Циклы

Проблема

Задача 1: вводятся 3 целых числа, найти их сумму.

Задача 2: вводится 10 целых чисел, найти их сумму.

Задача 3: вводится число n, а затем n целых чисел, найти их сумму.

Проблема

Задача 1: вводятся 3 целых числа, найти их сумму.

```
int x, y, z;
std::cin >> x >> y >> z;
std::cout << x + y + z << '\n';
```

Задача 2: вводится 10 целых чисел, найти их сумму.

```
int sum = 0;
int x;
std::cin >> x; sum += x;  // 1
std::cin >> x; sum += x;  // 2
// ...
std::cin >> x; sum += x;  // 10
```

Задача 3: вводится число n, а затем n целых чисел, найти их сумму.

```
???
```

Проблема

Задача 3: вводится число n, а затем n целых чисел, найти их сумму.

```
???
```

Можно расписать все вручную и даже написать скрипт, который бы генерировал код для фиксированного n.

Но что делать, когда n неизвестен на этапе компиляции?

Цикл `while`

Цикл - оператор, позволяющий организовать повторяющееся выполнение другого оператора.

```
while (<condition>) <statement>
```

- `condition` - либо выражение, либо объявление переменной с инициализатором. В любом случае значение должно быть приводимо к `bool`.
- `statement` - оператор (может быть составной)

Цикл `while`: решение задачи

Вводится n целых чисел, найти их сумму.

```
int n;
std::cin >> n;

int sum = 0;
while (n > 0) {
  int x;
  std::cin >> x;
  sum += x;
  --n;
}

std::cout << sum << '\n';
```

Цикл `while`: примеры

```
// бесконечный цикл
while (true) std::cout << 0;
```

```
x = -5;
while (int sqr = x * x) {
  ++x;
  std::cout << sqr << '\n';
}
```

```
// пустой цикл (крутится пока верен x)
while (x);
```

Замечание: последний цикл - *Undefined Behaviour*, если *x* не изменяет своего результата и не имеет побочных действий.

Цикл `do-while`

```
do <statement> while (<condition>);
```

Цикл `do-while` аналогичен циклу `while`, за исключением того, что оператор цикла выполняется до проверки условия.

Таким образом, гарантируется, что цикл совершит хотя бы одну итерацию.

```
int x;
do {
  std::cin >> x;
  std::cout << x * x << '\n';
} while (x);
```

Цикл `for`

Вернемся к решению задачи

Вводится n целых чисел, найти их сумму.

```
int n;
std::cin >> n;

int sum = 0;
int i = 0;  // номер вводимого числа
while (i < n) {
  int x;
  std::cin >> x;
  sum += x;
  ++i;
}

std::cout << sum << '\n';
```

Что вам не нравится в этом решении?

Цикл `for`

Вводится n целых чисел, найти их сумму.

```
int n;
std::cin >> n;

int sum = 0;
int i = 0;  // переменная инициализируется и видна вне цикла
while (i < n) {  // сколько итераций будет выполнено?
  int x;
  std::cin >> x;
  sum += x;
  ++i;  // как не забыть увеличить счетчик / найти его в коде
}

std::cout << sum << '\n';
```

Цикл `for`

`for ([init]; [condition]; [expression]) <statement>`

- `init` - либо выражение, либо объявление. Область действия объявленной сущности совпадает с областью оператора.
- `condition` - либо выражение, либо объявление переменной с инициализатором. В любом случае значение должно быть приводимо к `bool`.
- `expression` - произвольное выражение, выполняющееся в конце итерации
- `statement` - оператор, выполняющийся в цикле

```
for (int i = 0; i < n; ++i) ...
```

Цикл `for`

`for ([init]; [condition]; [expression]) <statement>`

Цикл `for` эквиваленте циклу `while` следующего вида:

```
{
  [init];
  while ([condition]) {  // while(true), если condition пуст
    <statement>
    [expression];
  }
}
```

но при этом гораздо лучше читаем, поэтому на практике чаще используется `for`.

Цикл `for`: примеры

```
for (int i = 0; i < n; ++i) std::cout << i << '\n';
```

```
for (int i = 0; i < n; i += 2) {
  std::cout << i << '\n';
}
```

```
for (std::cin >> x; x != 0; std::cin >> x) {
  std::cout << x * x << '\n';
}
```

Цикл `for`: примеры

```
// бесконечный цикл
for (;;) ...
```

```
// аналог while
for (; x;) ...
```

```
for (int i = 0, j = 0; i < n && j < m; ++i, ++j) ...
```

Управляющие операторы

Оператор `break`

Оператор `break` позволяет досрочно завершить выполнение цикла:

```
for (int i = 0; i < n; ++i) {
  int x;
  std::cin >> x;
  if (x == 0) {
    std::cout << "Division by zero\n";
    break;
  }
  std::cout << y / x << '\n';
}
```

Оператор `continue`

Оператор `continue` позволяет досрочно завершить **текущую итерацию**:

```
for (int i = 0; i < n; ++i) {
  int x;
  std::cin >> x;
  if (x == 0) {
    std::cout << "Division by zero\n";
    continue;  // завершаем эту итерацию - выполняем ++i, проверяем i < n
  }
  std::cout << y / x << '\n';
}
```

Оператор безусловного перехода

Оператор `goto`*

***оператор запрещенный на территории нашего курса, выполняющий функции иностранного агента**

Оператор `goto` позволяет совершить "прыжок" в произвольное место функции, обозначенное некоторой "меткой".

```
// программа считает x и завершит работу
int main() {
  int x;
  std::cin >> x;
  goto label;

  int y;
  std::cin >> y;
  std::cout << x + y << '\n';

label:
  return 0;
}
```

Оператор `goto`*

***оператор запрещенный на территории нашего курса, выполняющий функции иностранного агента**

Через `goto` может быть реализован цикл. Также он лежит в основе `switch`:

```
for (int i = 0; i < n; ++i) ...

// <=>

  int i = 0;
loop:
  if (i < n) {
    ...
    ++i;
    goto loop;
  }
```

Оператор `goto`*

***оператор запрещенный на территории нашего курса, выполняющий функции иностранного агента**

Что вы думаете про следующие куски кода?

```
  int x = 0;
label:
  int y = 1;
  ...
  goto label;
```

```
  goto label;
  int x = 0;
  int y = 1;
label:
  std::cout << x * y << '\n';
```

Оператор `goto`*

***оператор запрещенный на территории нашего курса, выполняющий функции иностранного агента**

В первом примере все корректно: переменная проинициализируется заново

```
  int x = 0;
label:
  int y = 1;
  ...
  goto label;
```

Во втором примере - *UB*: использование переменных без инициализации.

```
  goto label;
  int x = 0;
  int y = 1;
label:
  std::cout << x * y << '\n';
```

Оператор `goto`*

***оператор запрещенный на территории нашего курса, выполняющий функции иностранного агента**

Оператор `goto` сильно усложняет чтение и отладку программ.

Во всех (даже безвыходных) ситуациях можно обойтись без него.

"... the quality of programmers is a decreasing function of the density of `goto` statements int the programs they produce." Edgar Dijkstra, Go To Statement Considered Harmful, 1968