КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 05. Операторы и блоки в С

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

2028.10.06

Оглавление

Взаимодействие и правила поведения на занятиях

Язык общения — русский
Фото- и видеосъемка запрещается, болтовня и прочее мычание тоже
Использование мобильных гаджетов может вызвать проблемы

prep@lsi.bas-net.by
ftp://student@lsi.bas-net.by/

Старосты групп отправляют на **prep@** со своего личного ящика сообщение, в котором указывают свой телефон и ящик, к которому имеют доступ все студенты группы. В этом же сообщении в виде вложения приводят списки своих групп (**.odt** или **.doc**). Если у группы нет ящика, его следует создать. Все студенты группы должны мониторить этот ящик.

Формат темы этого сообщения: «**010901 Фамилия И.О. Список группы**»
Формат темы для **prep@** вообще: «**010901 Фамилия И.О. Суть сообщения**»

Правила составления сообщений
- текстовый формат сообщений;

Удаляется на сервере присланное в ящик prep@ все, что:
- без темы;
- имеет тему не в формате;
- содержит html, xml и прочий мусор;
- содержит рекламу, в том числе и сигнатуры web-mail серверов;
- содержит ссылки на облака и прочие гуглопомойки вместо прямых вложений.

Операторы и блоки

Синтаксис

оператор:

помеченный-оператор
составной-оператор
оператор-выражение
оператор-выбора
итерационный-оператор
оператор-безусловного-перехода

Семантика

Выражение — это последовательность операторов и операндов, которая задает вычисление значения, обозначает объект, обозначает функцию, генерирует побочные эффекты и/или выполняет комбинацию вышеперечисленного.

***Оператор* определяет действие, которое должно быть выполнено.**

За исключением явно указанного, операторы выполняются последовательно.

Блок позволяет группировать объявления и операторы в одну синтаксическую единицу.

Оператор метки

Синтаксис

помеченный-оператор:

1 *идентификатор* **:** *оператор*

2 **case** *константное-выражение* **:** *оператор*

3 **default** **:** *оператор*

Ограничения

2, 3 Метки **case** или **default** должна появляться только в операторе **switch**.

Дальнейшие ограничения на такие метки обсуждаются в инструкции **switch**.

Имена меток должны быть уникальными внутри функции.

Семантика

1 Любому оператору может предшествовать префикс, который объявляет идентификатор в качестве имени метки.

Сами по себе метки не изменяют поток управления, который беспрепятственно проходит через них.

Составной оператор

Синтаксис

составной-оператор:

    { }
    { *список-членов-блока* }

список-членов-блока:

    член-блока
    список-членов-блока *член-блока*

член-блока:

    объявление
    оператор

Семантика

Составной оператор является блоком.

Оператор-выражение и пустой оператор

Синтаксис

оператор-выражение:

```
1       выражение ;
2       ;
```

Семантика

Выражение в *оператор-выражение* вычисляется как пустое выражение ради его побочных эффектов, такие как присваивание и вызовы функций, которые имеют сторонние эффекты.

Пустой оператор (состоящий только из точки с запятой) не выполняет никаких операций.

Пример 1

Если вызов функции, возвращающей значение, вычисляется только ради его сторонних эффектов как оператор-выражение, возвращаемое значение можно аннулировать явным образом с помощью преобразования выражения в **`void`**-выражение, используя приведение типа:

```
int p(int);
/* ... */
(void)p(0);
```

Пример 2

Во фрагменте программы

```
char *s;
/* ... */
while (*s++ != '\0') // пропуск сиволов до конца строки
    ;
```

пустой оператор `;` используется для организации пустого тела цикла в итерационном операторе **`while`**.

Пример 3

Пустой оператор также может использоваться в качестве носителя метки непосредственно перед закрывающей **}** составного оператора.

```
while (loop1) {
    /* ... */
    while (loop2) {
        /* ... */
        if (want_out) {
            goto end_loop1;
        }
        /* ... */
    }
    /* ... */
    end_loop1: ;
}
```

Операторы выбора

Синтаксис

оператор-выбора:

```
1    if ( выражение ) оператор
2    if ( выражение ) оператор else оператор
3    switch ( выражение ) оператор
```

Семантика

Оператор выбора делает выбор из набора операторов в зависимости от значения управляющего выражения.

Оператор выбора — это блок, область действия которого является строгим подмножеством области действия блока, который его включает.

Каждый связанный с ним оператор также является блоком, область действия которого является строгим подмножеством области действия оператора выбора.

Оператор if

Синтаксис

```
1    if ( выражение ) оператор
2    if ( выражение ) оператор else оператор
```

Ограничения

Управляющее выражение оператора **`if`** должно иметь скалярный тип[1].

Семантика

В обеих формах первый субоператор выполняется, если значение выражения неравно 0.

В форме **`else`** второй субоператор выполняется, если значение выражения равно 0.

Если первый субоператор достигается через метку, второй субоператор не выполняется.

`else` связывается с лексически ближайшим предшествующим **`if`**, который допускается синтаксисом.

---

1 **Помним!!!** Скалярными типами называются арифметические типы и типы указателей. Арифметические типы — целочисленные и типы с плавающей точкой. Целочисленные типы — тип char, целочисленные типы со знаком и без знака, а также перечисляемые типы. Типы с плавающей точкой — действительные и комплексные типы с плавающей точкой.

Пример 1

```
int *array = malloc(array_sz * sizeof array); // или  sizeof int
if (NULL == array) {
    perror("array: ");
    exit(errno);
}
```

Он же, только немного по другому

```
int *array;
if (NULL == (array = malloc(array_sz * sizeof array))) {
    perror("array: ");
    exit(errno);              // памяти нет, значит, досвидания
}
```

Пример 2. Множественный уточняющий неоднородный выбор (лестница)

```
if ( color == red || color == green)
    solarize(color);
else if (form == triangle)
    paint(object, triangle);
else if (mass < mass_threshold)
    wipe(object);
else if (mass = mass_threshold)
    mark_limit( );
else                                  //  mass > mass_threshold
    zinc_to_boss( );
```

Почему лестница?

```
if ( color == red || color == green)
    solarize(color);
else if (form == triangle)
         paint(object, triangle);
     else if (mass < mass_threshold)
              wipe(object);
          else if (mass = mass_threshold)
                   mark_limit( );
               else                   //  mass > mass_threshold
                   zinc_to_boss( );
```

Он же с расставленными скобками **{ }** составного оператора

```
if ( color == red || color == green) {
    color = solarize(color);
    paint(object, color);
} else if (form == triangle) {
    paint(object, black
} else if (mass < mass_threshold) {
    wipe(object);
} else if (mass = mass_threshold) {
    mark_limit();
} else {                            //  mass > mass_threshold
    zinc_to_boss();
}
```

Оператор `switch`

Синтаксис

3 **switch** **(** *выражение* **)** *оператор*

Метки оператора **switch**

6.8.1-2 **case** *константное-выражение* **:** *оператор*

6.8.1-3 **default** **:** *оператор*

Оператор **switch** вызывает переход управления к *оператору*, являющемуся телом оператора **switch**, или пропуск этого *оператора* в зависимости от значения управляющего *выражения*, а также от наличия метки **default** и значений любых меток **case** в теле **switch**.

Управляющее *выражение* оператора **switch** должно иметь целый тип.

В операторе **switch** может быть не более одной метки **default**.

Метки **case** и **default** доступны только в пределах ближайшего окружающего оператора **switch**.

```
switch (expr) {
case c1:
   operator_c1
case c2:
   operator_c2
default:
   operator_default
}
```

Как работает switch

```
switch (expr) {
case const_expr_1:
   operator_1
case const_expr_2:
   operator_2
default:
   operator_default
}
```

1. Значение управляющего выражения **expr** целочисленно повышается.

2. Значение константного выражения в каждой метке **case** преобразуется к повышенному типу управляющего выражения. Здесь можно использовать как **char**, так и **wchar**:

case 'c', L'c': — нормально, символ, будь он однобайтным или широким, всегда приводится к целочисленной константе.

case "c", L"c": — ошибка: значение case-метки «строка» неприводимо к целочисленной константе.

3. Если преобразованное значение метки совпадает со значением управляющего выражения, управление переходит к оператору, расположенному после метки соответствующего **case**.

4. В противном случае, если есть метка **default**, управление переходит к помеченному ей выражению.

5. Если преобразованное константное выражение ни с чем не совпадает и при этом отсутствует метка **default**, никакая часть тела оператора **switch** не выполняется.

Любой вложенный оператор **switch** может иметь метку **default** или константное выражения **case** со значениями, которые дублируют константное выражения **case** во вкладывающем операторе **switch**.

Пример

```
switch (expr) {
    int i = 4;              // -Wswitch-unreachable
    f(i);
case 0:
    i = 17;                 // -Wimplicit-fallthrough=
    /* проваливаемся под метку default */
default:
    printf("%d\n", i);
}
```

Объект с идентификатором **i** существует в пределах автоматической продолжительности хранения (в пределах блока), но никогда не проинициализируется. Поэтому, если управляющее выражение **expr** имеет ненулевое значение, вызов функции **printf** будет иметь доступ к неопределенному значению. По этой же причине не может быть достигнут и вызов функции **f**.

Итерационные операторы

Синтаксис

итерационный-оператор:

while **(** *выражение* **)** *оператор*

do *оператор* **while** **(** *выражение* **)** **;**

for **(** *[выражение_1]* **;** *[выражение]* **;** *[выражение_3]* **)** *оператор*

for **(** *объявление [выражение]* **;** *[выражение]* **)** *оператор*

Ограничения

Управляющее выражение итерационного оператора должно иметь скалярный тип.

Объявляющая часть оператора **for** должна объявлять идентификаторы только для объектов, имеющих класс памяти **auto** или **register**.

Итерационный оператор приводит к многократному выполнению оператора «*оператор*», называемого телом цикла, до тех пор, пока управляющее выражение не станет равно 0.

Повторение происходит независимо от того, был ли вход в тело цикла осуществлен через итерационный оператор или с помощью безусловного перехода.

Перепрыгнутый код не выполняется. В частности, управляющее выражение оператора **for** или **while** перед входом в тело цикла не вычисляется, не вычисляется и *выражение-1* оператора **for**.

Итерационный оператор — это блок, область действия которого является строгим подмножеством области действия включающего его блока. Тело цикла также является блоком, область действия которого является строгим подмножеством области действия оператора итерации.

Итерационный оператор, управляющее выражение которого не является константным выражением, который не выполняет никаких операций ввода-вывода, не обращается к volatile-объектам и не выполняет никаких синхронизирующих или атомарных операций в своем теле и в управляющем выражении или (в случае оператора **`for`**) в его *выражении-3*, может быть проигнорирован компилятором (его выполнение сразу завершается).

Оператор while

while (*выражение*)
  оператор-тело-цикла

Вычисление управляющего выражения происходит всякий раз перед очередным выполнением тела цикла. Таким образом, тело цикла может и не выполниться, если управляющее выражение перед входом в оператор **while** будет равно нулю.

```
int i = 5;
while (i--) {
    do-something
}
```

Или так, что эквивалентно

```
int i = 5;
while (i) {
    do-something
    i--;
}
```

Оператор do

do

оператор-тело-цикла

while (*выражение*) ;

Вычисление управляющего выражения происходит после каждого выполнения тела цикла. Таким образом, тело цикла выполнится по крайней мере один раз.

```
int i = 5;
do
    do-something
while (i--)

i = 5;
do {
    horh(note, duration);
} while (i--)
```

Оператор for

Оператор

for (*пункт-1* ; *выражение-2* ; *выражение-3*) *оператор*

Оператор **for** ведет себя следующим образом:

Выражение *выражение-2* является управляющим выражением, которое вычисляется перед каждым выполнением тела цикла.

Выражение *выражение-3* вычисляется как `void`-выражение (не имеющее значения, и только производящее сторонний эффект), после каждого выполнения тела цикла.

Синтаксический элемент *пункт-1* может являться объявлением или выражением.

Если *пункт-1* является объявлением, область видимости/действия любых идентификаторов, которые оно объявляет, представляет оставшуюся часть объявления, два других выражения и весь цикл. Такой *пункт-1* обрабатывается до первого вычисления управляющего выражения.

Если *пункт-1* является выражением, оно вычисляется как `void`-выражение перед первым вычислением управляющего выражения.

Таким образом, *пункт-1* определяет инициализацию для цикла, возможно, объявляя одну или несколько переменных, которые могут использоваться в теле цикла.

Управляющее выражение выражение-2 определяет вычисления, выполняемые перед каждой итерацией, при этом выполнение цикла продолжается до тех пор, пока выражение не станет равным 0. *Выражение-3* определяет операцию (например, приращение), которая выполняется после каждой итерации.

И *пункт-1*, и *выражение-3* могут быть опущены.
Пропущенное *выражение-2* заменяется ненулевой константой.

```
for ( int i = 0 ; i < vmax ; i++) { // обычно так
    оператор
}

for ( int i = 0 ; i < vmax ; ) {    // тут выражение-3 опускаем
    оператор
    i++;                            // а счетчиком цикла управляем тут
}

for (;;) {   // "вечный" цикл
    оператор
}
```

Оператор безусловного перехода

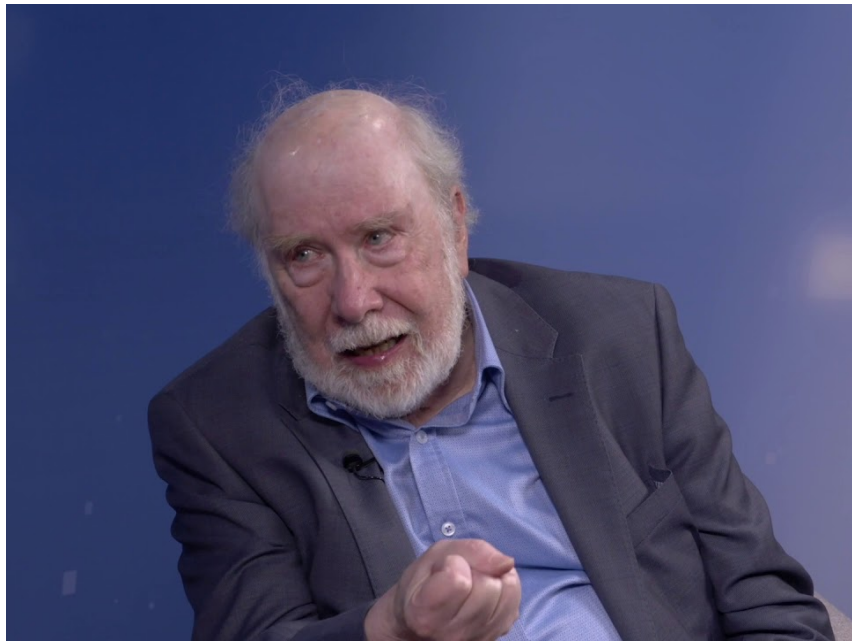
Синтаксис

оператор-безусловного-перехода:

```
goto идентификатор ;
continue ;
break ;
return [ выражение ] ;
```

Семантика

Оператор перехода вызывает безусловный переход в другое место.



Никлаус Вирт смотрит ... на один из операторов безусловного перехода как на ... Таки, догадайся, на какой он смотрит.

Никлаус Вирт — создатель языка программирования «Паскаль» и нескольких других языков (modula, modula-2, modula-3, oberon). Крут и уважаем настолько, что на <https://lurkmore.net>, о нем красноречивое молчание, но таки есть ссылка на pascal.

Цитата: «Pascal (Паскаль, ПаскакалЪ, Поссалъ, Пасквиль, ... Турбопаскаль (устар.),Труба (устар.), Делфя) — мертвый, но еще живой, язык программирования ... известным <beep>кодером ..., чтобы учить <beep>кодить школьников и ступидентов, и изначально являлся упрощённым Алголом, освоив который нетрудно было перейти на настоящий Алгол. Когда-то имел весьма солидную популярность, но сейчас ее полностью потерял, сохранившись в основном в виде Delphi для учебных и академических целей. Язык невозбранно привлекает возможностью писать почти как на обычном английском языке, а не ломать голову и пальцы о эзотерику истинности выражений `(1/3 == 0)` и `((-1 > (unsigned int)1)` и прочих извращенных приёмов, принятых в Це-подобных языках»... **Тем не менее, есть SAS Planet.**

Оператор goto

```
goto идентификатор ;
```

Семантика

Оператор **goto** вызывает безусловный переход к оператору с префиксом именованной метки в области действия функции, которая его включает.

Ограничения

Идентификатор в операторе **goto** должен называть метку, расположенную где-то в области действия функции, в которой он расположен.

Оператор **goto** не должен передавать управление с места вне пределов области видимости идентификатора, имеющей переменный изменяемый тип, в область действия этого самого идентификатора.

Пример переменного изменяемого типа — это VLA (variable length array). В данном случае размер области видимости не может быть определен на этапе компиляции, например, в ней есть массив переменного размера, который всякий раз при входе в эту область ее размер может быть устанавлена в различное значение.

```
 1: volatile int port;
 2:
 3: int main(void) {
 4:
 5:     for (int i = 5; i > 0 ; i-- ) {
 6:         if (port == 2)
 7:             goto L2;
 8:         else if (port == 3)
 9:             goto L3;
10:         { //
11:             L2: ;          // вне области переменной длинны
12:             int array[i]; // VLA – массив переменной длины
13:             L3: ;          // внутри области переменной длинны
14:         }
15:     }
16: }

$ LANG=C gcc test.c
test2.c:10:13: error: jump into scope of identifier with variably
modified type
   13 |       goto L3;
      |       ^~~~
test2.c:14:13: note: label 'L3' defined here
   17 |       L3: ;
      |       ^~
test2.c:13:17: note: 'array' declared here
   16 |       int array[i];
```

Пример 1

Иногда удобно «запрыгнуть» в середину «запутанного» набора операторов.

В следующее схеме представлен один из возможных подходов к ситуации, основанной на следующих трех допущениях:

1. Общий код инициализации обращается к объектам, видимым только в текущей функции.

2. Общий код инициализации слишком велик, чтобы оправдывать дублирование.

3. Код, определяющий очередную операцию находится в начале цикла. (Чтобы иметь возможность на него перейти, например, с помощью оператора **`continue`**).

```
goto first_time;
for (;;) {
    determine_next_operation
    if (need_to_reinitialize) {
        reinitialize_only_code        // код реинициализации
    first_time:                       // точка входа в цикл
        general_initialization_code   // общий код инициализации
        continue;
    }
    other_operations
}
```

Пример 2

Оператору **goto** не разрешается перепрыгивать любые объявления объектов с изменяемыми типами. Переход же в рамках области действия разрешен.

```
goto lab3;       // недопустимо: переход ВНУТРЬ области видимости VLA
{
    goto lab3;   // к этому моменту n и размер a[n] известны
    double a[n]; // изменяемый тип
    a[j] = 4.4;
lab3:
    a[j] = 3.3;
    goto lab4;   // допустимо: ВНУТРИ области видимости VLA
    a[j] = 5.5;
lab4:
    a[j] = 6.6;
}
goto lab4;       // недопустимо: переход ВНУТРЬ области видимости VLA
```

Оператор `continue`

```
continue ;
```

Ограничения

Оператор **`continue`** может появляться только в теле цикла или быть им.

Семантика

Оператор **`continue`** вызывает переход к коду возобновления цикла наименьшего вмещающего итерационного оператора, то есть К концу тела цикла. Точнее, в каждом из следующих операторов

```
while ( ) {
    ...
    continue;
    ...
again: ;
}
```

```
do {
    ...
    continue;
    ...
again: ;
} while ( )
```

```
for ( ) {
    ...
    continue;
    ...
again: ;
}
```

оператор **`continue`**, если он не содержится во вложенном итерационном операторе (в этом случае он интерпретируется внутри этого оператора), эквивалентен **`goto again;`**[2].

2 За меткой again: следует пустой оператор ;

Оператор `break`

```
break ;
```

Семантика

Оператор **`break`** завершает выполнение самого маленького включающего оператора **`switch`** или итерационного оператора.

Оператор **`break`** может появляться только в теле цикла или теле оператора **`switch`**.

```
switch (expr) {
case 1:
    op1;
case 2:
    op2;
case 3:
    op3;
default:
    opd; // всегда выполнится
}
```

```
switch (expr) {
case 1:
    op1;
    break;
case 2:
    op2;
    break;
case 3:
    op3;
    break;
default:
    opd;
}
```

```
switch (expr) {
case 1:
    op1;
    break;
case 2:
case 3:
    op3;
    break;
default:
    opd;
}
```

	a = 1	a = 2	a = 3	a = 4	a = ...

switch (a) {
case 1:
  op1
  goto def;
case 2:
  op2
  break;
case 3:
case 4:
  op34
default:
def:
  opd
}
...
//goto def;
	op1
	opd
		op2	op34
			opd
				op34
				opd
					opd

Оператор `return`

```
return ;
return выражение ;
```

Семантика

Оператор **`return`** завершает выполнение текущей функции и возвращает управление вызывающей стороне.

Функция может иметь любое количество операторов возврата[3].

Если выполняется оператор **`return`** с выражением, значение выражения возвращается вызывающей стороне в качестве значения выражения вызова функции.

Если выражение имеет тип, отличный от типа возврата функции, в которой оно появляется, значение преобразуется, как если бы оно было присвоено объекту, имеющему тип возврата функции.

Оператор `return` не является присваиванием.

Представление значений с плавающей точкой может иметь более широкий диапазон или точность, чем подразумевается типом возврата — в этом случае, для удаления дополнительных диапазона и точности, используется приведение типа.

Ограничения

Оператор **`return`** с выражением не должен появляться в функции, тип возврата которой **`void`**.

Оператор **`return`** без выражения должен появляться только в функции, тип возврата которой **`void`**.

---

3 Н. Вирт опять негодуэ

Пример

```
struct s { double i; } f(void); // объявление функции f()
union {
    struct {
        int f1;
        struct s f2;
    } u1;
    struct {
        struct s f3;
        int f4;
    } u2;
} g;

struct s f(void) {              // определение функции f()
    return g.u1.f2;
}
/* ... */
g.u2.f3 = f();
```

Здесь неопределенное поведение не возникает, хотя было бы, если бы присвоение было выполнено напрямую без использования вызова функции для извлечения значения.