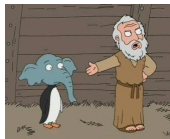


Язык программирования Си

Метки и переходы

```
switch (number % 4) {  
    case 0:  
        do {  
            ++i;  
        case 3: ++i;  
        case 2: ++i;  
        case 1: ++i;  
        } while (count-- > 0);  
}
```



Владимир Валерьевич Соловьёв
Huawei, НГУ
vladimir.conwor@gmail.com
t.me/conwor
vk.com/conwor

Метки

Метки (*labels*) - идентификаторы, обозначающие позиции в коде; должны быть уникальными внутри функции

Метки

Метки (labels) - идентификаторы, обозначающие позиции в коде; должны быть уникальными внутри функции

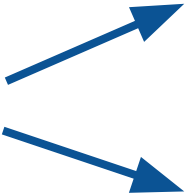
Используются для передачи управления: безусловной, с помощью оператора **goto**, и условной, с помощью оператора **switch**

Метки и goto

```
    int i = 0;
    goto check;
loop:
    printf("%d ", i);
    i++;
check:
    if (i < 10) {
        goto loop;
    }
```

Метки и goto

Метки

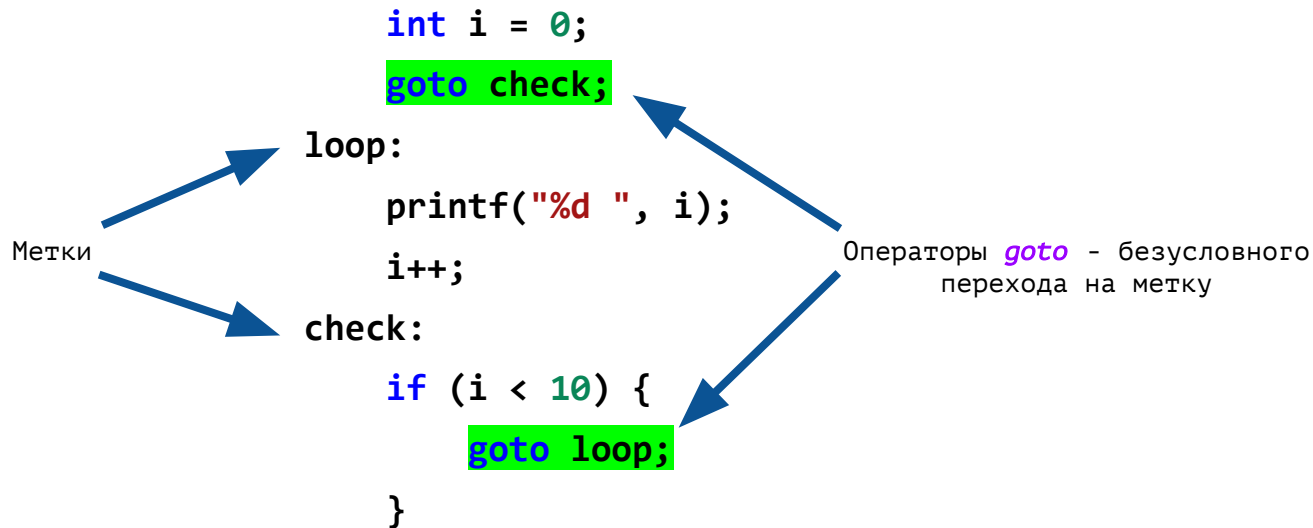


```
int i = 0;
goto check;

loop:
printf("%d ", i);
i++;

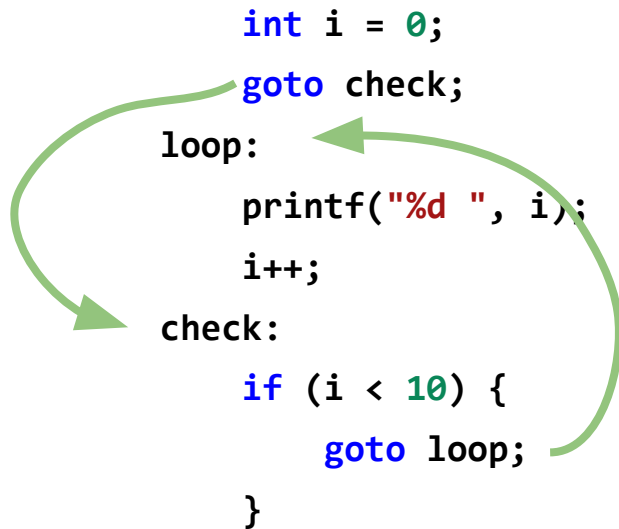
check:
if (i < 10) {
    goto loop;
}
```

Метки и goto

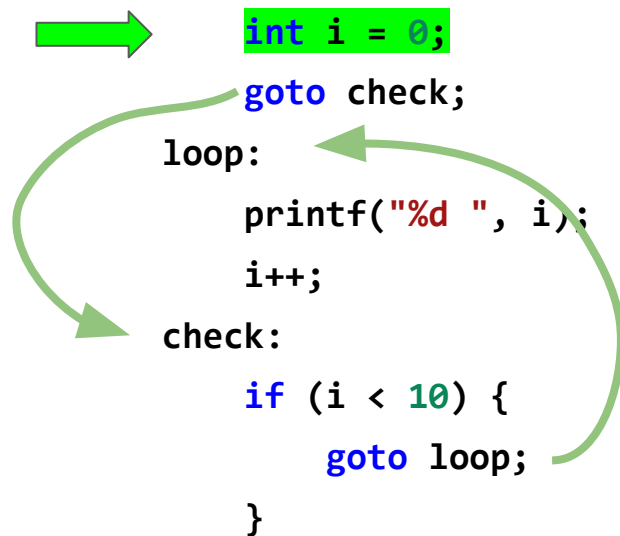


Метки и goto

```
int i = 0;  
goto check;  
loop:  
    printf("%d ", i);  
    i++;  
check:  
    if (i < 10) {  
        goto loop;  
    }
```

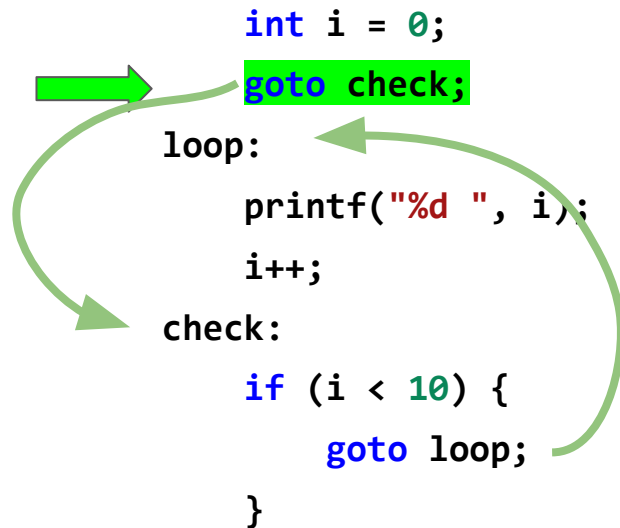


Метки и goto



i	0
---	---

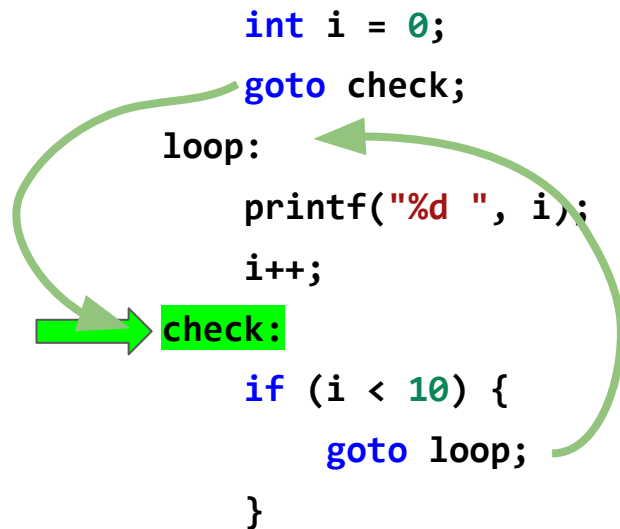
Метки и goto



i	0
---	---

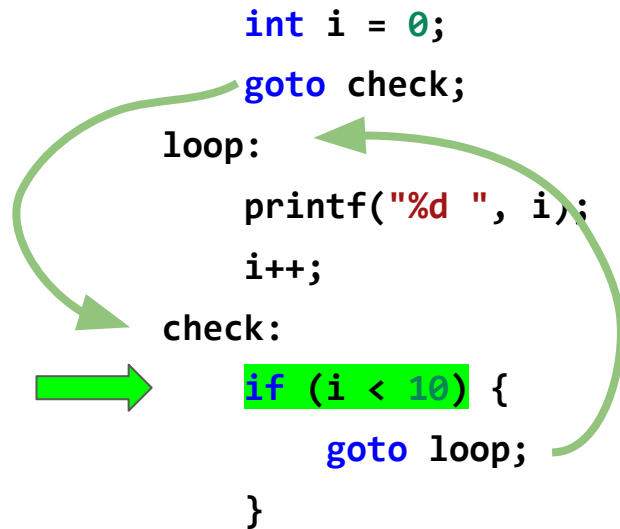
Метки и goto

```
int i = 0;  
goto check;  
loop:  
    printf("%d ", i);  
    i++;  
check:  
    if (i < 10) {  
        goto loop;  
    }
```



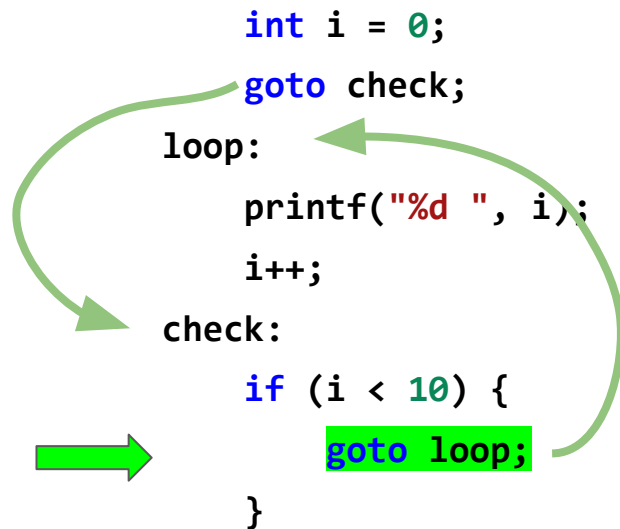
i	0
---	---

Метки и goto



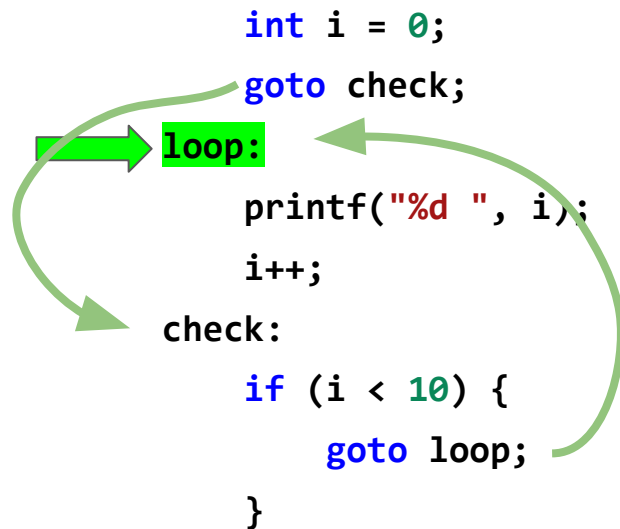
i	0
---	---

Метки и goto



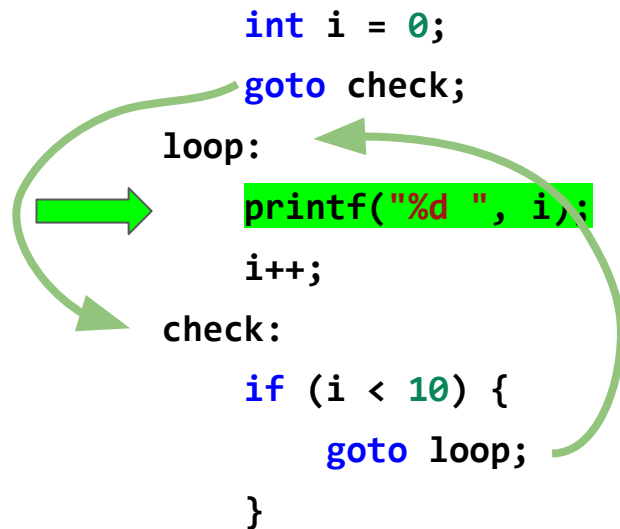
i	0
---	---

Метки и goto



i	0
---	---

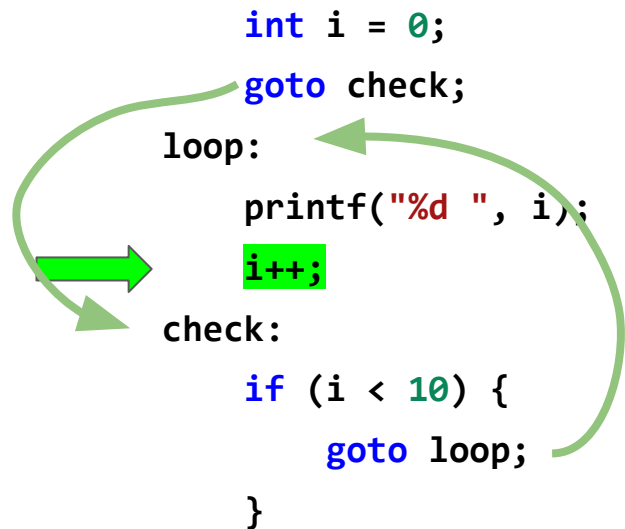
Метки и goto



i	0
---	---

> 0

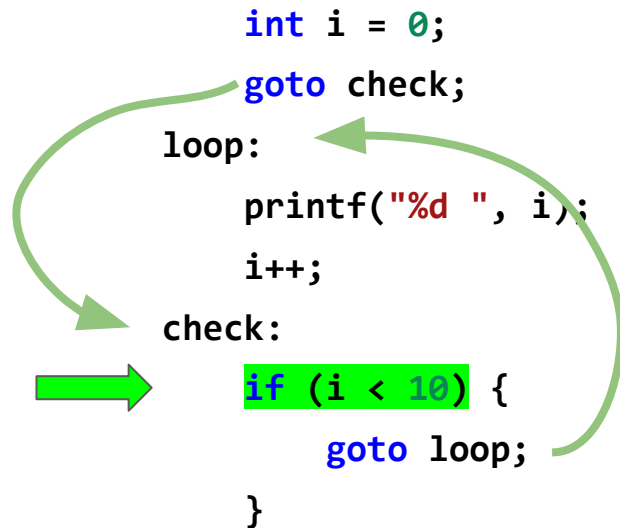
Метки и goto



i	1
---	---

> 0

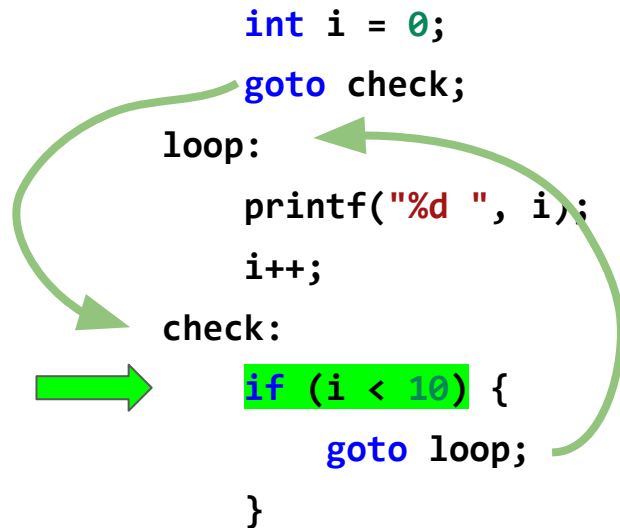
Метки и goto



i	1
---	---

> 0

Метки и goto

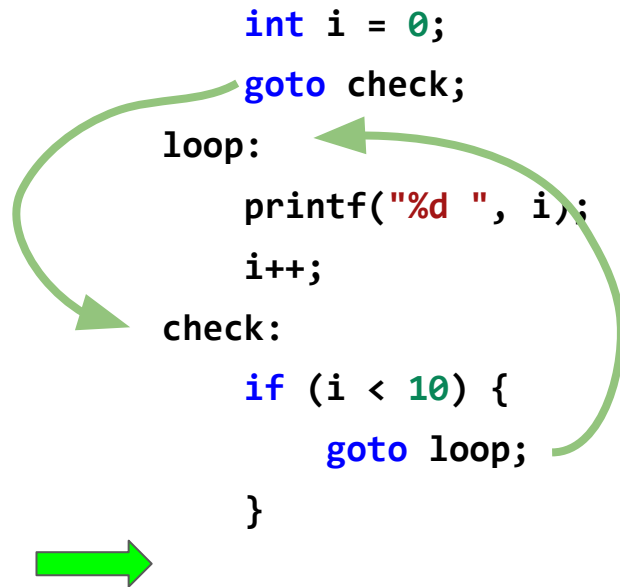


i	10
---	----

> 0 1 2 3 4 5 6 7 8 9

Через ещё 9 шагов

Метки и goto

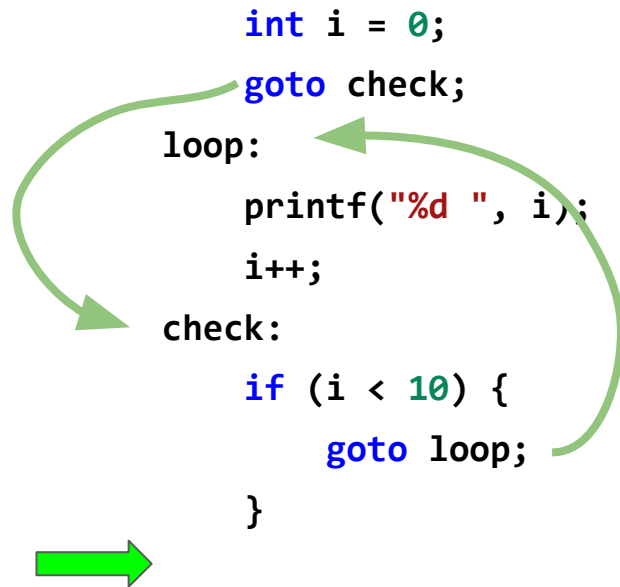


i	10
---	----

> 0 1 2 3 4 5 6 7 8 9

Через ещё 9 шагов

Метки и goto



```
for (int i = 0; i < 10; i++) {
    printf("%d ", i);
}
```

i	10
---	----

> 0 1 2 3 4 5 6 7 8 9

Через ещё 9 шагов

Метки и `goto`

`goto` может перейти только на метку внутри функции, в которой написан

Метки и **goto**

goto может перейти только на метку внутри функции, в которой написан

Никакого смысла писать обычные циклы и условные переходы, используя **goto**, разумеется, нет - только запутывает код и повышает риск ошибок

Метки и **goto**

goto может перейти только на метку внутри функции, в которой написан

Никакого смысла писать обычные циклы и условные переходы, используя **goto**, разумеется, нет - только запутывает код и повышает риск ошибок

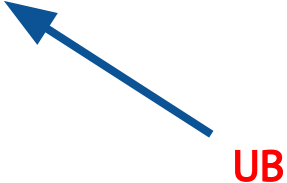
Самая большая ошибка с **goto** - переход в область видимости переменной в обход её инициализации (**UB**) или в область видимости **VM type** в обход его определения (ошибка компиляции)

Обход инициализации переменной

```
    goto foo;  
    int i;  
    scanf("%d", &i);  
foo:  
    printf("%d", i);  
    return 0;
```

Обход инициализации переменной

```
goto foo;  
int i;  
scanf("%d", &i);  
foo:  
printf("%d", i);  
return 0;
```



UB

Обход определения VM type

```
    goto foo;  
    int i;  
    scanf("%d", &i);  
    int arr[i];  
foo:  
    return 0;
```

Обход определения VM type

```
goto foo;  
int i;  
scanf("%d", &i);  
int arr[i];  
foo:  
return 0;
```

Ошибка компиляции!

main.c:4:5: error: jump into scope of identifier
with variably modified type

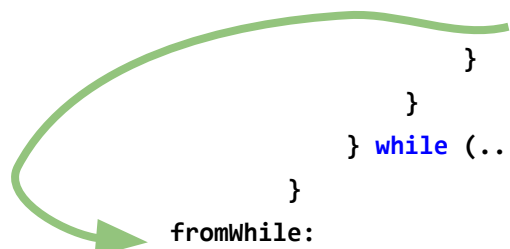
Единственное разумное использование **goto**

С помощью оператора **goto** можно выйти из нескольких вложенных циклов одним действием (оператор **break** выходит только из одного максимально вложенного цикла)

Единственное разумное использование **goto**

С помощью оператора **goto** можно выйти из нескольких вложенных циклов одним действием (оператор **break** выходит только из одного максимально вложенного цикла)

```
    for (...) {  
        while (...) {  
            do {  
                for (...) {  
                    if (...) {  
                        goto fromWhile;  
                    }  
                }  
            } while (...)  
        }  
    }  
fromWhile:  
    ...  
}
```



Единственное разумное использование **goto**

С помощью оператора **goto** можно выйти из нескольких вложенных циклов одним действием (оператор **break** выходит только из одного максимально вложенного цикла)

Однако, подобный код - редкость, и зачастую его лучше преобразовать каким-нибудь другим способом (например, разделить на функции)

Единственное разумное использование **goto**

С помощью оператора **goto** можно выйти из нескольких вложенных циклов одним действием (оператор **break** выходит только из одного максимально вложенного цикла)

Однако, подобный код - редкость, и зачастую его лучше преобразовать каким-нибудь другим способом (например, разделить на функции)

Знать оператор **goto** нужно, потому что это базовый элемент передачи управления, и потому что только его аналоги и используются в программировании на ассемблере (весь ваш следующий семестр); использовать в Си - не нужно

Оператор выбора (**switch**)

```
switch (x) {  
    case 0:  
        ...  
        break;  
    case 37:  
        ...  
        break;  
    case 42:  
        ...  
        break;  
    default:  
        ...  
}
```

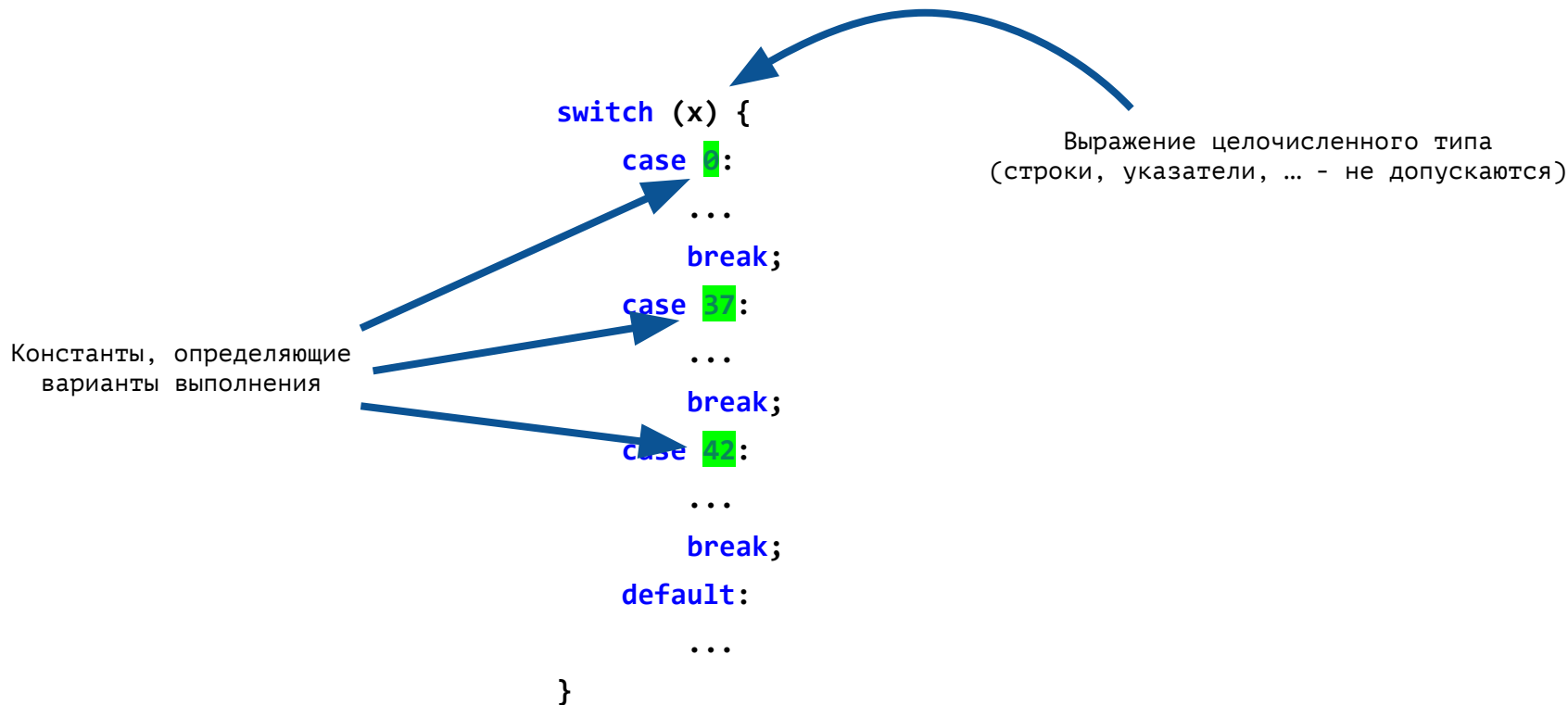
Оператор выбора (switch)

```
switch (x) {  
    case 0:  
        ...  
        break;  
    case 37:  
        ...  
        break;  
    case 42:  
        ...  
        break;  
    default:  
        ...  
}
```

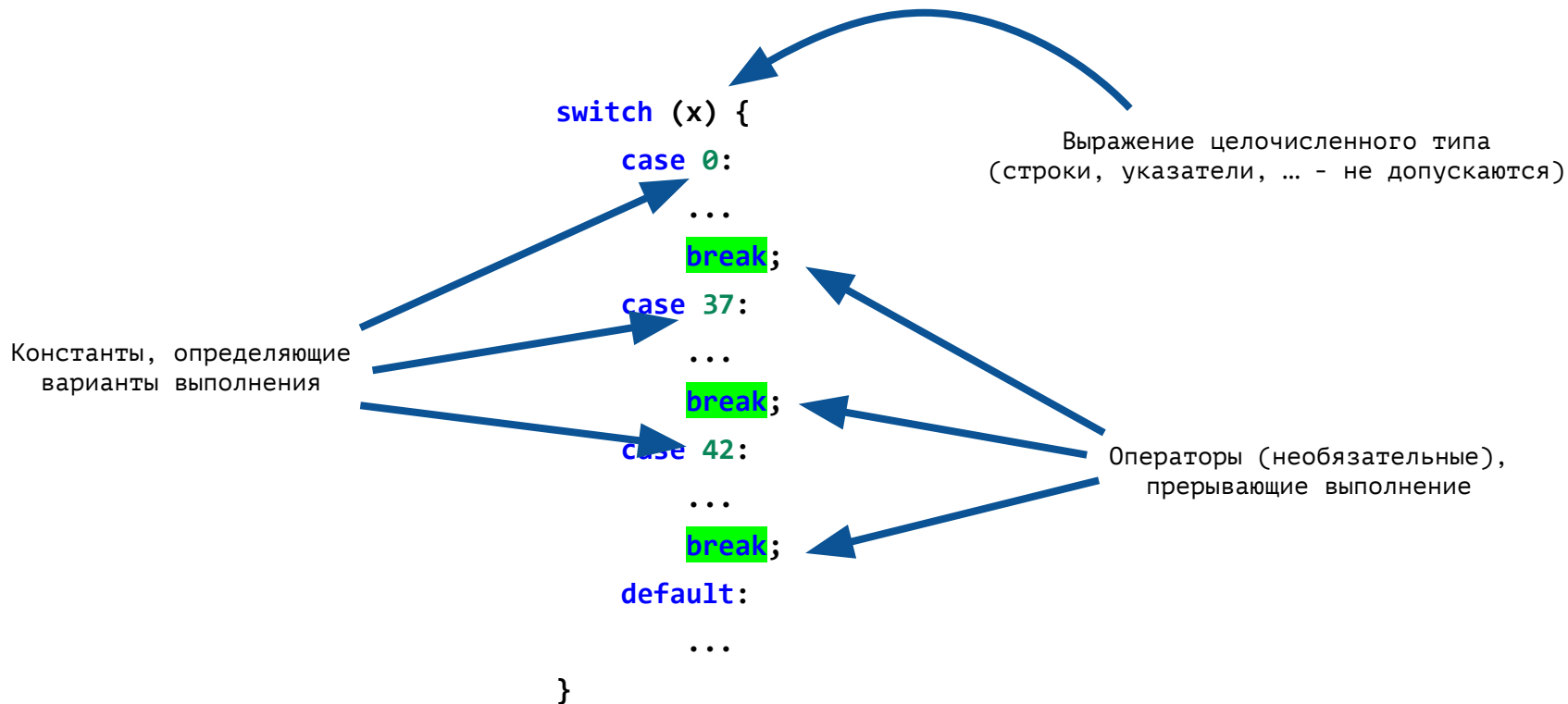


Выражение целочисленного типа
(строки, указатели, ... - не допускаются)

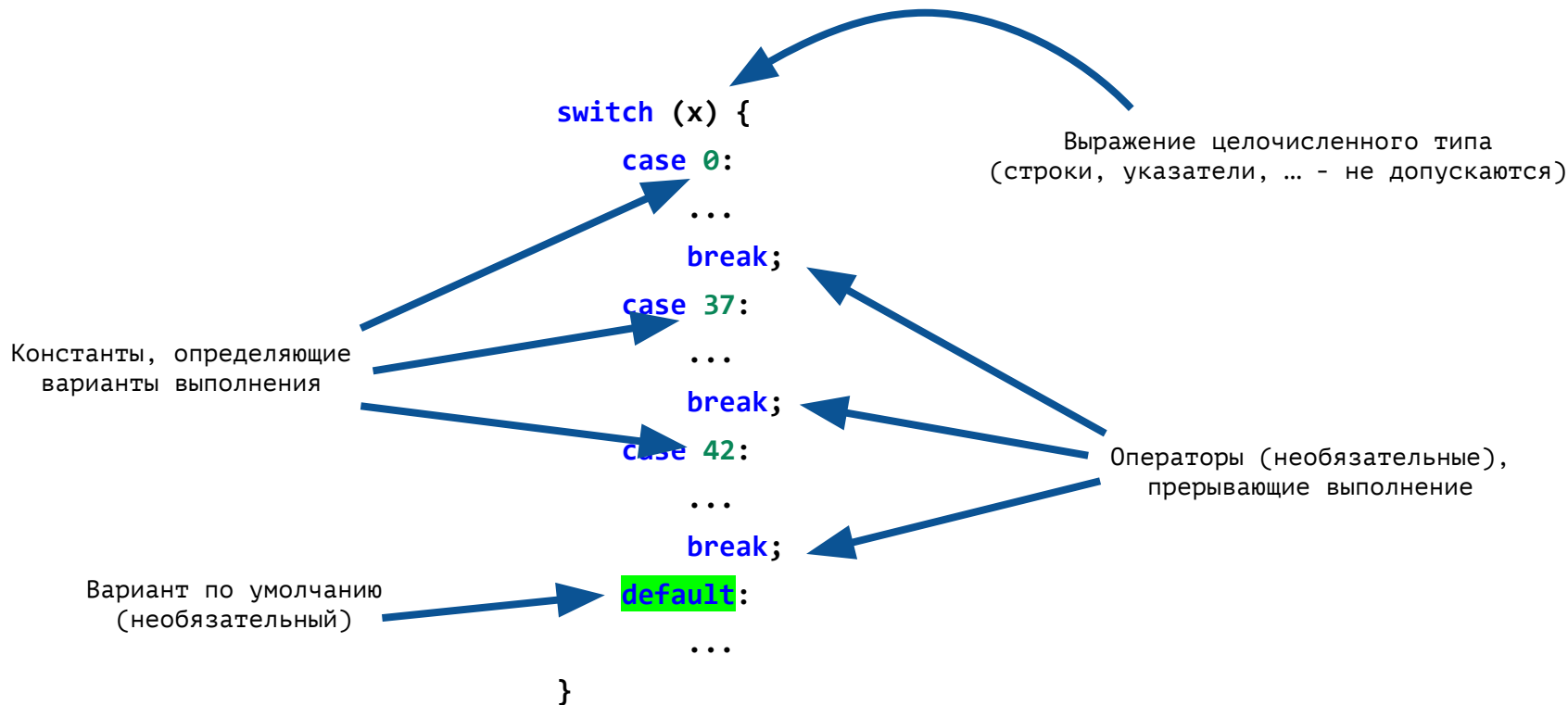
Оператор выбора (switch)



Оператор выбора (switch)



Оператор выбора (switch)



Семантика оператора выбора

Значение выражения сравнивается с константами, и управление передаётся на вариант кода, написанный под совпадающей константой

Семантика оператора выбора

Значение выражения сравнивается с константами, и управление передаётся на вариант кода, написанный под совпадающей константой

Если такой нет, то на вариант кода по умолчанию

Семантика оператора выбора

Значение выражения сравнивается с константами, и управление передаётся на вариант кода, написанный под совпадающей константой

Если такой нет, то на вариант кода по умолчанию

Если и его нет, то никакой вариант не выполняется

Операторы **break** внутри оператора выбора

Нужны для того, чтобы прервать выполнение варианта кода, если это нужно

Операторы **break** внутри оператора выбора

Нужны для того, чтобы прервать выполнение варианта кода, если это нужно

Если оператора **break** нет, то после выполнения выбранного варианта начнётся выполнение следующего, и так далее до конца, либо всё-таки до оператора **break** - такое поведение называется *fallthrough*

Операторы **break** внутри оператора выбора

Нужны для того, чтобы прервать выполнение варианта кода, если это нужно

Если оператора **break** нет, то после выполнения выбранного варианта начнётся выполнение следующего, и так далее до конца, либо всё-таки до оператора **break** - такое поведение называется *fallthrough*

Fallthrough поведение нужно обычно для того, чтобы сделать один вариант кода для нескольких разных значений

```
void what_is_it(char c) {  
    switch (c) {  
        case '0':  
        case '2':  
        case '4':  
        case '6':  
        case '8':  
            printf("even digit");  
            break;  
        case '1':  
        case '3':  
        case '5':  
        case '7':  
        case '9':  
            printf("odd digit");  
            break;  
        default:  
            printf("not a digit");  
    }  
}
```

Области видимости внутри **switch**

Отдельный **case** не создаёт области видимости, поэтому если вы заведёте локальную переменную под **case**, она будет видна во всём **switch** (и будет конфликтовать с другими переменными с таким же именем)

Области видимости внутри **switch**

Отдельный **case** не создаёт области видимости, поэтому если вы заведёте локальную переменную под **case**, она будет видна во всём **switch** (и будет конфликтовать с другими переменными с таким же именем)

Чтобы обойти это, можно окружить тело **case** фигурными скобками - создать **блок**

То же самое можно сделать вообще в любом месте, необязательно внутри **switch**

Области видимости внутри **switch**

```
switch (x) {  
    case 0:  
        int y = ...;  
        ...  
    case 1:  
        int y = ...;  
        ...  
    ...  
}
```

Области видимости внутри **switch**

```
switch (x) {  
    case 0:  
        int y = ...;  
        ...  
    case 1:  
        int y = ...;  
        ...  
    ...  
}
```



Ошибка компиляции

Области видимости внутри **switch**

```
switch (x) {  
    case 0: {  
        int y = ...;  
        ...  
    }  
    case 1: {  
        int y = ...;  
        ...  
    }  
    ...  
}
```

Области видимости внутри **switch**

```
switch (x) {  
    case 0: {  
        int y = ...;  
        ...  
    }  
    case 1: {  
        int y = ...;  
        ...  
    }  
    ...  
}
```

Области видимости внутри **switch**

```
switch (x) {  
    case 0: {  
        int y = ...;  
        ...  
    }  
    case 1: {  
        int y = ...;  
        ...  
    }  
    ...  
}
```

Истинное лицо оператора **switch**

Логiku **fallthrough** , отсутствие областей видимости в каждом **case** , ограничение на константы в **case** и целочисленность проверяемого значения можно понять, только зная, как реализован **switch**

Истинное лицо оператора **switch**

Логику **fallthrough** , отсутствие областей видимости в каждом **case** , ограничение на константы в **case** и целочисленность проверяемого значения можно понять, только зная, как реализован **switch**

Каждый **case** - это просто метка

Можно представить себе это примерно так: `case 42: => case_42:`

Истинное лицо оператора **switch**

Логику **fallthrough** , отсутствие областей видимости в каждом **case** , ограничение на константы в **case** и целочисленность проверяемого значения можно понять, только зная, как реализован **switch**

Каждый **case** - это просто метка

Можно представить себе это примерно так: `case 42: => case_42:`

Оператор **switch** - это набор условных операторов и операторов **goto** на соответствующие метки

Истинное лицо оператора **switch**

Логику **fallthrough** , отсутствие областей видимости в каждом **case** , ограничение на константы в **case** и целочисленность проверяемого значения можно понять, только зная, как реализован **switch**

Каждый **case** - это просто метка

Можно представить себе это примерно так: **case 42:** => **case_42:**

Оператор **switch** - это набор условных операторов и операторов **goto** на соответствующие метки

Фигурные скобки нужны для создания пространства имён меток и работы **break** операторов

```
switch (x) {  
    case 0:  
        ...  
        break;  
    case 1:  
        ...  
    case 2:  
        ...  
        break;  
    default:  
        ...  
}
```

```
switch (x) {  
    case 0:  
        ...  
        break;  
    case 1:  
        ...  
    case 2:  
        ...  
        break;  
    default:  
        ...  
}
```

```
if (x == 0) goto case_0;  
else if (x == 1) goto case_1;  
else if (x == 2) goto case_2;  
else goto default_case;
```

```
case_0:  
    ...  
    goto switch_end;  
case_1:  
    ...  
case_2:  
    ...  
    goto switch_end;  
default_case:  
    ...
```

```
switch_end:
```

Эффективность реализации выбора

Цепочка операторов сравнения будет работать в среднем за $O(N)$, где N - количество вариантов сравнения

Эффективность реализации выбора

Цепочка операторов сравнения будет работать в среднем за $O(N)$, где N - количество вариантов сравнения

Можно влиять на эту оценку переупорядочиванием вариантов (более частые - наверх), но это может сделать код хуже, а также не поможет при равномерном распределении частот вариантов

Эффективность реализации выбора

Цепочка операторов сравнения будет работать в среднем за $O(N)$, где N - количество вариантов сравнения

Можно влиять на эту оценку переупорядочиванием вариантов (более частые - наверх), но это может сделать код хуже, а также не поможет при равномерном распределении частот вариантов

Можно ли реализовать выбор эффективнее?

Вспоминаем про ограничения на целочисленные типы и константы в `case`

Lookup switch

Сортируем последовательность констант и реализуем оператор **switch** не последовательным сравнением с каждой константой, а бинарным поиском - это называется *lookup switch*

Lookup switch

Сортируем последовательность констант и реализуем оператор **switch** не последовательным сравнением с каждой константой, а бинарным поиском - это называется *lookup switch*

Сортировка будет происходить в compile-time, а в run-time поиск совпадающего варианта сократится до $O(\log_2(N))$

```
if (x == 0) goto case_0;  
if (x == 1) goto case_1;  
if (x == 2) goto case_2;  
if (x == 3) goto case_3;  
if (x == 4) goto case_4;  
if (x == 5) goto case_5;  
if (x == 6) goto case_6;  
if (x == 7) goto case_7;  
goto default_case;
```

```
if (x == 0) goto case_0;
if (x == 1) goto case_1;
if (x == 2) goto case_2;
if (x == 3) goto case_3;
if (x == 4) goto case_4;
if (x == 5) goto case_5;
if (x == 6) goto case_6;
if (x == 7) goto case_7;
goto default_case;
```

```
if (x <= 3) {
    if (x <= 1) {
        if (x == 0) goto case_0;
        if (x == 1) goto case_1;
    } else {
        if (x == 2) goto case_2;
        goto case_3;
    }
} else {
    if (x <= 5) {
        if (x == 4) goto case_4;
        goto case_5;
    } else {
        if (x == 6) goto case_6;
        if (x == 7) goto case_7;
    }
}
goto default_case;
```

```
if (x == 0) goto case_0;
if (x == 1) goto case_1;
if (x == 2) goto case_2;
if (x == 3) goto case_3;
if (x == 4) goto case_4;
if (x == 5) goto case_5;
if (x == 6) goto case_6;
if (x == 7) goto case_7;
goto default_case;
```

4 проверки в среднем

```
if (x <= 3) {
    if (x <= 1) {
        if (x == 0) goto case_0;
        if (x == 1) goto case_1;
    } else {
        if (x == 2) goto case_2;
        goto case_3;
    }
} else {
    if (x <= 5) {
        if (x == 4) goto case_4;
        goto case_5;
    } else {
        if (x == 6) goto case_6;
        if (x == 7) goto case_7;
    }
}
goto default_case;
```

```
if (x == 0) goto case_0;
if (x == 1) goto case_1;
if (x == 2) goto case_2;
if (x == 3) goto case_3;
if (x == 4) goto case_4;
if (x == 5) goto case_5;
if (x == 6) goto case_6;
if (x == 7) goto case_7;
goto default_case;
```

4 проверки в среднем

```
if (x <= 3) {
    if (x <= 1) {
        if (x == 0) goto case_0;
        if (x == 1) goto case_1;
    } else {
        if (x == 2) goto case_2;
        goto case_3;
    }
} else {
    if (x <= 5) {
        if (x == 4) goto case_4;
        goto case_5;
    } else {
        if (x == 6) goto case_6;
        if (x == 7) goto case_7;
    }
}
goto default_case;
```

3.25 проверки в среднем

```

if (x == 0) goto case_0;
if (x == 1) goto case_1;
if (x == 2) goto case_2;
if (x == 3) goto case_3;
if (x == 4) goto case_4;
if (x == 5) goto case_5;
if (x == 6) goto case_6;
if (x == 7) goto case_7;
goto default_case;

```

4 проверки в среднем

Разница небольшая, но по ассимптотике это $O(N)$ против $O(\log_2(N))$ - огромная разница при увеличении количества вариантов

```

if (x <= 3) {
    if (x <= 1) {
        if (x == 0) goto case_0;
        if (x == 1) goto case_1;
    } else {
        if (x == 2) goto case_2;
        goto case_3;
    }
} else {
    if (x <= 5) {
        if (x == 4) goto case_4;
        goto case_5;
    } else {
        if (x == 6) goto case_6;
        if (x == 7) goto case_7;
    }
}
goto default_case;

```

3.25 проверки в среднем

Напоминание: управление

Машинный код функций лежит в той же памяти, что и данные программы, у всех команд есть адреса

В процессоре есть *регистры* - быстрые ячейки памяти; одним из основных регистров является *счётчик команд (program counter, instruction pointer, PC, IP)* - регистр, в котором находится адрес следующей команды, которая должна исполниться

Вся передача управления основана на манипуляции регистром **PC**, например, при обычном исполнении регистр **PC** просто увеличивается на размер текущей команды

Table switch

Создадим таблицу адресов для каждого **case** и **default** варианта: номер элемента таблицы - значение константы в **case** , значение - адрес

Операция выбора реализуется за **$O(1)$** чтением элемента массива - это называется **table switch**

Table switch

```
if (0 <= x && x <= 6) {  
    PC = table[x];  
} else {  
    goto default_case;  
}
```

```
case_0: ...
```

```
case_1: ...
```

```
case_2: ...
```

```
case_3: ...
```

```
case_5: ...
```

```
case_6: ...
```

```
default_case: ...
```

Table switch

[6]	
[5]	
[4]	
[3]	
[2]	
[1]	
[0]	

```
if (0 <= x && x <= 6) {  
    PC = table[x];  
} else {  
    goto default_case;  
}
```

case_0: ...

case_1: ...

case_2: ...

case_3: ...

case_5: ...

case_6: ...

default_case: ...

Table switch

[6]	
[5]	
[4]	
[3]	
[2]	
[1]	
[0]	case_0

```
if (0 <= x && x <= 6) {  
    PC = table[x];  
} else {  
    goto default_case;  
}
```

case_0: ...

case_1: ...

case_2: ...

case_3: ...

case_5: ...

case_6: ...

default_case: ...

Table switch

[6]	case_6
[5]	case_5
[4]	
[3]	case_3
[2]	case_2
[1]	case_1
[0]	case_0

```
if (0 <= x && x <= 6) {  
    PC = table[x];  
} else {  
    goto default_case;  
}
```

case_0: ...

case_1: ...

case_2: ...

case_3: ...

case_5: ...

case_6: ...

default_case: ...

Table switch

[6]	case_6
[5]	case_5
[4]	default_case
[3]	case_3
[2]	case_2
[1]	case_1
[0]	case_0

```
if (0 <= x && x <= 6) {  
    PC = table[x];  
} else {  
    goto default_case;  
}
```

case_0: ...

case_1: ...

case_2: ...

case_3: ...

case_5: ...

case_6: ...

default_case: ...

Table switch

[6]	case_6
[5]	case_5
[4]	default_case
[3]	case_3
[2]	case_2
[1]	case_1
[0]	case_0

Генерируется компилятором
и размещает в статической
памяти

```
if (0 <= x && x <= 6) {  
    PC = table[x];  
} else {  
    goto default_case;  
}
```

case_0: ...

case_1: ...

case_2: ...

case_3: ...

case_5: ...

case_6: ...

default_case: ...

Table switch

Паразитные строки с адресом **default_case** нужны, чтобы переход осуществлялся за **$O(1)$**

Table switch

Паразитные строки с адресом **default_case** нужны, чтобы переход осуществлялся за **$O(1)$**

Если таблица констант сильно разреженная, накладные расходы могут перевесить плюсы по производительности

Table switch

Паразитные строки с адресом **default_case** нужны, чтобы переход осуществлялся за **$O(1)$**

Если таблица констант сильно разреженная, накладные расходы могут перевесить плюсы по производительности

В любом случае, независимо от стратегии, которую выберет компилятор, оператор **switch** будет всегда реализован эффективнее, чем серия сравнения при количестве вариантов больше примерно 8

Девайс Даффа

Один из самых необычных и красивых примеров на языке Си, созданный с использованием особенностей оператора `switch`, таких как `fallthrough` поведение и отсутствие блоков, образованных `case`

Для любознательных - [статья](#)

```
switch (number % 4) {  
    case 0:  
        do {  
            ++i;  
        case 3: ++i;  
        case 2: ++i;  
        case 1: ++i;  
        } while (count-- > 0);  
}
```