

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Урок № 15

Многомерные динамические массивы

Содержание

1. Многомерные динамические массивы	3
2. Примеры на многомерные динамические массивы	10
Пример 1. Организация двумерного «треугольного» динамического массива	10
Пример 2. Организация трехмерного динамического массива	11
3. Перечисляемые типы	13
Указатели на функции	19
Иллюстрируем на практике	23
Массивы указателей на функции	31
Ключевые слова auto и decltype.	
Автоматическое выведение типа	34
Хвостовой возвращаемый тип	44

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе [Adobe Acrobat Reader](#).

1. Многомерные динамические массивы

И, снова в бой! Мы с вами уже сталкивались с динамическими массивами, однако нам бы хотелось коснуться этой темы еще раз и рассказать вам кое-что о создании многомерных динамических массивов.

В предыдущих уроках мы уже рассмотрели назначение и особенности работы с операторами `new` и `delete` для динамического выделения и освобождения памяти, в том числе и для создания (удаления) динамических массивов. Так как операнды `new` могут быть не только константами, то мы можем создавать динамические массивы, размер которых определяется в ходе выполнения программы.

Предположим, что нужно хранить данные о результатах тестирования студентов по некоторым курсам. Во-первых, количество студентов заранее неизвестно и эту информацию должен предоставить пользователь уже после запуска программы. Во-вторых, каждый из студентов проходил тестирование по разному количеству предметов (т.е. количество оценок у студентов разное). Использование статических двумерных массивов в этом случае нецелесообразно.

В случае большого, заранее зарезервированного количества студентов при фактически небольшой их численности мы выделим память и будем использовать только ее часть, а не всю выделенную память. Или, является еще более критическим, объем зарезервированной памяти может оказаться недостаточным для реальных данных. Например,

выделяя память под десять предметов, т.к. именно такое было максимальное число курсов у студентов ранее, мы встречаем студента, который прослушал двенадцать предметов и нам некуда записать еще две оценки. Использование многомерных динамических массивов позволяет не только создавать хранилища произвольной размерности, но и устанавливать ее в процессе работы программы (после получения необходимых данных от пользователя).

Создание многомерных динамических массивов происходит на основе указателей на указатели.

Многомерный массив в C++ по своей сути одномерен, так как фактически представляет собой одномерный массив указателей (A), т.е. каждый элемент ($A[i]$) — это указатель на массив с конкретными значениями. На рисунке 1 эти массивы со значениями имеют одинаковую длину, но это не обязательно. Таким образом, динамический многомерный массив реализуется как массив указателей на массивы, значения в которых, в свою очередь, тоже могут быть указателями на массивы, образуя, например, трехмерный динамический массив.

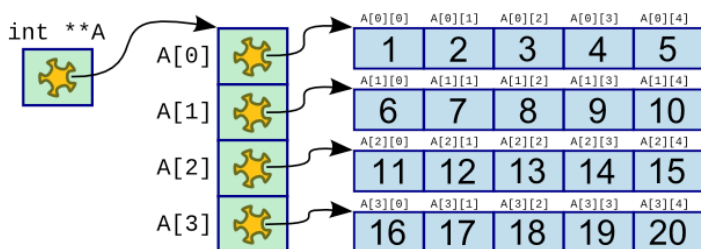


Рисунок 1

Рассмотрим пример создания двумерного динамического массива (матрицы 5×2 с целыми значениями).

Сначала объявляется указатель на указатель (`int** pArr`), который, в свою очередь, ссылается на массив указателей (`new int* [m1]`) заданного размера (количество строк, в нашем примере — две).

```
int** A = new int* [m1];
```

`m1` — количество строк; предполагается, что задано выше.

Результат данного шага обведен на рисунке 2 красным цветом.

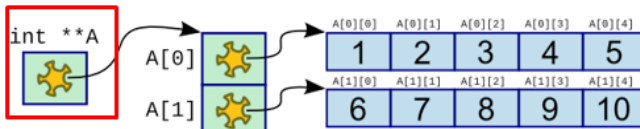


Рисунок 2

Строка матрицы (элемент основного массива) представляет собой массив указателей, длина которого соответствует нужному количеству столбцов (5 в нашем примере).

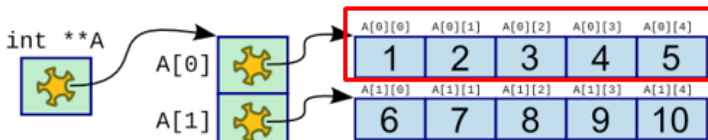


Рисунок 3

Каждый такой массив указателей (на фактические значения) необходимо инициализировать (выделить необходимый размер памяти). Данная операция реализуется с помощью цикла, количество итераций которого совпадает с количеством строк матрицы (размером основного

массива указателей). Результат работы первого шага такого цикла обведен красным цветом на рисунке 3.

```
for (i = 0; i < m1; i++)
{
    A[i] = new int[m2];
}
```

m2 — количество столбцов (пять); предполагается, что задано выше.

Как объявление, так и удаление двумерного динамического массива выполняется с помощью цикла, так как нужно вначале отдельно удалить каждый динамический массив, представляющий строку, а потом удалить динамический массив указателей на строки.

Т.е. мы удаляем массив в обратном порядке его создания.

```
// Последовательное удаление двумерного массива...
for (i = 0; i < m1; i++)
{
    delete[] A[i];
}
delete[] A;
```

Следующий пример иллюстрирует работу с динамическими массивами.

```
#include <iostream>

using namespace std;
int main()
{
```

```

int i, j;

// Переменные для описания характеристик массивов.
// m1 – количество строк, m2 – количество столбцов
int m1 = 5, m2 = 5;

/* Организация двумерного динамического массива
   производится в два этапа.
   Сначала создаётся одномерный массив указателей,
   а затем каждому элементу этого массива
   присваивается адрес одномерного массива.
   Для характеристик размеров массивов не
   требуется константных выражений.
*/

int** pArr = new int* [m1];

for (i = 0; i < m1; i++)
{
    pArr[i] = new int[m2];
}
// Доступ к элементам массива выполняется как обычно
pArr[3][3] = 100;
cout << pArr[3][3] << "\n";

// Последовательное удаление двумерного массива...
for (i = 0; i < m1; i++)
{
    delete[]pArr[i];
}
delete[]pArr;

return 0;
}

```

Так как каждый столбец массива независимо выделяется динамически, можно сделать динамически

выделенные двумерные массивы, которые не являются прямоугольными.

Например, для нашего примера со студентами и их результатами тестирования по предметам (каждая выделенная отдельным цветом строка — одномерный динамический массив):

8	9	12	10	11	
9	11	11			
10	10	11	12		
8	7	8	8	8	8

Рисунок 4

Также из предыдущего опыта работы с динамическими массивами известно, что мы можем задавать их длину во время выполнения программы.

```
#include <iostream>

using namespace std;
int main()
{
    int i, j;

    // Переменные для описания характеристик массивов.
    // m1 — количество строк, m2 — количество столбцов
    int m1, m2;

    cout << "How many rows:\n";
    cin >> m1;

    int** pArr = new int* [m1];
    for (i = 0; i < m1; i++)
    {
```



```

    cout << "How many cells in the row-"<<i<<"?\n";
    cin >> m2;
    pArr[i] = new int[m2];

    // заполнение строки двумерного динамического
    // массива данными и вывод
    for (j = 0; j < m2; j++)
    {
        pArr[i][j] = i * j;
        cout << pArr[i][j] << " ";
    }
    cout << "\n";
}

// Последовательное уничтожение двумерного массива...
for (i = 0; i < m1; i++)
{
    delete[]pArr[i];
}
delete[]pArr;

return 0;
}

```

2. Примеры многомерных динамических массивов

Пример 1. Организация двумерного «треугольного» динамического массива

Сначала создаётся одномерный массив указателей, затем каждому элементу этого массива присваивается адрес одномерного массива. При этом размер (количество элементов) каждого нового массива на единицу меньше размера предыдущего. Включённая в квадратные скобки переменная, которая является операндом операции `new`, позволяет легко сделать это.

```
#include <iostream>

using namespace std;
int main()
{
    // m1 — количество строк, m2 — количество столбцов
    int m1 = 5, m2 = 5;
    int i, j, k;
    int** pXArr = new int* [m1];
    k = m2;

    // Организация двумерного динамического массива
    for (i = 0; i < m1; i++, k--)
    {
        pXArr[i] = new int[k];
    }

    for (i = 0; i < m1; i++, m2--)
    {
```

```

for (j = 0; j < m2; j++)
{
    // заполнение строки двумерного
    // динамического массива данными и вывод
    pXArr[i][j] = 5;
    cout << pXArr[i][j] << " ";
}
cout << "\n";
}

// Последовательное уничтожение двумерного массива...
for (i = 0; i < m1; i++)
{
    delete[]pXArr[i];
}
delete[]pXArr;

return 0;
}

```

Пример 2. Организация трехмерного динамического массива

Создание и уничтожение трёхмерного массива требует дополнительной итерации. Однако здесь также нет ничего принципиально нового.

```

#include <iostream>

using namespace std;
int main()
{
    // m1 — количество строк (высота),
    // m2 — количество столбцов (ширина), m3 — глубина

```

```

int m1 = 5, m2 = 3, m3 = 2;
int i, j;

// указатель на указатель на указатель :)
int*** pXArr = new int** [m1];

// Организация трехмерного динамического массива
for (i = 0; i < m1; i++)
{
    pXArr[i] = new int* [m2];
    for (j = 0; j < m2; j++)
    {
        pXArr[i][j] = new int[m3];
    }
}

// Доступ к элементам массива
pXArr[1][2][3] = 750;
cout << pXArr[1][2][3] << "\n";

// Удаление в последовательности, обратной созданию
for (i = 0; i < m1; i++)
{
    for (j = 0; j < m2; j++)
    {
        delete[] pXArr[i][j];
    }
    delete[] pXArr[i];
}

delete[] pXArr;

return 0;
}

```

3. Перечисляемые типы

Перечисление (перечисляемый тип данных) — это такой тип данных, где любое допустимое значение (которое называется перечислитель) представляет собой именованную (символьную) целочисленную константу.

Этот набор символьных констант определяет множество всех допустимых значений, которые может принимать переменная, относящаяся к данному типу.

Перечисления рекомендуется использовать в таких ситуациях, когда переменная может принимать значения из заранее известного конечного множества.

Например, названия четырех времен года: `WINTER`, `SPRING`, `SUMMER` и `AUTUMN`.

Перечисляемые типы полезны для повышения читабельности кода.

Особенно удобно использование перечислений при реализации сложных ветвлений или использовании многочисленных case-ов в операторе `switch`. Предположим, что нам необходимо проверить код дня недели (не понедельник ли сегодня). Если где-то в коде написать проверку на равенство `1` (предполагая, что код Пн — это `1`), то непонятно, что такое `1`: кг, литр, метр. А вот если вместо `1` написать `MON` (т.е. Monday), то сразу понятно, что речь идет о первом дне недели — понедельнике.

Помимо этого, используя перечисления, мы сгруппируем все допустимые значения дней недели в одной группе, обозначив каждую константу символьным и понятным названием.

Для создания перечислимого типа используется ключевое слово `enum` и некоторый набор значений, который определяет пользователь. Набор значений необходимо заключить в фигурные скобки. По факту это набор целых именованных констант, которые представляются своими идентификаторами. Эти константы называются перечислителями (или перечислимыми константами).

Рассмотрим объявление перечисляемого типа:

```
enum Seasons {WINTER, SPRING, SUMMER, AUTUMN};
```

С его помощью создается целочисленный тип — набор из четырех названий времен года, именующих целочисленные константы.

Перечислимые константы — это идентификаторы `WINTER`, `SPRING`, `SUMMER` и `AUTUMN`.

Идентификаторы перечисляемого типа (`Seasons` в примере выше) обычно начинаются с заглавной буквы, а имена перечислителей (`WINTER`, `SPRING`, `SUMMER`, `AUTUMN`) — пишутся полностью заглавными.

После объявления перечисления каждому его перечислителю автоматически ставится в соответствие целое число, значение которого зависит от его позиции в списке данного перечисления. По умолчанию первому перечислителю присваивается значение `0`, а каждому следующему — на `1` больше, чем предыдущему.

Переменным типа `Seasons`, который был определен пользователем, может быть присвоено только одно из четырех значений, объявленных в данном перечислении.

Рассмотрим еще один традиционный и популярный пример использования перечислений:

```
enum Months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL,
             AUG, SEP, OCT, NOV, DEC};
```

В данном примере первое значение перечислителя (JAN) установлено в 1 (номер месяца «январь» в году), таким образом, оставшиеся значения (FEB, MAR, APR,...) увеличиваются на 1: FEB=2, MAR=3 и т.д. (до 12 для DEC).

В качестве значений для констант перечисления можно использовать как положительные, так и отрицательные целые числа. Если для какого-то перечислителя не было определено значение, то оно будет назначено автоматически, на 1 больше, чем значение предыдущего перечислителя.

Примечание: после назначения значения константе перечисления попытка присвоить ей другое значение приведет к синтаксической ошибке.

Основные моменты использования перечислений.

1. Использование переменных перечисляемого типа вместо нескольких целых констант повышает читабельность кода.
2. Идентификаторы перечислителей внутри перечисления должны быть уникальными, но при этом допускается наличие нескольких констант с одинаковыми целыми значениями.
3. Константы перечислений могут инициализироваться не только любыми целочисленными значениями, но и выражениями, например:

```
enum Ages {JOE = 45, MARIA, BOB = 25, SUE = JOE + 2};
```

В данном примере для перечислителя **MARIA** не было назначено значение явным образом, т.е. было применено правило по умолчанию: значение предыдущей константы $(JOE = 45) + 1 = 46$.

4. Каждое перечисление является отдельным типом. Типом элемента перечисления является само перечисление, т.е. **JOE** имеет тип **Ages**.
5. Можно объявить перечислимую константу анонимно, т.е. не указывая идентификатор типа.

```
enum {SINGLE, MARRIED};
```

Это распространенный способ объявления мнемонических целочисленных констант

```
enum {SINGLE, MARRIED} status;
```

Второй пример объявляет переменную перечислимого типа **status**, с допустимыми значениями этой переменной **SINGLE** и **MARRIED**.

6. Неявное преобразование перечислений в обычные целочисленные типы возможно, обратное преобразование — нет.

```
enum myBoolean {FALSE, TRUE} b;
enum signal {OFF, ON} a = ON; // а инициализируется в ON
enum answer {NO, YES, MAYBE = -1} c;
int i, j = true; // верно true преобразуется в 1
a = OFF; // верно
i = a; // верно i становится 1
b = a; // неверно два различных типа
b = (boolean)a; // верно, явное преобразование типов
```


Рассмотрим практический пример с использованием перечисляемого типа данных.

Предположим, что нам необходимо реализовать меню для пользователя с двумя элементами: ввод нового логина (изменение логина, предусмотренного по умолчанию) и вывод данных (логина пользователя). Пользователь вводит код пункта меню и, если такой код предусмотрен, то выполняется часть кода, реализующая логику данного пункта меню.

Для проверки введенных кодов пунктов меню будем использовать оператор `switch`.

Для улучшения читабельности кода реализуем пункты меню с использованием перечисляемого типа данных. Каждому перечислителю автоматически присваивается целочисленное значение в зависимости от его позиции в списке перечисления. По умолчанию, первому перечислителю присваивается целое число `0`, а каждому следующему — на единицу больше, чем предыдущему. Так как нам необходимо задать для первого элемента меню «Ввод данных» значение `1`, то выполним присваивание нужных нам значений перечислителей.

```
#include <iostream>

using namespace std;

enum menuItems
{
    ENTER_DATA = 1,
    OUTPUT_DATA=2,
    QUIT=3
};
```

```

int main()
{
    int userChoice;
    char userLogin[20]= "admin";

    do {
        cout << "Your choice:\n";
        cout << "1 - enter data (login)\n";
        cout << "2 - output data\n";
        cout << "3 - quit\n";
        cin >> userChoice;

        switch (userChoice) {
            case ENTER_DATA:
            {
                cout << "Please, enter data \n";
                cin >> userLogin;
                break;
            }
            case OUTPUT_DATA:
            {
                cout << "Your login:"<< userLogin<<"\n";
                break;
            }
            case QUIT:
            {
                cout << "See you!";
                break;
            }
            default:
                cout << "Wrong menu item!";
        }
    } while (userChoice != 3);

    return 0;
}

```

Результаты работы программы:

```

Your choice:
1 - enter data (login)
2 - output data
3 - quit
2
You login:admin
Your choice:
1 - enter data (login)
2 - output data
3 - quit
1
Please, enter data
Anna
Your choice:
1 - enter data (login)
2 - output data
3 - quit
2
You login:Anna
Your choice:
1 - enter data (login)
2 - output data
3 - quit

```

Рисунок 5

Указатели на функции

Из предыдущих уроков нам известно, что указатель — это переменная, которая хранит адрес другой переменной в памяти.

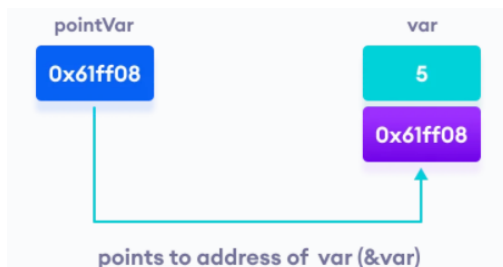


Рисунок 6

Указатели на функции имеют аналогичное назначение, только вместо адресов обычных переменных в памяти они хранят адреса функций (указывают на функции).

Ведь, как и обычные переменные, функции хранятся в памяти и, соответственно, имеют свой адрес.

Прежде чем мы начнем создавать и работать с указателями на функции, необходимо вспомнить, что любая функция характеризуется: именем функции; типом значения, которое возвращает данная функция; параметрами (а именно, типами параметров функции и порядком их следования в списке параметров).

1. Если мы используем имя функции без последующих круглых скобок и параметров, то в этом случае имя функции будет указателем на эту функцию, а его значение — это адрес размещения данной функции в памяти. Например,

```
// код функции myF находится, например, в ячейке
// памяти 005B1051
int myF()
{
    return 7;
}

int main()
{
    // переходим к адресу 005B1051
    // и выполняем код по адресу 005B1051
    myF();
}
```

Идентификатор `myF` — это имя нашей пользовательской функции.

Когда функция вызывается (с помощью оператора `()`), точка выполнения основной программы переходит к адресу этой вызываемой функции.

`myF` без последующих круглых скобок — это адрес размещения функции в памяти (`005B1051`).

Например, если в предыдущем коде внести следующие изменения в строку, использующую функцию `myF`:

```
int main()
{
    // выводим в консоль адрес функции
    cout<<myF;
    return 0;
}
```

Аналогичным способом вывода адреса функции в консоль может быть следующий (с использованием оператора `&`, который возвращает адрес операнда, в данном случае — функции):

```
cout << &myF;
```

То в консоль будет выведен адрес функции `myF`, так как мы отправили указатель на функцию `myF` непосредственно в `cout` (вместо вызова функции `myF()`)

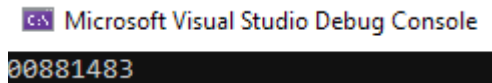


Рисунок 7

2. Это значение (адрес нашей функции `myF` в памяти) можно присвоить указателю, который далее в программе будет использоваться для вызова функции `myF`.

3. Когда мы определяем указатель, то нужно, чтобы тип возвращаемого значения, количество, типы параметров и порядок их следования совпадали с соответствующими характеристиками функции, адрес которой и будет хранить наш указатель (после его инициализации или с помощью оператора присваивания).
4. Для определения указателя на функцию используется следующий синтаксис:

```
тип_функции (*имя_указателя) (спецификация_параметров);
```

Например:

```
int (*func111Ptr) (char);
```

Мы определили указатель `func111Ptr` на функцию с одним параметром типа `char`, которая возвращает значение типа `int`.

Скобки вокруг `*func111Ptr` необходимы для соблюдения приоритета операций.

В противном случае `int *func1Ptr (char)` будет интерпретироваться как предварительное объявление (прототип) функции `func1Ptr` с параметром типа `char`, которая возвращает указатель на целочисленное значение.

Второй пример:

```
char * (*func222Ptr) (char *, int);
```

Мы определили указатель `func222Ptr` на функцию с двумя параметрами: первый — типа указатель на `char` и второй — типа `int`, которая возвращает значение типа указатель на `char`.

Иллюстрируем на практике

Рассмотрим простейшую программу, использующую указатель на функцию.

Вначале создадим две простые функции без параметров, которые не возвращают результат (просто выводят разные строки текста в консоль).

```
void f1()      // Определение f1.
{
    cout << "Load f1()\n";
}

void f2()      // Определение f2.
{
    cout << "Load f2()\n";
}
```

Теперь нам нужно объявить указатель. Нам достаточно одного указателя, т.к. у наших двух функций совпадают параметры и тип возвращаемого значения.

```
void (*ptr)();
```

Теперь нам необходимо инициализировать наш указатель на функцию функцией, т.е. присвоить указателю адрес функции, следующим образом:

```
ptr = f2;
```

После этой строки кода адрес функции `f2` находится в переменной `ptr`.

Теперь мы можем использовать указатель (`ptr`) на функцию `f2` для вызова самой функции `f2`.

Для этого нам нужно выполнить операцию разыменования указателя:

```
(*ptr)(); // вызываем функцию f2(), используя ptr
```

Общий код нашего примера:

```
#include <iostream>
using namespace std;

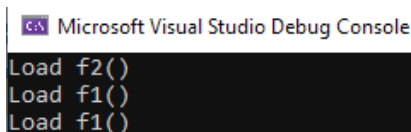
void f1()    // Определение f1.
{
    cout << "Load f1()\n";
}

void f2()    // Определение f2.
{
    cout << "Load f2()\n";
}

int main()
{
    void (*ptr)(); ptr — указатель на функцию.
    ptr = f2; // Присваивается адрес f2().
    (*ptr)(); // Вызов f2() по ее адресу.
    ptr = f1; // Присваивается адрес f1().
    (*ptr)(); // Вызов f1() по ее адресу.
    ptr(); // Вызов эквивалентен (*ptr)();

    return 0;
}
```

Результат выполнения программы:



```
Microsoft Visual Studio Debug Console
Load f2()
Load f1()
Load f1()
```

Рисунок 8

Здесь значением переменной `ptr` является адрес функции. Использование оператора разыменования для данного указателя (`*ptr`) дает нам возможность обращаться к этой функции по ее адресу.

Однако следует помнить, что использование только фрагмента кода `*ptr` без последующих круглых скобок предоставляет только адрес функции и такая строка приведет к ошибке. Для запуска функции по адресу, который мы храним в переменной `ptr` нам нужно использовать `*ptr()`.

Но и здесь есть важная особенность: использование `*ptr()` вместо (`*ptr`)() приведет к ошибке, т.к. у операции `()` более высокий приоритет, чем у операции обращения по адресу `*`. То есть при записи `*ptr()` программа вначале будет пытаться обратиться функции `ptr()`, которой нет, что вызовет синтаксическую ошибку. А уже потом будет выполняться операция разыменования.

В строке определения указателя на функцию мы можем также его инициализировать значением адреса некоторой функции. Опять же, следует помнить, что тип и сигнатура этой функции должны соответствовать определяемому указателю.

Это же правило нужно соблюдать и при выполнении операции присваивания указателю адреса некоторой функции или значения другого указателя на функцию.

При использовании указателей для вызовов функции это правило также имеет место: типы, количество, порядок следования фактических параметров, используемых при обращении к функции по адресу, должны соответствовать формальным параметрам вызываемой функции.

Рассмотрим перечисленные выше особенности на практике:

```
// Определение функций
char f1(char)
{
    //...
}

char f2(int)
{
    //...
}

void f3(float)
{
    //...
}

int* f4(char *)
{
    //...
}

int main()
{
    // Указатели на функции
    char (*pt1)(int);
    char (*pt2)(int);
    void (*ptr3)(float) = f3; // Инициализированный
                             // указатель.

    pt1 = f1;    // Ошибка - несоответствие сигнатур.
    pt2 = f3;    // Ошибка - несоответствие типов
                 // (значений и сигнатур).
    pt1 = f4;    // Ошибка - несоответствие типов.
    pt1 = f2;    // Правильно.
    pt2 = pt1;   // Правильно.
```

```

char c = (*pt1) (44); // Правильно.

c = (*pt2) ('\t'); // Ошибка — несоответствие
                  // сигнатур.

return 0;
}

```

Следующая программа отражает гибкость механизма вызовов функций с помощью указателей.

```

#include <iostream>

using namespace std;
int add(int n, int m)
{
    return n + m;
}

int division(int n, int m)
{
    return n / m;
}

int mult(int n, int m)
{
    return n * m;
}

int subtr(int n, int m)
{
    return n - m;
}

int main()
{
    int (*par) (int, int); // Указатель на функцию.
}

```

```

int a = 6, b = 2;
char c = '+';

par = add;

while (c!= ' ')
{
    cout << "\n Arguments: a = " << a << ", b = "
        << b;
    cout << ". Result for c = '\" << c << "\"':";
    switch (c)
    {
        case '+':
        {
            par = add;
            c = '/';
            break;
        }
        case '-':
        {
            par = subtr;
            c = ' ';
            break;
        }
        case '*':
        {
            par = mult;
            c = '-';
            break;
        }
        case '/':
        {
            par = division;
            c = '*';
            break;
        }
    }
}

```

```

    a = (*par)(a, b); // Вызов по адресу
    cout << a << "\n";
}

return 0;
}

```

Цикл продолжается, пока значением переменной `c` не станет пробел. В каждой итерации указатель `par` получает адрес одной из функций, и изменяется значение `c`. По результатам программы легко проследить порядок выполнения ее операторов.

Внимание!

- *Параметры по умолчанию не будут работать с функциями, вызванными через указатели на функции.*
- *Параметры по умолчанию обрабатываются во время компиляции (т.е. вам нужно предоставить аргумент для параметра по умолчанию во время компиляции).*
- *Однако указатели на функции обрабатываются во время выполнения.*
- *Следовательно, параметры по умолчанию не могут обрабатываться при вызове функции через указатель на функцию.*
- *В этом случае вам нужно будет явно передать значения для параметров по умолчанию.*

Рассмотрим на примере.

Предположим, что у нас есть функция, возвращающая результат сложения двух чисел, которые передаются через

параметры. Причем второй параметр — это параметр по умолчанию.

```
int myF(int a, int b = 5)
{
    return a+b;
}
```

При вызове данной функции с одним параметром второй подставляется по умолчанию (5 в нашем примере):

```
cout << myF(2);
```

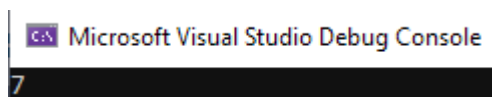


Рисунок 9

Попробуем повторить вызов функции `myF(2)` через указатель.

```
int (*ptr)(int, int);
ptr = myF;
cout<<(*ptr)(2);
```


 **C2198** 'int (__cdecl*)(int,int)': too few arguments for call

Рисунок 10

Изменим код, задав значение второго параметра:

```
int (*ptr)(int, int);
ptr = myF;
cout<<(*ptr)(2, 7);
```

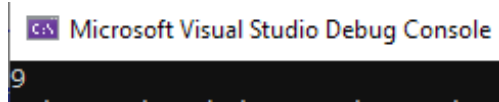


Рисунок 11

Таким образом, использование параметров по умолчанию с функциями, вызванными через указатели на функции, не дает явных преимуществ, т.к. в вызове функции в любом случае необходимо явно указывать их значения.

Массивы указателей на функции

Для удобной работы с набором указателей на функции рекомендуется использовать массивы указателей. В таких массивах значением элементов является указатель на определенную функцию.

Например,

```
float (*ptrArray[4]) (char)
```

Здесь мы объявили массив из четырех элементов (каждый из которых является указателем на разные функции, но с одинаковым типом возвращаемого значения, *float*, и одинаковой сигнатурой: один параметр типа *char*).

Для того, чтобы запустить, например, третью функцию (адрес которой храниться во втором элементе массива) можно использовать такой подход:

```
float num = (*ptrArray[2]) ('f');
```

Особенно удобным использование массивов указателей является при создании меню, каждый элемент которого реализуется отдельной функцией.

Адреса этих функций хранятся в массиве указателей на функции.

Пользователю будет предлагаться выбрать (ввести) номер желаемого пункта меню и этот номер далее мы будем использовать как индекс, по которому нужно считать из массива адрес нужной функции и запустить ее для выполнения нужной пользователю операции.

Предположим, что наша программа предназначена для обработки файлов.

```
#include <iostream>
using namespace std;

#include <iostream>
using namespace std;

/* Определение функций для обработки меню
   (функции фиктивны т. е. реальных действий
   не выполняют):
*/

void act1(char* name)
{
    cout << "Create file..." << name;
}

void act2(char* name)
{
    cout << "Delete file... " << name;
}

void act3(char* name)
{
    cout << "Read file... " << name;
}
```



```

void act4(char* name)
{
    cout << "Mode file... " << name;
}

void act5(char* name)
{
    cout << "Close file... " << name;
}

int main()
{
    //Создание и инициализация массива указателей
    void (*MenuAct[5])(char*) = {act1, act2, act3,
                                   act4, act5};

    int number; // Номер выбранного пункта меню.
    char FileName[30]; // Строка для имени файла.

    // Реализация меню
    cout << "\n 1 - Create";
    cout << "\n 2 - Delete";
    cout << "\n 3 - Read";
    cout << "\n 4 - Mode";
    cout << "\n 5 - Close";

    while (1) // Бесконечный цикл
    {
        while (1)
        {
            /* Цикл продолжается до ввода
               правильного номера.*/
            cout << "\n\nSelect action: ";
            cin >> number;
            if ((number >= 1) && (number <= 5))
            {
                break;
            }
        }
    }
}

```

```

        else
        {
            cout << "\nWrong number!";
        }
    }
    if (number != 5)
    {
        cout << "Enter file name: ";
        cin >> FileName; // Читать имя файла.
        // Вызов функции по указателю на нее
        (*MenuAct[number - 1])(FileName);
    }
    else
    {
        break;
    }
}
return 0;
}

```

Пункты меню повторяются, пока не будет введен номер 5 — закрытие.

Ключевые слова `auto` и `decltype`.

Автоматическое выведение типа

Вплоть до C++11 программистам нужно было явно указывать тип данных переменной до ее использования в коде.

Например:

- объявление переменной с последующим присвоением ей значения:

```

double myVar;
myVar = 4.2;

```

- инициализация переменной (присвоение переменной значения при ее объявлении)

```
double myVar = 4.2;
```

Рассмотрим последний пример. Компилятору и так известно, что `4.2` — это литерал типа `double`, т.е. явное указание перед именем переменной `myVar` типа `double` кажется излишним. Полезным представляется наличие у переменной возможности принять нужный тип данных, основываясь на соответствующем инициализируемом значении.

Начиная с C++ 11 мы имеем эту возможность, благодаря ключевому слову `auto`, при использовании которого тип переменной будет определяться самим компилятором при ее инициализации. Такой подход называется автоматическим выводением (определением) типа.

Рассмотрим на примере:

```
auto myVar = 4.2;
```

`4.2` — это литерал типа `double`, поэтому и переменная `myVar` «приобретет» тип `double`.

Ключевое слово `auto` можно использовать и с выражениями:

```
auto myExpr = 2 + 5;
```

Выражение `2 + 5` будет обработано компилятором как целочисленное, поэтому и переменная `myExpr` «приобретет» тип `int`.

Конечно же, ключевое слово `auto` можно использовать и при работе с указателями:

```
auto ptr = &myVar;
```

В этом случае `ptr` будет хранить адрес вещественной переменной `myVar` со значением 4.2.

Внимание! Данный подход будет корректно работать только с инициализированными переменными, т. к. простое объявление переменной (без инициализации ее значением) не предоставит компилятору информации о том, какой тип нужно автоматически назначить.

В C++ 14 функциональность ключевого слова `auto` была расширена возможностями автоматического определения типа возвращаемого значения функции.

Рассмотрим возможности и особенности ключевого слова `auto` при работе с функциями на примерах.

Пусть необходимо реализовать функцию, находящую минимум из двух чисел.

```
#include <iostream>
using namespace std;

int minF()
{
    int a = 20;
    int b = 10;
    if (a < b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

```
int main()
{
    auto myMin = minF();
    cout << myMin;
    return 0;
}
```

В этом примере в функции `main()` тип значения, возвращаемого функцией `minF()` определяется автоматически.

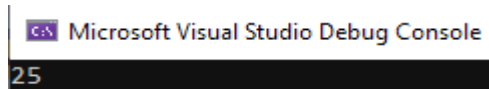


Рисунок 12

В этом есть смысл, если мы заранее не знаем, какой тип результата мы получим после запуска некоторой функции.

Промоделируем данную ситуацию более конкретно (ведь в предыдущем примере кода тип результата, возвращаемого функцией `minF()`, очевиден).

Пусть значение переменных `a` и `b` в функции `minF()` определяется автоматически и к найденному минимуму нужно добавить `10.5`.

Тогда и тип возвращаемого результата в объявлении функции `minF()` также следует заменить на `auto`

```
#include <iostream>
using namespace std;

auto minF()
{
    auto a = 20;
    auto b = 10;
```

```

    if (a < b)
    {
        return a+10.5;
    }
    else
    {
        return b+10.5;
    }
}

int main()
{
    auto myMin = minF();
    cout << myMin;
    return 0;
}

```

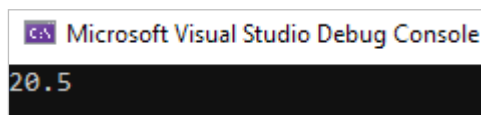


Рисунок 13

В этой ситуации использование `auto` вместо типа возвращаемого значения функции не говорит компилятору о необходимости автоматического определения типа, а только дает ему команду искать возвращаемый функцией тип в конце кода тела функции. То есть, так как тип результата выражения `10.5+10` (срабатывает ситуация `a < b`) является вещественным, то компилятор делает вывод, что и функция `minF()` должна быть типа `double`.

Несмотря на то, что приведенный выше способ кажется удобным, такой подход не рекомендуется по следующим причинам.

Если мы используем ключевое слово `auto` при определении переменной, то мы в правой части выражения указываем значение (выражение), на основании которого компилятор определит тип данных этой новой переменной. В случае же с функциями ситуация иная: у компилятора нет такого контекста, по которому он всегда однозначно может интерпретировать тип возвращаемого значения. А в случае, когда у функции есть несколько точек возврата (несколько `return` в теле функции), то при разных типах данных, возвращаемых после `return`, вообще будет ошибка на этапе компиляции.

Рассмотрим эту возможную ошибочную ситуацию на примере. Предположим, что нам нужно создать функцию, которая в зависимости от кода товара определяет размер скидки. И в одних случаях размер скидки — целочисленное значение, а в других — вещественное.

Попытка применить в этой задаче `auto` для функции, например, таким образом:

```
auto checkDiscount(int prodID)
{
    if (prodID == 1)
    {
        return 10;
    }

    else
    {
        return 20.5;
    }
}
```

Выдаст ошибку:

```
deduced return type "double" conflicts with previously deduced type "int"
```

Рисунок 14

Таким образом, предположение, что использование `auto` для автоматического определения типа возвращаемого функцией значения, будет работать в ситуации с разными типами данных (переменных), возвращаемых после `return` — ошибочно.

Наряду с ключевым словом `auto` в обновлённом стандарте C++11 появилось и ключевое слово `decltype`. Ключевое слово `decltype` используется для вывода типов выражений, получаемых в качестве своего аргумента.

Общий синтаксис:

```
decltype (expression) variable_name;
```

```
int x = 1.2;
decltype(x) j;
```

Выражение `decltype(x)` извлекает тип `x`, которым является `int`, и компилятор создает переменную `j` типа `int`.

Результат работы с использованием `auto` будет аналогичным:

```
int x = 1.2;
auto j=x;
```

Но в этом случае `auto` «указывает» компилятору заполнить пропущенный тип переменной `j`, автоматически определив его на основе выражения справа.

Рассмотрим еще один пример:

```
auto a1 = 0; // int
decltype (a1) a2 = a1; // same as auto a2=a1;
typedef decltype (a1) ATYPE; // ATYPE is a synonym
                                // for int
XTYPE a3 = 5;
```

Как это работает? Выражение `decltype(a1)` извлекает тип `a1`, которым является `int`. Этот тип назначается новому типу `ATYPE`. Затем `ATYPE` используется в объявлении новой переменной `a3`, которая имеет тот же тип, что и `a1`.

Использование вместо

```
typedef decltype (a1) ATYPE;
```

выражения

```
typedef auto ATYPE=a1;
```

приведет к ошибке.

Основное различие между `decltype` и `auto` заключается в типе возвращаемого значения.

Рассмотрим следующий код:

```
int y = 10;
int& r = y;
auto x = r;
```

Тип переменной `x` был автоматически определен как целочисленный (`int`) с помощью `auto`, т.е. ссылка была проигнорирована

Следующий пример:

```
int y = 10;
const int& r = y;
auto x = r;
```

показывает, что и квалификатор `const` и `&` (оба) были проигнорированы `auto` (тип переменной `x` был автоматически определен как целочисленный)

Теперь посмотрим, как в этой ситуации поведет себя `decltype`

```
#include <iostream>
using namespace std;

int main()
{
    int y = 10;
    int &r = y;
    int x = 5;
    decltype(r) var=x;
    var = 12;
    cout<<x;
    return 0;
}
```

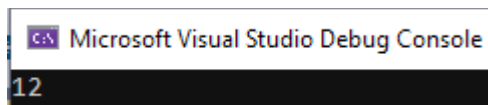


Рисунок 15


Как показывают результаты работы программы переменная `var` была создана как ссылка, что позволило с ее помощью изменить значение переменной `x`.

Рассмотрим еще один пример использования `decltype`

```
#include <iostream>
using namespace std;

int main()
{
    int x = 5;
    decltype(x) j = x + 5;

    cout << j;
    return 0;
}
```

 Microsoft Visual Studio Debug Console

10

Рисунок 16

В этом примере переменная `j` такого же типа, как и переменная `x` (`int`), и инициализирована значением выражения `x + 5`.

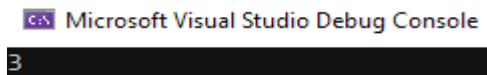
Теперь рассмотрим использование `decltype` с функциями:

```
#include <iostream>

using namespace std;
int myF()
{
    int a = 20;
    return a + 5;
}

int main()
{
    decltype(myF()) varF=3;
}
```

```
cout << varF;
return 0;
}
```


 Microsoft Visual Studio Debug Console

3

Рисунок 17

Строка кода `decltype(myF()) varF=3` извлекла тип возвращаемого функцией `myF()` значения (`int`) и инициализировала переменную этого типа `varF` значением `3`.

`decltype` является не альтернативой, а расширением (дополнением) возможностей `auto` (например, при работе со ссылками или при использовании `typedef`, как в примерах выше).

Хвостовой возвращаемый тип

В стандарте C ++ 11 появилась новая возможность в синтаксисе объявления функций — указать **trailing return type** (или **tail return type**, «хвостовой возвращаемый тип»).

В этом случае необходимо поместить тип значения, которое возвращается функцией, в конечную часть («хвост») прототипа функции.

В традиционном синтаксисе при объявлении функции этот тип данных указывается левой части, перед именем данной функции, например:

```
double myF(double, int);
```

Стандарт C ++ 11 позволяет помещать тип возвращаемого значения в конечную часть («хвост») прототипа

функции. Этот подход требует помещения ключевого слова **auto** в левую часть прототипа, перед именем функции, например:

```
auto myF(double, int) -> double;
```

Или, в случае без прототипа:

```
auto myF(double x, int y) -> double {
    //.....
};
```

В этом случае ключевое слово **auto** не выполняет вывод типа, а является только лишь частью синтаксиса.

При использовании этого подхода, компилятор «делает выводы» о типе возвращаемого значения по конечной части прототипа функции (после стрелки).

В этом случае в теле функции после ключевого слова **return** должна быть переменная (выражение, литерал) соответствующего типа.

Например, для нашего случая:

```
auto myF(double x, int y) -> double
{
    return x + y;
};
```

Для простых функций (как в нашем примере) эта новая возможность не дает никакого преимущества. Однако в ситуациях, когда тип возвращаемого значения сложнее, подход с использованием хвостового возвращаемого типа делает код более понятным и кратким, чем при традиционном синтаксисе.

Например, если наша функция возвращает указатель на массив, то в случае использования традиционного синтаксиса ее объявление будет иметь вид:

```
int (*myFunc1 (int arr[][5], int n)) [5]
{
    return &arr[n];
}
```

Здесь наша функция `myFunc1` принимает двумерный массив `arr`, состоящий из строк (их количество задается вторым параметром `n`) и пяти столбцов (фактически данный массив в качестве элементов содержит `n` массивов из пяти элементов).

Напомним, как выглядит объявление указателя на двумерный динамический массив (в случае, когда правый индекс, количество столбцов, константа):

```
int (*arr)[5] = new int[n][5];
```

Скобки `()` здесь потребуются для соблюдения приоритета.

Соответственно, если наша функция возвращает указатель на двумерный динамический массив, то перед именем функции размещается символ `*` и имя функции с ее аргументами помещается в скобки для соблюдения приоритета:

```
int (*myFunc1 (int arr[][5], int n)) [5]
```

Когда разработчик использует принципы хвостового возвращаемого типа, то понять, какого типа будет значение, возвращаемое функцией, будет проще.

```
auto myFunc1(int arr[][5], int n) -> int(*)[5]
{
    return &arr[n];
}
```

Здесь мы использовали ключевое слово `auto` для автоматического определения типа данных и, согласно принципам использования хвостового возвращаемого типа, перенесли тип `int(*)[5]` после символа `->`, оставив слева только имя функции и ее аргументы.

В случае, когда сложность возвращаемого типа функции еще больше возрастет, то преимущества нового подхода станут еще более очевидны. Например, нам нужно определить функцию, которая возвращает указатель на функцию:

```
auto myFunc1(int arr[][5], int n) -> int(*)[5]
{
    return &arr[n];
}

int(*(*myFunc2())(int arr[][5], int n))[5]
{
    return myFunc1;
}
```

Так как функция `myFunc2` возвращает указатель на функцию `myFunc1`, которая, в свою очередь, возвращает указатель на двумерный динамический массив:

```
int(*myFunc1(int arr[][5], int n))[5]
```

то при объявлении функции `myFunc2` мы используем часть объявления функции `myFunc1 (int arr[][5], int n))[5]`

и покажем, что `myFunc2` возвращает указатель, добавив перед `(*myFunc2())` еще один символ `*` и обернув вторыми скобками для соблюдения приоритета `int>(*myFunc2())`.

Использованием хвостового возвращаемого типа в объявлении функции `myFunc2()` будет выглядеть читабельнее:

```
auto myFunc2() -> int (*)(int arr[][5], int n)[5]
{
    return myFunc1;
}
```

Также при традиционном подходе к объявлению функций программисту нужно оценивать, каким именно будет тип возвращаемого значения, анализируя код тела функции.

Используя хвостовой возвращаемый тип в объявлении совместно с ключевым словом `decltype`, мы можем «поручить» процесс этой оценки компилятору.

```
#include <iostream>
using namespace std;

auto myFunc1(float a, int b, int c) -> decltype((a+b)/c)
{
    return (a + b)/c;
}

int main()
{
    cout << myFunc1(5, 2, 3) << "\n";
    cout << myFunc1(1, 1, 1);
    return 0;
}
```

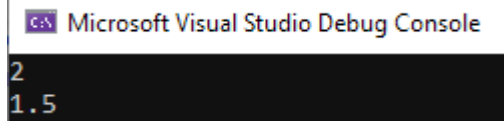



Рисунок 18

В нашем примере аргументы функции имеют как целочисленный, так и вещественный тип. Как видно из результатов работы программы, возможны ситуации и с целочисленным, и с вещественным результатом. Однако процесс оценки «ситуации» и подбора соответствующего типа для возвращаемого функцией значения будет выполнен компилятором, а не программистом.

Рассмотрим следующий пример, когда внутри одной функции вызывается другая.

```
auto myFunc1(float a, int b, int c)->decltype((a+b)/c)
{
    return (a + b)/c;
}

auto myFunc2(float a, int b, int c, int d)->
    decltype(myFunc1(a,b,c)/d) {
    return myFunc1(a, b, c)/d;
}
```

Допустим, что код (тело функции) `myFunc1()` программисту недоступен. В этом случае предугадать, какого типа будет результат, возвращаемый функцией `myFunc1()` в точку ее вызова в `myFunc2()` не просто.

Используя хвостовой возвращаемый тип в объявлении `myFunc2()` совместно с ключевым словом `decltype`, мы избавились от этой неоднозначности.