы

```
int array[5] = { 4, 5, 8, 9, 12 };

int array[5] = { 4, 5};

int array[5] = { };

int array[5] { 4, 5, 8, 9, 12 };

int array[] = { 0, 1, 2, 3, 4 }; // список инициализаторов автоматически определит длину массива
```

Массивы

Массивы и перечисления.

```
const int numberOfStudents(5);
int testScores[numberOfStudents];
testScores[3] = 65;
```



```
enum StudentNames
{
    SMITH, // 0
    ANDREW, // 1
    IVAN, // 2
    JOHN, // 3
    ANTON, // 4
    MAX_STUDENTS // 5
};
int main()
{
    int testScores[MAX_STUDENTS];
    testScores[JOHN] = 65;
    return 0;
}
```

Массивы

Передача массивов в функции

Массив передается не весь а только указатель на первый элемент.
Необходимо дополнительно передавать и количество элементов в массиве.

void passArray(int array[], int size);

void passArray(const int array[5]) - const спасет от модификации данных.

Массивы

Массивы и циклы

```
int students[] = { 73, 85, 84, 44, 78};
const int numStudents = sizeof(students) / sizeof(students[0]);
int totalScore = 0;
// Используем цикл для вычисления totalScore
for (int person = 0; person < numStudents; ++person)
    totalScore += students[person];
```

Задача

Напишите программу, которая находит количество повторяющихся элементов в заданном массиве целых чисел.

Массивы

Многомерные массивы

```
int array[2][4]; // 2-элементный массив из 4-элементных массивов

Инициализация двумерного массива:
int array[3][5] = { 0 };

int array[3][5] =
{
  { 2, 4  }, // строка №0 = 2, 4, 0, 0, 0
  { 1, 3, 7 }, // строка №1 = 1, 3, 7, 0, 0
  { 8, 9, 11, 12 } // строка №2 = 8, 9, 11, 12, 0
};
```

Массивы

```
for (int row = 0; row < numRows; ++row) // доступ по строкам
    for (int col = 0; col < numCols; ++col) // доступ к каждому элементу в строке
        std::cout << array[row][col];
```

Массивы

Многомерные массивы

```
int array[4][3][2];
```

Массивы

```
int array[4] = { 5, 8, 6, 4 };
```

Массив последовательно располагается в памяти.

Array - указатель на первый элемент массива.

Array[1] = *(Array + 1)

Динамические массивы

Для выделения динамического массива и работы с ним используются отдельные формы операторов new и delete: `new[]` и `delete[].`

Массивы

Инициализация динамических массивов

```
int *array = new int[length]();
```
- инициализация нулями

```
int *array = new int[5] { 9, 7, 5, 3, 1 };
```
- *инициализируем динамический массив*

Важно чтобы было указан размер массива!

```
int *dynamicArray1 = new int[] {1, 2, 3};
```
// не ок: неявное указание длины динамического массива

Массивы

Изменение длины массивов

C++ не предоставляет встроенный способ изменения длины массива, который уже был выделен. Можно скопировать в новое место.

Ссылки

Ссылка — это тип переменной в языке C++, который работает как псевдоним другого объекта или значения.

```
int value = 7; // обычная переменная
int &ref = value; // ссылка на переменную value
```

Ссылки

Ссылки нулевыми быть не могут.

```
int a = 7;
int &ref1 = a; // ок: a - это неконстантное l-value
const int b = 8;
int &ref2 = b; // не ок: b - это константное l-value
int &ref3 = 4; // не ок: 4 - это r-value


int value1 = 7;
int value2 = 8;
int &ref = value1; // ок: ref - теперь псевдоним для value1
ref = value2; // присваиваем 8 (значение переменной value2) переменной
value1. Здесь НЕ изменяется объект, на который ссылается ссылка!
```

Ссылки

Ссылки как более легкий способ доступа к данным

```
struct Something
{
    int value1;
    float value2;
};

struct Other
{
    Something something;
    int otherValue;
};
Other other;
```

```
int &ref = other.something.value1;


other.something.value1 = 7;
ref = 7;
```

Ссылки vs. Указатели

1. Синтаксис объявления: Ссылка объявляется с использованием символа амперсанда (&), в то время как указатель объявляется с использованием звездочки (*).

```
int x = 10;

int& ref = x; // ссылка

int* ptr = &x; // указатель
```

2. Присваивание и инициализация: Ссылка должна быть инициализирована при объявлении и не может быть переинициализирована после этого, тогда как указатель может быть неинициализирован или указывать на другой объект в любое время.

3. Использование оператора разыменования: Для доступа к значению, на которое указывает указатель, используется оператор разыменования (*), а для ссылки это не требуется.

```
int x = 10;
int& ref = x; // ссылка
int* ptr = &x; // указатель
int value1 = ref; // Доступ к значению по ссылке
int value2 = *ptr; // Доступ к значению по указателю
```

Ссылки vs. Указатели

4. Нулевое значение: Ссылка всегда должна ссылаться на объект, тогда как указатель может иметь значение nullptr или быть нулевым.

```
int* ptr = nullptr; // указатель, который не указывает ни на что
```

5. Уровень защиты: Все ссылки являются константами по умолчанию и не могут ссылаться на другой объект после инициализации, в то время как указатель может быть изменен для указания на другой объект

6. Синтаксис использования: Для использования значения, на которое ссылается указатель, требуется оператор разыменования, а для ссылки это не требуется, что делает код более лаконичным и понятным.

7. Передача аргументов в функцию: При передаче ссылок в функцию параметр изменяется непосредственно в вызывающей функции, в то время как указатель должен быть разыменован в функции для изменения значения по адресу.

8. Существует арифметика указателей.

Строки C-style

Строка C-style — это простой **массив** символов, который использует нуль-терминатор.

Пример:

```
char mystring[] = "string";
```

Строки C-style

Язык C++ предоставляет множество функций для управления строками C-style, которые подключаются с помощью **заголовочного файла** cstring.

strcpy_s() позволяет копировать содержимое одной строки в другую

strlen() возвращает длину строки C-style

функция strcat() — добавляет одну строку к другой (опасно);

функция strncat() — добавляет одну строку к другой (с проверкой размера места назначения);

функция strcmp() — сравнивает две строки (возвращает 0, если они равны);

функция strncmp() — сравнивает две строки до определенного количества символов (возвращает 0, если до указанного символа не было различий).

Строки C-style

Стоит ли использовать строки C-style?

Знать о строках C-style стоит, так как они используются не так уж и редко, но использовать их без веской на то причины не рекомендуется. Вместо строк C-style используйте std::string (подключая заголовочный файл string), так как он проще, **безопаснее** и гибче.

std::array

std::array — это фиксированный массив, который не распадается в указатель при передаче в функцию. Это уже объект. И может использовать все алгоритмы стандартной библиотеки.

```
std::array<int, 4> myarray; - Размер должен быть известен еще на этапе компиляции.
```

```
std::array<int, 4> myarray = { 8, 6, 4, 1 }; // список инициализаторов
std::array<int, 4> myarray2 { 8, 6, 4, 1 }; // uniform-инициализация
```

Доступ к значениям массива через оператор индекса:

```
myarray[2] = 7;
```

```
myarray.at(1) = 7; - доступ с проверкой.
```

std::array

Задача: Напишите программу, которая находит среднее арифметическое элементов массива и выводит наименьший элемент, который ближе всего к этому среднему значению.

std::vector

std::vector (или просто ***«вектор»***) — это тот же динамический массив, но который может сам управлять выделенной себе памятью.

Нет учнчки памяти, потому что вектор сам следит за пимятью

```
std::vector<int> array;
std::vector<int> array2 = { 10, 8, 6, 4, 2, 1 };
std::vector<int> array3 { 10, 8, 6, 4, 2, 1 };
array = { 0, 2, 4, 5, 7 }; // ок, длина array теперь 5
array = { 11, 9, 5 }; // ок, длина array теперь 3

Доступ по индексу
array[7] = 3; // без проверки диапазона
array.at(8) = 4; // с проверкой диапазона
```

std::vector

Задача: найти наибольшую сумму рядом стоящих элементов.

Умные указатели

Проблема

```
#include <iostream>
void myFunction()
{
    Item *ptr = new Item;
    int a;
    std::cout << "Enter an integer: ";
    std::cin >> a;
    if (a == 0)
        return; // функция выполняет досрочный возврат, вследствие чего ptr не
будет удален!
    // Делаем что-либо с ptr здесь
    delete ptr;
}
```

Умные указатели

Особенность классов — это **деструкторы**, которые автоматически выполняются при выходе объекта класса из области видимости. При выделении памяти в конструкторе класса, память будет освобождена в деструкторе при уничтожении объекта класса. Это лежит в основе парадигмы программирования **RAII**.

Умные указатели

auto_ptr - первая попытка реализовать умный указатель

Недостатки:

1) Семантика перемещения реализована через конструктор копирования и оператор присваивания. При передачи auto_ptr возникнут проблемы с копированием указателя и двойного удаления
2) используется оператор delete, который не работает с массивами.
3) Плохо работает с некоторыми стандартными контейнерами из-за того что контейнеры предполагают что будет копирование а не перемещение.

std::auto_ptr устарел и не должен использоваться!

Умные указатели

std::unique_ptr

Умный указатель std::unique_ptr является заменой std::auto_ptr в C++11.

При использовании std::unique_ptr маскируется под обычный указатель и имеет такое же поведение с *, ->, сравнении с null.

Умные указатели

std::make_unique() - шаблон функции, который создает объект типа шаблона и инициализирует его аргументами, переданными в функцию.

Рекомендуется использовать std::make_unique() вместо использования умного указателя std::unique_ptr из-за проблемы безопасности использования исключений.

```
some_function(std::unique_ptr<T>(new T), function_that_can_throw_exception());
```

Если после создания T вызовется исключение, то объект удален не будет.
std::make_unique() - решает эту проблему.

Как делать не надо

```
Item *item = new Item;
std::unique_ptr<Item> item1(item);
std::unique_ptr<Item> item2(item);
```

```
Item *item = new Item;
std::unique_ptr<Item> item1(item);
delete item;
```

Умные указатели

std::shared_ptr предназначен для случаев, когда несколько умных указателей совместно владеют одним динамически выделенным ресурсом.

Выполняйте копирование существующего std::shared_ptr, если вам нужно более одного std::shared_ptr, указывающего на один и тот же динамически выделенный ресурс.

Умные указатели

std::make_shared()

Причина использования std::make_shared() такая же, как и при использовании функции std::make_unique(): проще, безопаснее и производительнее за счет того, что std::shared_ptr отслеживает, сколько умных указателей владеют ресурсом.

Умные указатели

Реализация std::shared_ptr

Используется внутренний счетчик сколько ссылается объектов на указатель. Удаляется только когда счетчик равен нулю.

Умные указатели

std::shared_ptr и массивы

std::shared_ptr поддерживает массивы, но std::make_shared() стал поддерживать массивы только с 17 стандарта.

Умные указатели

std::weak_ptr

Циклическая зависимость (или ***«циклические ссылки»***) — это серия «ссылок», где текущий объект ссылается на следующий, а последний объект ссылается на первый.

std::weak_ptr является наблюдателем — он может наблюдать и получать доступ к тому же объекту, на который указывает std::shared_ptr (или другой std::weak_ptr), но **не считаться владельцем этого объекта**.

Умные указатели

Недостатком умного указателя std::weak_ptr является то, что его нельзя использовать напрямую (нет оператора `->`). Чтобы использовать std::weak_ptr, вы сначала должны конвертировать его в std::shared_ptr (с помощью **метода lock()**)