Лекция № 4

Инициализация переменных

Программа на языке С (и С++) является инструментом по переработке входных данных в промежуточные и, в конечном итоге, выходные. Любая переменная программы в любой момент времени, содержит некоторые данные. После того как программа положила в переменную необходимое значение, там хранится это значение. До этого же в переменной хранится совершенно непредсказуемое значение, которое сформировалось в этой области памяти в результате работы операционной системы и предыдущих программ. То есть, если такая переменная будет участвовать в вычислении некоторого выражения, то и результат такого вычисления да результат работы всей программы будут непредсказуемы. Отсюда вытекает одно из основных правил разработки программ.

Правило. Прежде чем переменная будет впервые использована в вычислении одного из выражений программы, она должна быть проинициализирована.

Другими словами, прежде чем переменная будет упомянута в правой части оператора присваивания или в логическом выражении условного оператора, в нее должно быть помещено значение, которое необходимо по алгоритму решения задачи.

Определение. Под *инициализацией* переменной понимают присваивание переменной некоторого значения до первого использования этой переменной в программе.

Инициализировать переменную можно несколькими способами.

Первый способ заключается в применении *оператора присваивания* выражение которого есть константа или константное выражение, или же все переменные, использованные в выражении, уже были проинициализированы.

Пример.

```
a = 5.0, b = 1.0, c = 100.0; x = 3 + a; y = b / c – x;
```

Второй способ заключается во вводе значения в переменную посредством оператора вдова.

Пример.

```
cin >> a; cin >> b >> c >> x;
```

Третий способ заключается в инициализации переменной в операторе описания. Язык С (и С++) позволяет совмещать оператор описания переменной и оператор присваивания. То есть, если соответствующие переменные ранее небыли описаны, то в языке С (и С++) следующие конструкции являются корректными.

Пример.

```
double a = 5.0, b = 1.0, c = 100.0;
long double x = 3 + a,  y = b / c – x;
```

Инициализировать требуется не только простые переменные базовых типов, но и массивы. Так как ни оператор присваивания, ни оператор ввода не предназначены для работы с массивами, то инициализации приходится подвергать каждый элемент массива:

Пример.

```
double a [15], b [10];
a [0] = 5.0, a [1] = 1.0, a [2] = 100.0, …;
cin >> b [0] >> b [1] >> b [2] >> …;
```

Инициализация массива при его описании имеет особенности. Для инициализации массива достаточно перечислить значения, которые должны быть размещены в элементах массива. Значения заключаются в фигурные скобки и перечисляются через запятую.

Пример.

```
double a [5] = {2.3, 34, 5.01, 2, -12.3}, b [4] = {-3.2, -4, -13, 5.1};
```

Язык C (и C++) позволяет указывать значения не для всех элементов массива, а только для начальных. Остальные, конечные, элементы массива, для которых не указаны значения, инициализируются значением **0** (ноль).

Правило. При описании массива его элементы могут проинициализированы. Начальные значения элементов массива указываются в фигурных скобках через запятую. Значений, инициирующих элементы массива, должно быть не больше чем длина массива, указанная при его описании. Если значений перечислено меньше чем длина массива, то конечные элементы, которым не хватило значений, инициируются автоматически значением ноль.

Пример.

```
double a [15] = {2.3, 34, 5.01, 2, -12.3};
        // пять первых элементов a[0]÷a[4] инициализировано
        // перечисленными значениями;
        // остальные элементы a[5]÷a[14] получат значение 0
```

Помимо этого при таком описании массива можно явно не указывать длину массива. Тогда компилятор устанавливает длину массива равную количеству инициализирующих значений.
Пример.

```
double a [] = {2.3, 34, 5.01, 2, -12.3}; // описание массива длиной 5
```

Инициализировать во время описания можно и двумерные массивы. При этом значения инициализации можно перечислять единым списком или группировать для лучшего восприятия текста программы.
Пример.

```
int a [2][3] = {2, 3, 5, -2, -4, 8}; // описание матрицы
int b [2][3] = {{2, 3, 5}, {-2, -4, 8}}; // описание матрицы
```

В данном примере матрицы a и b имеют одинаковые размеры и инициализированы одинаковыми значениями.

Аналогично массивам из базовых целочисленных или вещественных типов инициализировать можно и массивы из элементов типа char, которые хранят строки текста.
Пример.

```
char str1 [20] = “Привет Томску!”;
```

Здесь, конечно же, длина текста должна быть меньше длины строки, так как в массиве должен еще разместиться и завершающий роль. И также как и в случае с другими массивами можно не указывать длину массива типа char явно. Тогда длина будет определена компилятором по длине инициализирующего текста, но больше на один элемент для размещения завершающего нуля (элемента с кодом ‘\0’).

```
char str2 [] = “Привет Томску!”;
```

Здесь длина массива компилятором устанавливается равной 15, то есть 14 ячеек под инициализационный текст и ячейка под замыкающий код ‘\0’.

Указатели

При разработке сложных программ часто удобно оперировать не только переменными и массивами, но и адресами их размещения. Но всё чем оперирует программа должно храниться в ячейках памяти определенного типа. Такими ячейками являются переменные базового типа: целочисленные, вещественные и т.п., а для хранения адресов — указатели.

Определение. Указатель — это переменная, которая может хранить адреса любых ячеек памяти базовых типов, в том числе и тех, где хранятся другие адреса переменных.

Если переменная-указатель a хранит адрес целой переменной b, говорят, что a указывает на b. Добраться до *данных (значений)* через указатель, на которые тот *указывает*, так же просто как если бы переменная-указатель хранила сами данные.

Синтаксис описания элемента типа указатель:

<*спецификация типа*> ∗<*идентификатор*>;

Пример.

```
int *i; // говорят, что значение, записанное по адресу i, имеет тип int
double *g;
char *d;
```

Для определения адреса размещения переменной с целью оперирования с ним или помещения его в указатель используют операцию *определения адреса* ‘&’.

Пример.

```
double *y;
double x;
y = &x;
```

Таким образом, оператором y=&x присваивается *адрес* x некоторой *переменной* y.

Предупреждение. Операцию *определения адреса* ‘&’ можно применять только к переменным и элементам массива. К выражениям, константам и выражениям, содержащим константы, и другим конструкциям операцию *определения адреса* ‘&’ нельзя применять. Например, выражения &(x + 7) и &28 недопустимы.

Извлечь значение, содержащееся по адресу, можно с помощью операции *обращение по адресу* ‘*’. То есть значение, хранимое в памяти, может быть доступно не только по имени переменной, но и по указателю, если он ссылается на эту же область памяти.

Пример.

```
double *y, x, z;
y = &x; x = 3.1415926;
z = *y;
```

В итоге и переменная x и переменная z имеют значение 3.1415926. При этом указатель y ссылается на ту же область памяти, что и переменная x. Естественно, что можно не только читать данные по адресу, хранящемуся в указателе, но засылать данные по этому адресу, например, *y = z + 2.71828;.

Указатели и массивы

Описание массива связано тесно с указателем. Рассмотрим два описания. Первое

```
double a[5];  // a[0], ... , a[4];
```
(1)

Здесь a — является константой-указателем, которая принимает значение адреса массива, то есть фактически адрес местоположения первого элемента массива — a[0]. Второе

```
double *y;
```
(2)

Здесь y это указатель на тип double, но в то же время этот указатель можно рассматривать и как массив с начальным адресом y, но с нулевым числом элементов (то есть без элементов). Необходимо помнить, что компилятор отводит память только *под указатель* y и *ничего* под сам *массив*. Указатель y принимает неизвестное значение, до тех пор, пока ему не будет присвоено конкретное значение.

У таких описаний два принципиальных отличия:

- Описание (1) заставляет компилятор отвести память соответствующего размера под размещение всех элементов массива вещественных чисел (здесь конкретно под 5 элементов) и запомнить эту область памяти в константе a.

- Описание (2) отводит память только под сам указатель y, адрес, хранящийся в нём, не определен, никакой дополнительной памяти, например под массив, не отводится.

Необходимо понимать, что имя массива это тоже указатель, но указатель-константа. То есть он хранит адрес массива (что эквивалентно — адрес первого элемента массива), но в ходе работы программы этому указателю-константе нельзя присвоить новое значение. Другое дело указатель. Это переменная и ей, в ходе работы программы, можно присваивать значения и даже неоднократно. И более того эта переменная может участвовать в выражениях как логических, так и арифметических.

Инициализировать указатель можно, как и переменные других типов, используя оператор присваивания или при описании указателя. Так, же можно использовать и оператор ввода, но *не всякий* оператор ввода позволяет это, да и это должна быть очень специфичная программа, чтобы потребовалось вводить адрес из файла или с клавиатуры.
Пример.

```
double a[10], b[20], *pa1 = a, *pa2 = &a[0], *pb1 = b, *pb2 = &b[0];
double *pa3, *pa4, *pb3, *pb4;
pa3 = a; pa4 = &a[0]; pb3 = b; pb4 = &b[0];
```

Здесь и имя массива a и все указатели pa1, pa2, pa3, pa4 (после выполнения всех присвоений) ссылаются на одну и ту же область памяти. То же самое относится и к массиву b и указателям pb1, pb2, pb3, pb4. Поэтому обратиться к одному и тому же элементу массива, например к элементу с индексом 5 из массива a, можно по эквивалентным конструкциям: a[5] или pa1[5] или pa2[5] или pa3[5] или pa4[5]. Совершенно аналогично можно обратиться к любой другой ячейке массива a и массива b.

Динамическое выделение памяти

Эффективность применения указателей становится очевидна в сложных программах обрабатывающих переменное количество данных. В таких программах, как правило, на момент их создания неизвестно, сколько данных будет загружено из файла или клавиатуры. Да и количество данных меняется при каждом запуске программы. Точнее говоря, в таких программах количество данных становится известно только после старта программы. Поэтому необходимая длина массива, для хранения этих данных, также становится известна уже во время работы программы.

Типовым решением этой проблемы является использование подпрограмм динамического выделения памяти. Общий принцип работы этих подпрограмм следующий: программа вызывает такую подпрограмму, указывая необходимый объем памяти, подпрограмма запрашивает эту память у операционной системы, получает ее и возвращает программе адрес этой выделенной области памяти. Полученный адрес выделенной области памяти программа помещает в указатель. В дальнейшем программа через этот указатель и обращается к этой области памяти.

В языке C для выделения памяти можно использовать целое семейство функций: alloc, calloc, ..., и т.п., которые и выделяют память и возвращают в программу адрес этой памяти. В языке C++ для выделения памяти наиболее активно используют оператор new. Общий вид записи этого оператора следующий:

указатель = **new** *спецификация типа* **[***повторитель***]**;

Здесь *указатель* сохранит адрес выделенной памяти, оператор выделит память достаточную для хранения массива, каждая ячейка которого того типа, который указан в *спецификации типа*, а длина массива будет равна количеству, указанному в *повторителе*. При этом надо отметить, что повторитель может быть как *константой*, так и *переменной* и даже *выражением целого типа*. Этим и обеспечивается динамика и гибкость программы: по ходу работы программы узнаем/вычисляем размер массива и запрашиваем необходимый объем, как ресурс операционной системы. Если необходимо выделить память всего лишь под единственный элемент, например структуру, то повторитель и квадратные скобки могут не указываться. То есть, допустим формат оператора new и как:

указатель = **new** *спецификация типа*;

Динамически выделяемая память это ресурс операционной системы. В ходе работы программы каждый блок памяти, выделяемый программе, помечается системой как ‘занятый’. И даже после завершения работы программы эти блоки так и остаются ‘занятыми’. Поэтому программа должна обязательно не только захватывать память для своей корректной работы, но и *обязательно высвобождать* эту память. В языке С для этих целей применяют подпрограммы `freemem` и ей подобные. В языке С++, в дополнение к этим подпрограммам, также имеется специальный оператор `delete`. Общий вид записи этого оператора следующий:

`delete` *<указатель>*`;`

При освобождении памяти из под массива, на который ссылается указатель, лучше использовать оператор вида:

`delete[]` *<указатель>*`;`

При работе с динамически выделяемой памятью необходимо строго придерживаться следующего правила.

Правило. Перед использованием указатель должен быть инициализирован, например адресом имеющегося статического массива или указывать на динамический массив.
Правило. Если в ходе работы программа *выделяла* динамическую память, то перед завершением работы вся память должна быть *освобождена*.

Можно привести фрагменты типового использования в программе массива с динамическим выделением памяти.
Пример.

```
double *y;
int n, i;
…
n = выражение;
…
y = new double [n];
…
for (i = 0; i < n; i++)
  cin >> y [i];
…
delete[] y;
…
```

Здесь отражены основные действия с указателем и динамической памятью, на которую он ссылается: указатель описывается; определяется длина необходимого массива; выделяется память под размещение такого массива; используется этот массив в соответствии с алгоритмом программы; высвобождается память, выделенная под массив.

Операции над указателями

Указатели являются переменными базового типа языка С (и С++). Поэтому над указателями, как и над переменными других типов, можно выполнять арифметические действия. При этом необходимо помнить, что результаты этих действий несколько специфичны, так как они выполняются над указателями, хотя выражение выглядит как выражение, оперирующее с целочисленными данными.

К элементам массива можно обращаться и через индекс элемента и по смещению элемента от начала массива. То есть для массива `a[100]` на произвольный элемент, например с индексом 50, ссылаться можно и как `a[50]` и как `*(a+50)`. Вторая форма интерпретируется следующим образом: от начала массива, на который указывает константа-указатель `a`, сместиться на 50 ячеек, и выполнять действие над нею. Необходимо помнить, что ‘50’ здесь номер ячейки и в той и в другой форме записи.

Аналогично можно оперировать и с указателями-переменными. Так, если y указатель на тот же массив a, то к той же ячейке памяти (элементу с индексом ‘50’ массива a) можно обратиться и как `y[50]` и как `*(y+50)`.

Примечание. В языке C (и C++) обращаться к ячейкам массива через имя или указатель можно и таким экзотическим способом: *индекс*[*имя*] и *индекс*[*указатель*]. Так к ячейке с индексом 50 можно обратиться и как `50[a]` и как `50[y]`.

Естественно, что указатель может ссылаться не только на элемент массива с индексом ‘0’, но и любой другой. Так допустимы конструкции `y = &a[10]` и `z = &a[23]`, где y и z указатели на ячейку того же типа, что и элемент массива a.

Кроме различных вариантов обращения к ячейкам массива над указателями можно выполнять арифметические действия. Если указатели адресуют элементы одного и того же массива, то их *можно сравнивать*, *складывать*, *вычитать*. При вычитании указателей, результатом будет *количество элементов*, расположенных между уменьшаемым и вычитаемым объектами (на которые указывают указатели). Но при этом:

- *Нельзя* сравнивать указатели на *разные* массивы.
- *Нельзя* вычитать два указателя, относящихся к *разным* массивам, складывать, умножать, делить, сдвигать.

Это вполне понятное требование, так как от запуска к запуску программы, взаимное расположение массивов различно, и поэтому результаты будут различны и непредсказуемы. В то же время указатели, ссылающиеся на элементы одного массива, всегда расположены одинаково, по отношению друг к другу.

Среди арифметических операций отдельно следует рассмотреть операции автоматического инкрементирования и декрементирования. Так вполне допустимы конструкции:

`y++, y--, ++y, --y`

Необходимо помнить, что здесь указатель переходит на следующий или предыдущий элемент массива, а не сдвигается на один байт. То есть работу такого оператора можно описать формулой:

ptr++ → *содержимое* (*ptr*) + `sizeof` (*тип*).

В некоторых алгоритмах очень удобно установить указатель на нужную ячейку и пробегать по ячейкам, расположенным подряд, с помощью таких операций.

Следует быть осторожным при смешивании операций инкрементирования и декрементирования к указателям и к данным, на которые эти указатели указывают. Естественно, что инкремент и декремент можно применить и к указателю и к содержимому. И в таких случаях необходимо очень аккуратно записывать такую и команду и тщательно проверять запись, чтобы убедиться, а к чему же в действительности будет применена операция к указателю или к содержимому.

Пример.

```
double *y;
```

`*(y++), *y++;` — увеличивается значение указателя y. И по обновленному адресу осуществляется доступ к значению.

`*(++y), *++y;` — по адресу y осуществляется доступ к значению. Затем увеличивается значение указателя y.

`(*y)++` — используется значение y, а затем увеличивается значение по адресу y. *Скобки обязательны*! Так как приоритет операций одинаков и в этом случае операции выполняются справа налево (см. таблицу приоритетов операций и здесь выше).

`++*y, ++(*y)` — увеличивается значение по адресу y., а затем используется в выражении.

Приоритеты операций инкрементирования и декрементирования и порядок их применения к указателю и содержимому справа налево требуют внимательного разбора: “А каков будет результат?”.

Аналогичным образом можно обращаться и ячейкам статически описанной матрицы, то есть двумерного массива. Так пусть имеется матрица целочисленных значений описанная как:

```
int d[50][20];
```

Тогда опираясь на имя матрицы как на указатель-константу можно обращаться к различным элементам матрицы:

- `d` — указатель на нулевую строку матрицы (та что с индексом 0).
- `*(d+1)` — указатель на первую строку матрицы; он отстоит от начала `d` на 20 элементов.
- `*(d+2)` — указатель на вторую строку двумерного массива; он отстоит от начала `d` на 40 элементов.
- `*(d+4)+3` — адрес третьего элемента в четвертой строке матрицы; то есть эквивалентно определению адреса `&d[4][3]`.

Стандартное выражение `d[i][j]`, обращение к элементу матрицы, на языке C (и C++) интерпретируется как `*(*(d+i)+j)`, которое также может использоваться в программе.

Массивы указателей

На основе указателей, как и на базе других элементов, можно формировать массивы. В этом случае описание статического массива указателей будет выглядеть как:

тип имя`*[`*количество*`]`;

Пример.

```
double *arr[15];  // описание массива указателей на double
```

Массив указателей может формироваться и динамически. Тогда указатель на такой массив будет описываться как:

тип `**`*имя*;

Пример.

```
double **ptr;    // описание указателя на указатель или массив указателей типа
                 double
```

Здесь переменная `ptr` является указателем на указатель типа `double`. Выделить память под такой массив можно аналогично динамическому массиву базовых типов:

указатель = **new** *спецификация типа* `*` `[`*повторитель*`]`;

Пример.

```
double **ptr;  // описание указателя на указатель или массив указателей типа
               double
int i;
ptr = new double * [15];  // выделение памяти под массив указателей на double
for (i = 0; i<15; i++)
  ptr [i] = new double;  // выделение памяти под каждый double
…
for (i = 0; i<15; i++)
  delete[] ptr [i];     // освобождение памяти из-под каждого double
delete[] prt;     // освобождение памяти из-под массива
```

Здесь *указатель* сохранит адрес памяти выделенной под массив, а в операторе цикла каждый оператор `new` выделит память под индивидуальный элемент типа `double`.

Освобождение памяти производится в обратном порядке: высвобождается память из-под каждого элемента (пока элементы массива помнят, а где же это выделялось), а затем уже высвобождается память, отведенная под сам массив.

Обратиться к значениям, на которые ссылаются элементы такого массива можно по конструкции: `*ptr[i]`.

Пример.

```
for (i = 0; i<15; i++)
  cin >> *ptr [i];  // ввод значений типа double, в ячейки на которые ссылаются
                    // элементы массива ptr
```

Двумерные динамические массивы

Динамически можно выделять память не только под одномерные массивы, но и под двумерные. Одномерные динамические массивы размещаются в памяти аналогично статическим; разница лишь во времени создания — на стадии компилирования или во время выполнения программы. Для выделения памяти под двумерные массивы приходится применять вспомогательный массив указателей. Выделение памяти под двумерный массив может производиться по следующей схеме:

- Описать указатель на массив указателей заданного типа.
- Определиться с размерами требуемой матрицы, например, **N** строк и **M** столбцов.
- Выделить память под массив указателей. Длина массива равна количеству строк **N**.
- В цикле выделить память под каждую строку: динамический массив длиной **M** с элементами заданного типа.
- Заполнить элементы полученной матрицы и выполнять с ней необходимые действия.

Освобождение памяти выполняется в обратном порядке:

- В цикле освободить память из-под каждой строки динамического массива.
- Освободить память из-под массива указателей.

Шаблон программы использующей динамическую матрицу, может выглядеть следующим образом:
Пример.

```
const N = 15, M = 31;
double **matr;   // описание указателя на указатель или массив указателей типа
                 double
int i, j;
matr = new double * [N];  // выделение памяти под массив указателей на double
for (i = 0; i<N; i++)
  matr [i] = new double [M];  // выделение памяти под массив из double
…
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    cin >> matr [i][j];
…
for (i = 0; i<N; i++)
  delete[] matr [i];// освобождение памяти из-под каждого double
delete[] matr; // освобождение памяти из-под массива
```

Обратите внимание, что, несмотря на такую громоздкую схему выделения памяти под матрицу, оперирование с элементами матрицы осуществляется по такому же синтаксису, что и для статической матрицы. Стандартное выражение `d[i][j]`, обращение к элементу матрицы, можно так же заменять на эквивалентную конструкцию допустимую для статически описанных матриц: `*(*(d+i)+j)`.

Типизированные указатели. Пустой указатель

Указатели, как правило, ссылаются на переменные определенного типа. Такие указатели называются *типизированными*. При реализации выражений с участием таких указателей компилятор учитывает длину переменной такого типа, дабы корректно изменять значение указателя, например в операциях инкрементирования и декрементирования. То есть на работу с указателями накладывается одно ограничение: *символьный* указатель указывает только на *символьные данные*, *вещественный — на вещественные* и так далее.

Помимо этого существует нетипизированный указатель. Его описание осуществляется ключевым словом `void`: `void *ptr`. Он может ссылаться на любую ячейку, любого типа. То есть ему можно присваивать значения любого типизированного указателя, например:

Пример.

```
void *vo = NULL;

vo = ptr;
vo = matr;
vo = matr [0];
```

Но вот переслать адрес из нетипизированного указателя в типизированный, можно только с использованием операции переопределение типа.

Типизированному и нетипизированному указателю может быть присвоено пустое значение NULL. Типовое применение такого значения, следующее: при описании указатель инициализируется значением NULL; в ходе работы программы можно проверять указатель; если он не равен NULL, то указатель ссылается на какую-то память. Такая проверка очень активно используется в программах работающих с указателями. В частности, если функциям выделения памяти не удается выделить её, то они возвращают NULL, как признак фиаско.

Приложение 1