КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 06.2 Объявления. Декларатоpы и инициализация.

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

Оглавление

Объявления....3
Декларатоpы....3
Декларатоpы указателей....6
Декларатоpы массивов....7
Объявление функций (включая прототипы)....11
Имена типов....18
Инициализация....19
Инициализация агрегатов....22
Позиционная инициализация....29

Объявления

Синтаксис объявления:

спецификаторы-объявления ***[список-деклараторов[1]-с-инициализацией]*** **;**

Декларатор

Каждый декларатор объявляет один идентификатор и утверждает, что когда в выражении появляется операнд той же формы, что и декларатор, он обозначает функцию или объект с областью действия, продолжительностью хранения и типом, указанными спецификаторами объявления.

Декларатор в скобках идентичен декларатору, не заключенному в скобки, но скобками может быть изменена привязка сложных деклараторов.

```
int a, b, c;
int arr[sz], *p, foo(), (*bar)(int);
```

Цитата из предыдущей темы:

Спецификаторы объявления состоят из последовательности спецификаторов, которые указывают:

- *тип связывания;*
- *продолжительность хранения;*
- ***часть типа** сущностей (объекты, функции), на которые эти декларатоpы указывают.*

1 декларатор¸описатель, объявитель — содержит идентификаторы, подлежащие объявлению

Синтаксис

декларатор:

[указатель] прямой-декларатор

прямой-декларатор:

идентификатор

(*декларатор* **)**

прямой-декларатор **[** *[список-квалификаторов-типа] [выражение-присваивания]* **]**

прямой-декларатор **[** **static** *[список-квалификаторов-типа] выражение-присваивания* **]**

прямой-декларатор **[** *список-квалификаторов-типа* **static** *выражение-присваивания* **]**

прямой-декларатор **[** *[список-квалификаторов-типа]* ***** **]**

прямой-декларатор **(** *список-типов-параметров* **)**

прямой-декларатор **(** *[список-идентификаторов]* **)**

указатель:

***** *[список-квалификаторов-типа]*

***** *[список-квалификаторов-типа] указатель*

список-квалификаторов-типа:

квалификатор-типа

список-квалификаторов-типа квалификатор-типа

```
int * const cp, * const volatile cvp; // квалификация указателя допустима
int a, const b;  // квалификация фунд. типа в деклараторе недопустима
int const c;     // квалификатор const относится к спецификатору типа
int (const c);   // квалификатор const относится к декларатору фундамен. типа
```

список-типов-параметров:

список-параметров

список-параметров , ...

список-параметров:

объявление-параметров

список-параметров , *объявление-параметров*

объявление-параметров:

спецификаторы-объявления декларатор

спецификаторы-объявления [абстрактный-декларатор]

список-идентификаторов:

идентификатор

список-идентификаторов , *идентификатор*

Декларатоpы указателей

`*` **D**
`*` *список-квалификаторов-типа*[2] **D**

Пример

Следующая пара объявлений демонстрирует разницу между «переменным указателем на константное значение» и «константным указателем на значение переменной».

```
const int    *ptr_to_constant;
int * const   constant_ptr;
```

Содержимое любого объекта, на которое указывает **ptr_to_constant**, не должно изменяться через этот указатель, но сам **ptr_to_constant** может быть изменен, чтобы указывать на другой объект.

Точно так же содержимое **int**, на которое указывает **constant_ptr**, может быть изменено, но сам **constant_ptr** всегда должен указывать на одно и то же местоположение.

Объявление константного указателя **constant_ptr** может быть сделано более понятным с помощью включения определения для типа «указатель на **int**».

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

объявляют **constant_ptr** как объект, имеющий тип «**const**-квалифицированный указатель на **int**».

---

2 const, restrict, volatile, _Atomic

Декларторы массивов

```
1    D[ ]
2    D[ список-квалификаторов-типа[3] ]
3    D[ выражение ]
4    D[ список-квалификаторов-типа выражение ]
5    D[ static выражение ]
6    D[ static список-квалификаторов-типа выражение ]
7    D[ список-квалификаторов-типа static выражение ]
8    D[ * ]
8    D[ список-квалификаторов-типа * ]
```

Если размер отсутствует, тип массива является неполным типом.

Если размер равен *, а не является выражением, тип массива является типом *массива переменной длины* неопределенного размера, который может использоваться только в объявлениях или именах типов с областью прототипа функции.

Таким образом, * может использоваться только в объявлениях функций, которые не являются определениями. Такие массивы, тем не менее, являются полными типами.

3 const, volatile, _Atomic

Массивы переменной длины (VLA)

Если размер массива не является целочисленным константным выражением или тип элемента массива не имеет известного константного размера, такой тип массива является типом *массива переменной длины*.

Размер массива может не являться целочисленным константным выражением. В этом случае:

- если оно встречается в объявлении в области действия прототипа функции, оно обрабатывается так, как если бы оно было заменено на *;
- в противном случае, каждый раз, когда оно вычисляется, его значение должно быть больше нуля.

Размер каждого экземпляра типа массива переменной длины
не изменяется в течение срока его жизни.

Примеры

```
float fa[11], *afp[17];
```

объявляет массив чисел с плавающей точкой и массив указателей на числа с плав. точкой.

Следует обратить внимание на различие между декларациями

```
extern int *x;
extern int y[];
```

Первая объявляет **x** как указатель на **int**, вторая объявляет **y** как массив типа **int** неопределенного размера (неполного типа), память для которого выделена в другом месте.

Пример

Все объявления с переменными изменяемыми типами (VM-типами) должны находиться либо в области блока, либо в области прототипа функции.

Объекты массива, объявленные со спецификатором класса памяти **`_Thread_local`**, **`static`** или **`extern`**, не могут иметь тип массива переменной длины (VLA).

Однако объект, объявленный с помощью спецификатора класса памяти **`static`**, может иметь VM-тип (то есть указатель на тип VLA).

Все идентификаторы, объявленные с типом VM, должны быть обычными идентификаторами и, следовательно, не могут быть членами структур или объединений.

```
extern int n;          // переменный размер
int A[n];              // недопустимо: VLA в области видимости файла
extern int (*p2)[n]; // недопустимо: VM-тип в области видимости файла
int B[100];            // допустимо: в области видимости файла, но не VM-тип
void fvla(int m, int C[m][m]); // допустимо: VLA в области видимости  прототипа
...
void fvla(int m, int C[m][m]) { // допустимо: adjusted to auto pointer to VLA

    typedef int VLA[m][m];      // допустимо: block scope typedef VLA

    struct tag {
        int (*y)[n];     // недопустимо: y не обычный идентификатор
        int z[n];        // недопустимо: z не обычный идентификатор
    };

    int D[m];            // допустимо: auto VLA
    static int E[m];     // недопустимо: static block scope VLA
    extern int F[m];     // недопустимо: F имеет внешн. связывание и является VLA
    int (*s)[m];         // допустимо: auto pointer to VLA
    extern int (*r)[m];  // недопустимо: r имеет внешн. связывание и указ. на VLA
    static int (*q)[m] = &B; // допустимо: q указывает на VLA в статическом блоке
}
```

Объявление функций (включая прототипы)

```
D(void) -- у функции нет параметров
D( )    -- информация о количестве или типах параметров не предоставляется
D( список-типов-параметров )
D( список-идентификаторов )
```

Ограничения

При объявлении функции не должен указываться тип возврата, который является типом функции или типом массива.

Иными словами, функция не может возвращать функции[4] и массивы.

Единственный спецификатор класса памяти, который может присутствовать в объявлении параметра, это **`register`**.

4 Функция C, строго говоря, не является функцией первого класса (высшего порядка).

Семантика

Список типов параметров определяет типы и может объявлять идентификаторы для параметров функции.

Особый случай безымянного параметра типа **`void`** как единственного элемента в списке указывает, что у функции нет параметров.

Объявление параметра как «массива типа» корректируется на «квалифицированный указатель на тип», где квалификаторы типа (если они есть) — это те, которые указаны внутри **`[ ]`** при выводе типа массива.

Если внутри **`[ ]`** при выводе типа массива присутствует ключевое слово **`static`**, то при каждом вызове функции значение соответствующего *фактического аргумента* должно обеспечивать доступ к первому элементу массива с как минимум таким количеством элементов, как указано в выражении размера.

Объявление параметра в качестве «функции, возвращающей *тип*» корректируется на «указатель на функцию, возвращающую *тип*».

Если список заканчивается многоточием **`( , ...)`**, информация о количестве или типах параметров после запятой не указывается. Для доступа к аргументам, которые соответствуют многоточию, могут использоваться макросы, определенные в заголовке **`<stdarg.h>`**.

Если декларатор функции не является частью определения этой функции, параметры могут иметь неполный тип и могут использовать нотацию **`[*]`** в своих последовательностях спецификаторов деклараторов для указания типов массива переменной длины.

Спецификатор класса памяти в спецификаторах объявления параметра, если он присутствует, игнорируется, если только он не является одним из членов списка типов параметров В определении функции.

Список идентификаторов объявляет только идентификаторы параметров функции.

Пустой список в декларатоpе функции, который является частью определения этой функции, указывает на то, что функция не имеет параметров.

Пустой список в декларатоpе функции, который не является частью определения этой функции, указывает на то, что информация о количестве или типах параметров не предоставляется.

Пример 1
Объявление

```
int f(void), *fip(), (*pfi)();
```

объявляет:

- функцию **f** без параметров, возвращающую **int**;
- функцию **fip** без спецификации параметров, возвращающую указатель на **int;**
- указатель **pfi** на функцию без спецификации параметра, возвращающую **int**.

Особенно полезно сравнить последние два.

Связка ***fip()** — это ***(fip())**, так что данная конструкция предполагает вызов функции **fip** и последующее использование косвенного обращения к значению указателя для получения **int**.

В декларатоpе **(*pfi)()** дополнительные скобки необходимы, чтобы указать, что косвенное обращение к **pfi** дает указатель на функцию, который затем используется для вызова функции; функция же возвращает **int**.

Пример 2
Объявление

```
int (*apfi[3])(int *x, int *y);
```

объявляет массив **apfi** из трех указателей на функции, возвращающие **int**. Каждая из этих функций имеет два параметра, которые являются указателями на **int**.

Идентификаторы **x** и **y** объявляются только для наглядности и выходят из области видимости прототипа в конце объявления **apfi**.

Пример 3
Объявление

```
int (*fpfi(int (*)(long), int))(int, ...);
```

объявляет функцию **fpfi**, которая возвращает указатель на функцию, возвращающую **int**.

Функция **fpfi** имеет два параметра

- первый — **int (*)(long)**, указатель на функцию, возвращающую **int**, с одним параметром типа **long int**;

- второй — **int**.

Указатель, возвращаемый **fpfi**, указывает на функцию, которая имеет один параметр **int** и принимает ноль или более дополнительных аргументов любого типа.

Пример 4

Следующий прототип имеет параметр изменяемого типа.

```
void addscalar(int n, int m, double a[n][n*m+300], double x);

int main( ) {
    double b[4][308];
    addscalar(4, 2, b, 2.17);
    return 0;
}

void addscalar(int n, int m, double a[n][n*m+300], double x) {

    for (int i = 0; i < n; i++) {
        for (int j = 0, k = n*m+300; j < k; j++) {
            a[i][j] += x; // a -- указатель на VLA с n*m+300 элементами
        }
    }
}
```

Пример 5

Ниже приведены четверки совместимых объявлений прототипов функций.

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

```
void f(double (* restrict a)[5]);
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
void f(double a[restrict static 3][5]);
```

Следует обратить внимание, что в последнем объявлении также указывается, что аргумент, соответствующий **a** в любом вызове **f**, должен быть ненулевым указателем на первый из как минимум трех массивов по 5 **double**, чего нет у других.

Имена типов

Иногда необходимо указывать тип. Это достигается с помощью *имени типа*, которое синтаксически является декларатором типа для функции или объекта, но в этом декларатор отсутствует идентификатор.

Пустые скобки в имени типа интерпретируются как «функция без указания параметра», а не как избыточные скобки вокруг пропущенного идентификатора.

Пример

Нижеуказанные конструкции представляют типы без указания идентификаторов

(a) **`int`** **`int`**

(b) **`int *`** указатель на **`int`**

(c) **`int *[3]`** массив из трех указателей на **`int`**

(d) **`int (*)[3]`** указатель на массив из трех **`int`**

(e) **`int (*)[*]`** указатель на VLA с неопределенным числом **`int`**

(f) **`int *()`** ф-ция без указания параметров, возвращающая указатель на **`int`**

(g) **`int (*)(void)`** указатель на функцию без параметров, возвращающих **`int`**

(h) **`int (* const [])(unsigned int, ...)`** массив из неопределенного числа постоянных указателей на функции, каждая с одним параметром, который имеет тип **`unsigned int`** и неопределенное число других параметров, возвращающая **`int`**.

Инициализация

Синтаксис

инициализатор:

выражение-присваивания

{ *список- инициализаторов* **}**

{ *список- инициализаторов* **,** **}**

список-инициализаторов:

[позиционный-инициализатор] *инициализатор*

список-позиционных-инициализаторов **,** *инициализатор*

список-позиционных-инициализаторов **,** *позиционный-инициализатор* *инициализатор*

позиционная-инициализация:

список-позиционных-инициализаторов **=**

список-позиционных-инициализаторов:

позиционный-инициализатор

список-позиционных-инициализаторов *позиционный-инициализатор*

позиционный-инициализатор:

[*константное-выражение* **]**

. *идентификатор*

Семантика

Инициализатор определяет начальное значение, сохраняемое в объекте.

Безымянные члены объектов структуры и типа объединения не участвуют в инициализации.

Если объект, имеющий автоматическую продолжительность жизни и не инициализируется явно, его значение является неопределенным.

Правила

Если объект со статической или потоковой длительностью хранения не инициализирован явно, то:

- если его тип является типом указателя, он инициализируется нулевым указателем;
- если его тип является арифметическим типом, он инициализируется положительным или беззнаковым нулем;
- если это агрегат, каждый элемент инициализируется (рекурсивно) в соответствии с вышеприведенными правилами, при этом любое заполнение (padding) инициализируется нулевыми битами;
- если это объединение, первый именованный элемент инициализируется (рекурсивно) в соответствии с этими правилами, и любое заполнение инициализируется нулевыми битами;

Инициализатор для скаляра должен быть одним выражением, опционально (необязательно) заключенным в фигурные скобки.

К начальному значению объекта (это значение выражения после преобразования) применяются те же ограничения и преобразования типов, что и при простом присваивании, считая тип скаляра за неквалифицированную версию объявленного типа.

При условии, что был **#include**'d **<complex.h>**, объявления

```
int i = 3.5;
double complex c = 5 + 3 * I;
```

определяют и инициализируют **i** значением 3, а **c** значением 5. 0 + i3. 0.

Инициализация агрегатов

Массив символьного типа может быть инициализирован символьным строковым литералом или строковым литералом UTF-8, необязательно заключенным в фигурные скобки.

Элементы массива инициализируются последовательностью байт строкового литерала (включая завершающий нулевой символ, если есть место или если массив имеет неизвестный размер).

Следующее объявление

```
char s[] = "abc", t[3] = "abc";
```

определяет «простые» объекты с типом символьного массива **s** и **t**, элементы которых инициализируются символьными строковыми литералами.

Эта декларация идентична следующей

```
char s[] = { 'a', 'b', 'c', '\0' }, t[] = { 'a', 'b', 'c' };
```

Содержимое этих массивов может быть изменено.

С другой стороны, декларация

```
char *p = "abc"; //
```

определяет **p** с типом «указатель на символ» и инициализирует его, чтобы он указывал на объект с типом «массив символов» длиной 4, элементы которого инициализируются символьным строковым литералом.

При попытке использовать указатель p для изменения содержимого этого массива, поведение не определено.

Массив с типом элемента, совместимый с квалифицированной или неквалифицированной версией **wchar_t**, **char16_t** или **char32_t**, может быть инициализирован широким строковым литералом с соответствующим префиксом кодирования (**L**, **u** или **U** соответственно), необязательно заключенным в фигурные скобки.

Элементы массива инициализируются следующими один за другим широкими символами широкого строкового литерала, включая завершающий нулевой широкий символ, если есть место.

Инициализатор для объекта, имеющего агрегатный тип или тип объединения, должен быть заключенным в скобки списком инициализаторов для элементов или именованных членов этого агрегатного типа.

Подобъекты текущего объекта (в случае отсутствия позиционной инициализации) инициализируются в порядке, соответствующем типу инициализируемого объекта — элементы массива в порядке возрастания индекса, элементы структуры в порядке объявления и с первого поименованного члена объединения.

Примеры

Следующее объявление

```
int x[] = { 1, 3, 5 };
```

определяет и инициализирует **x** как одномерный массив объектов, который имеет три элемента — размер массива не указан, однако присутствует три инициализатора.

Следующее объявление

```
int y[4][3] =  { { 1, 3, 5 }, { 2, 4, 6 }, { 3, 5, 7 }, };
```

является определением с инициализацией, полностью заключенной в скобки:
1, 3, и 5 инициализируют первую строку **y** (объект массива **y[0]**), а именно **y[0][0]**, **y[0][1]** и **y[0][2]**. Аналогично, следующие две строки инициализируют **y[1]** и **y[2]**.

Инициализатор заканчивается рано, поэтому **y[3]** инициализируется нулями.

Точно такой же эффект мог быть достигнут с помощью

```
int y[4][3] = {
      1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

Инициализатор для **y[0]** не начинается с левой фигурной скобки, поэтому для нее используются три элемента из списка.

Аналогично, следующие тройки последовательно используются для **y[1]** и **y[2]**.

Пример

Объявление

```
int z[4][3] = { { 1 }, { 2 }, { 3 }, { 4 } };
```

инициализирует первый столбец **z**, как указано, и инициализирует остальные нулями.

Пример

Объявление

```
struct {
    int a[3];
    int b;
} w[] = {{ 1 }, 2 };
```

является определением с инициализацией, непоследовательно указанной в скобках.

Объявление определяет массив с двумя структурами элементов — **w[0].a[0]** равен 1 и **w[1].a[0]** равен 2; все остальные элементы равны нулю.

Пример неполной, но последовательно заключенной в скобки инициализации

Объявление

```
short q[4][3][2] = {
    { 1 },
    { 2, 3 },
    { 4, 5, 6 }
};
```

Объявление определяет объект трехмерного массива: **q[0][0][0]** равен 1, **q[1][0][0]** равен 2, **q[1][0][1]** равен 3, а значения 4, 5, И 6 инициализируют **q[2][0][0]**, **q[2][0][1]**, and **q[2][1][0]**, соответственно. Все остальные элементы инициализируются нулями.

Инициализатор для **q[0][0]** не начинается с левой фигурной скобки, поэтому можно использовать до шести элементов из текущего списка. Но в списке представлен только один, поэтому значения для остальных пяти элементов инициализируются нулями.

Аналогично, инициализаторы для **q[1][0]** and **q[2][0]** не начинаются с левой фигурной скобки, поэтому каждый использует до шести элементов, инициализируя свои соответствующие двумерные субагрегаты.

Если бы в любом из списков было больше шести пунктов, было бы выдано диагностическое сообщение.

Тот же самый результат инициализации мог быть достигнут с помощью:

```
short q[4][3][2] = {
    1, 0, 0, 0, 0, 0,
    2, 3, 0, 0, 0, 0,
    4, 5, 6
};
```

или с помощью:

```
short q[4][3][2] = {
    {{ 1 },},
    {{ 2, 3 }, },
    {{ 4, 5 }, { 6 },}
};
```

в полной скобочной форме.

Следует отметить, что полностью инициализированные и заключенные в квадратные скобки формы инициализации, как правило, с меньшей вероятностью вызывают путаницу.

Позиционная инициализация

Позиционная инициализация используется для инициализации конкретных подобъектов агрегатного типа.

Инициализация затем продолжается далее по порядку, начиная со следующего подобъекта после того, который описывается позиционным инициализатором.

```
int a[10] = {1, [3] = 5, 7, 9};
```

После инициализации члена объединения (**union**) следующим подобъектом является объект, содержащий объединение.

Каждый список позиционных инициализаторов начинает свое описание с объекта, связанного с ближайшей парой фигурных скобок.

Таким образом, позиционный инициализатор может указывать только точный подобъект агрегата или объединения, который связан с окружающей парой скобок.

Если агрегат или объединение содержит элементы или члены, которые являются агрегатами или объединениями, эти правила инициализации рекурсивно применяются к субагрегатам или содержащимся объединениям.

Если в списке, заключенном в фигурные скобки, меньше инициализаторов, чем элементов или членов агрегата или меньше символов в строковом литерале, используемом для инициализации массива известного размера, чем элементов в массиве, оставшаяся часть агрегата инициализируется неявно так же, как объекты, которые имеют статическую длительность хранения.

Если инициализируется массив неизвестного размера, его размер определяется по наибольшему индексируемому элементу с явным инициализатором.

Тип массива завершается в конце его списка инициализатора.

Если *позиционный инициализатор* имеет форму

[*константное-выражение* **]**

то текущий объект (определенный ниже) должен иметь тип массива, а выражение должно быть выражением целочисленной константы.

Если массив имеет неизвестный размер, допустимо любое неотрицательное значение.

Если позиционный инициализатор имеет форму

. *идентификатор*

тогда текущий объект (определенный ниже) должен иметь структуру или тип объединения, а идентификатор должен быть именем члена этого типа.

Пример

Элементы структуры могут быть инициализированы ненулевыми значениями независимо от их порядка:

```
div_t answer = { .quot = 2, .rem = -1 };
```

Пример

Позиционные инициализаторы могут использоваться для обеспечения явной инициализации, когда неряшливо оформленные списки инициализаторов могут быть неправильно поняты:

```
struct {
    int a[3], b;
} w[] = { [0].a = {1}, [1].a[0] = 2 };
```

Пример

С помощью одного позиционного инициализатора можно «выделить» пространство с обоих концов массива:

```
int a[MAX] = {
    1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

В приведенном выше примере, если **MAX** больше десяти, в середине будут некоторые нулевые элементы; если оно меньше десяти, некоторые из значений, предоставленных первыми пятью инициализаторами, будут переопределены вторыми пятью.

Пример

Может быть инициализирован любой член объединения:

```
union {
    /* ... */
} u = { .any_member = 42 };
```