```
// the program counts x and exits
int main() {
  int x;
  std::cin >> x;
  goto label;

  int y;
  std::cin >> y;
  std::cout << x + y << '\n';
  label:
  return 0;
}
```

Через `goto` может быть реализован цикл:

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

Также оператор `goto` лежит в основе оператора `switch`, так как все случаи (`case` и `default`) являются метками для перехода. Поэтому даже следующая программа скомпилируется:

```
switch (x) {
  case 0: std::cout << "Zero\n"; break;
  case 1: std::cout << "One\n"; break;
  deault: std::cout << "Other\n"; // with a missing letter f
}
```

Оператор `goto` сильно усложняет чтение и отладку программ.
Во всех (даже безвыходных) ситуациях можно обойтись без него.

6 Указатели

6.1 операция(operator) взятия адреса

Согласно модели памяти C++ каждая ячейка имеет свой адрес. Для того, чтобы узнать адрес, по которому лежит объект в памяти, необходимо использовать `&`

```
int x;
std::cout << &x << '\n'; // displayed in hexadecimal
```

Операцию можно применять только к *lvalue* значениям. Более того, **возможность взятия адреса можно использовать в качестве критерия lvalue**:

```
// it is possible
&x, &(x = 8), &(++x);

// that's not possible
&5, &(x + 8), &(x++);
```

Адреса в C++ имеют специальный тип — указатель на тип объекта, у которого был взят адрес.

```
int x;
const float y = 0;

&x; // pointer to int [int*]
&y; // pointer to const float [const float*]
```

6.2 Указатели

Указатель — тип данных, позволяющий хранить адрес другого объекта в памяти.

Пусть `T` — некоторый тип, тогда `T*` — указатель на `T`. Таким образом, указатели — это целое семейство типов.

```
int* // pointer to int
float* // pointer to float
long long* // pointer to long long
const char** // pointer to pointer to const char
double*** // pointer to pointer to pointer to double
```

6.3 Примеры

Примерами использования указателей могут быть следующие:

```
int x = 0;
const float y = 0;

int* px = &x;
int* px2 = &(++x); // == px

const float* cpy = &y;
const int* cpx = &x;

int** ppx = &px;
const int** cppx = &cpx;
```

Указатели учитывают константность!

```
float* py = &y; // CE: py breaks the constness of y
```

6.4 Разыменование

По указателю можно получать значение объекта и даже изменять его.

Для этого воспользуемся операцией разыменования (`*`)

Формально: операция(operator) разыменования — унарная операция(operator), применяемая к указателям и возвращающая *lvalue* низлежащего типа.

```
int x = 1;
float y = 2.5;

int* px = &x;
int* py = &y;

*px = 11;
*py = 3.5;
std::cout << *px << '␣' << *py; // 11 3.5
std::cout << x << '␣' << y; // 11 3.5
```

6.5 Арифметика указателей

Как известно, адреса в C++ представляют собой целые числа. Но арифметика с этими числами представляет собой немного другое, чем работа с `int` , например.

Правила арифметики звучат так:

- Указатели одинакового низлежащего типа можно вычитать друг из друга. Результат — количество элементов данного типа, которое поместится в этом промежутке (разница в байтах / `sizeof(T)`).

```
int x, y, z;
std::cout << &z - &x << '\n'; // most likely 2
```

- К указателям можно прибавлять целые числа. Результат — указатель, сдвинутый на `n` шагов размера `sizeof(T)` .

```
int32_t* p = ...;
p + 5; // + 20 bytes
p - 4; // - 16 bytes
p += 10;
p -= 12;
++p;
p--;
```

- Остальные арифметические операции недопустимы.

Отметим, что разность указателей на переменные или сдвиг указателя на переменную с точки зрения стандарта не дает разумного результата.

Разыменование подобных указателей приводит к *UB*.

Для чего на самом деле нужна арифметика указателей, узнаем, когда будем говорить о массивах.

6.6 Указатели и const

Указатели могут быть константными с двух точек зрения.

- *Указатель на константу.* Константным является объект, на который указывают:

```
int x = 0;
const int y = 0;

const int* px = &x; // or int const*
const int* py = &y; // or int const*

std::cin >> *px; // CE: the object is considered immutable
std::cout << *py; // OK: object is readable
```

- *Константный указатель.* Константным является сам указатель:

```
int x;
int y;

int* const px = &x;

std::cin >> *px; // OK
std::cout << *py; // OK

px = &y; // CE: pointer immutable
++px; // CE
```

Константности можно комбинировать:

```
const int* px = ...; // pointer to constant
int const* py = ...; // pointer to constant
int* const pz = ...; // constant pointer
const int* const pa = ....; // constant pointer to constant
int const* const pa = ...; // constant pointer to constant
```

Лайфхак: читайте типы указателей справа налево.
`int const* const` — constant pointer to constant int

6.7 Указатели на указатели

Указатель — это тип, а значит, на него тоже можно создать указатель.

```
int x;
int* px = &x;
int** ppx = &px;
int*** pppx = &ppx;
```

Указатели на указатели тоже дружат с `const` :

```
const float* const** const p = ...;
// constant pointer to pointer to constant pointer to constant float
```

6.8 Нулевой указатель

Попытка чтения неинициализированной переменной приводит к *UB*. То же самое касается и указателей.

```
int* p;
std::cout << p << '␣' << *p; // UB
```

При этом указателям в общем случае нельзя присвоить конкретного начального числового значения:

```
int* p = &x; // Ok
int* q = p; // Ok
int* r = 10; // CE: conversion from int to int* is forbidden
```

Для того, чтобы задать начальное значение "пустого"указателя и отличить корректный указатель от некорректного, вводится понятие *нулевого указателя*.

Любому указателю можно присвоить значение 0, которое соответствует нулевому адресу

Напоминание: по стандарту по нулевому адресу не может находиться ничего, поэтому разыменование подобного указателя приводит к UB.

```
int* p = 0;
const float* pp = 0;
```

К сожалению, подобная возможность иногда приводит к проблемам, так как возникает ассиметрия: целые числа присваивать нельзя, а 0, внезапно, можно.

Правильным с точки зрения современного C++ способом присвоить нулевое значение указателю является литерал `nullptr` .

```
int* p = nullptr;
const float* pp = nullptr;
```

`nullptr` значение специального типа (`std::nullptr_t`), которое неявно приводится к нулевому указателю любого типа.

Правило: для обозначения нулевых указателей используйте только `nullptr` .

6.9 Указатель на void

Напоминание: `void` — специальный тип, обозначающий отсутствие объекта.

Если хочется сохранить адрес ячейки памяти без учета типа объекта, который там может лежать, можно воспользоваться `void*` :

```
int x;
void* p = &x;

std::cout << p; // Ok
std::cin >> *p; // CE: not sure what to write
std::cout << *p; // CE: not sure what to read
```

6.10 "Висячий"указатель

"Висячим"указателем называется указатель, который указывает на объект, область действия которого уже закончилась.

```
int main() {
  int* p = nullptr;
  {
    int x = 0;
    p = &x;
  }
  return *p;
}
```

Чтение или запись данных по такому указателю приводит к *UB*!

Важно: не допускайте провисания указателей!

6.11 Динамическое выделение памяти

До сих пор мы лишь заводили переменные в глобальной области и на стеке.

У хранения данных в глобальной области и на стеке есть ряд ограничений:

1. Они имеют строго фиксированное время жизни. То есть глобальные переменные будут жить до конца программы, а локальные - до конца текущего блока.
2. Как правило, эти области ограничены несколькими мегабайтами. Однако реальные программы часто требуют гораздо больше.

6.11.1 операция(operator) new

Выражения вида `new T` / `new T(...)` / `..new T`. создают в динамической области объект типа T и возвращают указатель на него.

```
int* pa = new int; // without initialization
int* pb = new int(11); // initialization number 11
int* pc = new int{13}; // initialization number 13
```

Полученный указатель используется для доступа к объекту

Объекты в динамической области живут с момента вызова `new` и до окончания работы программы.

Таким образом, можно создавать объекты внутри блоков, которые будут доступны и вне него:

```
int* p = nullptr;
{
  int x = 11;
  p = new int(13);
}
std::cout << x; // CE: x already destroyed
std::cout << *p; // OK
```

Но это ведет к тому, что память постепенно "засоряется".

6.11.2 операция(operator) delete

Память в динамической области можно (и нужно) очистить до завершения программы, как только она перестала быть нужной.

`delete <pointer>` — удаляет объект, возвращает память системе.

```
int* p = new int;
// ...
delete p;
```

Замечание 1: удалять можно только ту память, которая была выделена с помощью `new` . В остальных ситуациях `delete` приводит к *UB*.

Замечание 2: удаление нулевого указателя — корректная операция(operator).

6.11.3 Утечка памяти

При некорректной работе с указателями или несвоевременном очищении динамической памяти может возникать утечка памяти - неконтролируемый рост лишней памяти.

```
int* p = new int;
p = nullptr; // lost pointer to dynamic memory - leak
new float; // memory allocated, pointer not saved
p = new int; // forgot to remove after use - leak
```

Аккуратное использование `new` и указателей и своевременное использование `delete` - залог успешной борьбы с утечками.

6.11.4 Иные методы выделения динамической памяти

В языке C основным способом выделения динамической памяти была функция `malloc`

Ключевые отличия:

1. Возвращает указатель `void*` .
2. Требует явного указания количества байт.
3. Не инициализирует память.

```
void* v = std::malloc(11);
int* p = static_cast<int*>(std::malloc(sizeof(int)));
*p = 0;
```

Для очищения используется функция `free` :

```
std::free(v);
std::free(p);
```

В C++ есть аналог функций `malloc` / `free` — функции `operator new` / `operator delete` :

```
void* v = operator new(11);
int* p = static_cast<int*>(operator new(sizeof(int)));
*p = 0;

operator delete(v);
operator delete(p);
```

Они используются крайне редко и в специфических ситуациях.
Основной способ — операции `new` / `delete` .

7 Ссылки

7.1 Определение

Ссылка - альтернативное имя объекта в памяти. Ссылку можно воспринимать как указатель, который не нужно разыменовывать.

```
int main() {
  int x = 0
  int& rx = x; // rx - alternate name x
  x = 1;
  std::cout << x << ' ' << rx << '\n'; // 1 1
  rx = 2;
  std::cout << x << ' ' << rx << '\n'; // 2 2
  return 0;
}
```

Как следует из определения, ссылка связывается не с переменной, а с областью памяти:

```
int main() {
  int x;
  int array[10];
  int* p = ...;

  int& rx = x; // x reference
  int& ra = array[0]; // reference to the first element of the array
  int& rp = *p; // reference to the memory pointed to by p
  return 0;
}
```

7.2 Правила работы со ссылками

Существуют основные правила работы со ссылками:

1. Ссылки связываются раз и навсегда. Нельзя поменять область памяти, на которую ссылается ссылка.

```
int& rx = x; // rx <=> x
rx = y; // x is written to y, rx is not associated with y
```

2. Из п.1 следует, что ссылки по определению константные.

```
int& const rx = x; //this is not possible (this is assumed automatically)
```

3. Ссылки могут быть связаны только с областью памяти (*lvalue*).

```
int& x = 0; //CE
```

4. Нельзя создать ссылку на `void` .

5. Нельзя создать ссылку на ссылку

6. Ссылка обязана быть проинициализирована (связана) при создании.

```
int& rx; //CE
```

7. Нельзя создавать массивы ссылок.

```
int& array[2]{x, y}; //CE
```

7.3 Пример

```
int x = 0;
int y = 1;

int& rx = x; // rx is associated with x
int& ry; // CE: reference must be bound at initialization
rx = y; // x is written to y, rx still refers to x
ry = y; // CE

int& rz = rx; // rz is associated x
int&& rt = rx; // reference to reference cannot be created

// References can only be associated with a memory area (lvalue)
int& z = 0; // CE
int& t = x + y; // CE
```

7.4 Ссылка на константу

Можно создать ссылку на константу. Такая ссылка предоставляет права только на чтение.

```
int x = 0;
const int cx = 1;

const int& rx = x; // OK
const int& rcx = cx; // OK
int& rx = cx; // CE
```

Исключение: константные ссылки продлевают время жизни временных объектов, то есть могут связываться с *prvalue*.

```
int& x = 0; // CE
const int& cx = 0; // OK
```

Константная ссылка предоставляет "убежище"(место в памяти) для значений, поэтому у них можно брать адрес и ссылаться на них:

```
const int* p = &cx; // OK
const int& rcx = cx; // OK
```

8 Массивы

8.1 Определение

Массив — это последовательность элементов **одного типа**, расположенных **непрерывно в памяти**, к которым имеется **доступ по индексу** через некоторый **уникальный идентификатор**.

`int array[10];`

Важно: выражение, стоящее в квадратных скобках, должно быть положительным константным значением, известным на этапе компиляции!

```
int n;
std::cin >> n;
int array[n]; // According to the C++ Standard, this is not possible.
```

Рассмотрим массив более подробно.

`int array[10];`

- Данный массив хранит 10 **int'ов**
- Доступ к каждому элементу можно получить с помощью имени `array`, через операцию `[]`:

  `array[0], array[2], array[9]; //numbering from 0`
- **Гарантируется**, что все они идут в памяти подряд, без разрывов:

  `&a[i] - &a[j] == i - j;`

8.2 Пример решения простой задачи

Задача: *Вводится 100 чисел. Вывести их в порядке возрастания.*

```
int array[100];
for (int i = 0; i < 100; ++i) {
  std::cin >> array[i];
}
for (int i = 0; i < 100; ++i) {
  int min_idx = 0; // index of minimum
  for (int j = 1; j < 100; ++j) {
    if (array[j] < array[min_idx]) min_idx = j;
  }
  std::cout << array[min_idx] << '␣';
  array[min_idx] = 1000000; // assume values are less than 1000000
}
```

8.3 Операции над массивом

Операции над массивами проще рассмотреть на примерах.

```
int array[10];
array[1]; // index access operation
sizeof(array); // 40: returns the total size in bytes
sizeof(array) / sizeof(int); // 10: amount of elements
&array; // array address ( int(*)[10] )
```

8.4 Инициализация массивов

- Неинициализированный массив

  `int a[10];`

- Инициализация нулями

  `int a[10]();`

  `int b[20]{};`

- Заполнение значениями

  `int a[5]{1, 2, 3}; //1 2 3 0 0 (the rest is filled with zeros)`

  `int b[]{1, 2, 3}; //1 2 3 (size is calculated automatically)`

- Копирование запрещено

  `int b[5] = a; //CE`

8.5 Связь массивов и указателей

В большинстве ситуаций массив автоматически преобразуется в указатель на свой нулевой элемент:

```
int array[10];
std::cout << array; // the address of the null element is displayed
std::cout << array + 5; // fifth element address
```

Верно и обратное — к указателям можно применять `[]`, то есть воспринимать их как массивы:

```
int* p = array;
std::cout << p[2]; // the second element of the array

p += 5;
std::cout << p[2]; // the seventh element of the array
```

Это работает следующим образом:

```
p[2]; // equivalent *(p + 2)
2[p]; // yes, that's the same *(2 + p)
```

Важно понимать, что массив ≠ указатель на первый элемент.

```
int array[10];
int* p = array;

std::cout << sizeof(array) << '␣' << sizeof(p); // 40 8
&array; // has type int(*)[10]
&p; // has type int**
```

Но в большинстве ситуаций массив действительно ведет себя как указатель (неявно приводится к указателю)

Сравнивать массивы с помощью операций `>`, `<`, `==`, ... не имеет смысла, так как реально сравниваются указатели на первые элементы.

```
int a[]{1, 2, 3};
int b[]{1, 2, 3};
a == a; // always true
a == b; // always false
```

8.6 Многомерный массив

Массивы могут иметь несколько размерностей

```
int a[10]; // one-dimensional array
a[1]; // accessing an element
int b[5][20]; // two-dimensional array
b[3][11]; // accessing an element
int c[7][9][3]; // 3D array
c[3][8][0]; // accessing an element
...
```

8.6.1 Инициализация многомерных массивов

Инициализация многомерных массивов может быть произведена различными вариантами:

```
int a[2][3]{1, 2, 3, 4}; // padding by line
// 1 2 3
// 4 0 0

int b[][2]{1, 2, 3, 4}; // the first dimension can be inferred
// 1 2
// 3 4

int c[3][2]{{1, 2}, {3, 4}, {5, 6}}; // 1 2
// 3 4
// 5 6
```

8.6.2 Многомерные массивы и указатели

Как и одномерные массивы, многомерные приводятся к указателю на первый элемент. Первый элемент многомерного массива — массив меньшей размерности.

```
int a[1][2][3];
a[0]; // type int[2][3]

int b[2][3];
b[1][1] <=> *(b[1] + 1) <=> *(*(b + 1) + 1) <=> *(&b[0][0] + 3 + 1);
```

8.7 Правила работы с массивами

1. Нельзя создавать массивы ссылок и массивы функций, но можно создавать массивы указателей и массивы указателей на функции.

```
int& a[10]; // CE
int b[20](int); // CE

int* c[30]; // OK (array of pointers)
int (*d[40])(int); // OK (array of function pointers)
```

2. Нельзя создать массив с неизвестным числом элементов, но можно его объявить.

```
int a[]; // CE (definition)
int b[] = {1, 2, 3}; // OK: int[3]
extern int c[]; // OK (announcement)
```

3. При сравнении массивов сравниваются адреса нулевых элементов (не значения!). Это значит, что результат сравнения на равенство для разных массивов всегда `false` .

```
int a[3]{1, 2, 3};
int b[3]{1, 2, 3};
std::cout << (a == b) << '␣' << (a == a); // 0 1
```

4. Массивы нельзя присваивать друг другу (исключение — строки при инициализации).

```
int a[3];
int b[3]{1, 2, 3};
a = b; // CE
```

5. Лайфхак: если массив — это поле структуры или класса, то чудесным образом присваивание начинает работать

```
struct S {
  int array[3];
};

S c{1, 2, 3};
S d{4, 5, 6};
c = d; // OK (c.array == {4, 5, 6})
```

8.8 Динамические массивы

Создаются с помощью оператора `new[]` , который возвращает указатель на нулевой элемент массива.

`int* array = new int[10];`

Так как результат — указатель на элемент (не на массив!), его нельзя использовать в предыдущих контекстах

```
sizeof(new int[10]) == sizeof(int*); // 8
sizeof(*array) == sizeof(int); // true
```

Размер динамического массива, в отличие от стекового, **не обязан** быть константой времени компиляции!

```
int n = 10;
// Work with n ...
int a[n]{1, 2, 3}; // prohibited by the standard, but compilers allow
int* b = new int[n]{1, 2, 3}; // OK
```

И не забывайте убирать за собой!

Важно: память, выделенную с помощью `new[]` , необходимо очищать с помощью `delete[]`

```
int* array = new int[10];
// ...
delete[] array;
```

8.9 Многомерные динамические массивы

Многомерный динамический массив можно моделировать массивом массивов:

```
// memory allocation
int** array = new int*[n];
for (int i = 0; i < n; ++i) {
  array[i] = new int[m];
}

// filling
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < m; ++j) {
    std::cin >> array[i][j];
  }
}

// cleansing
for (int i = 0; i < n; ++i) {
  delete[] array[i];
}
delete[] array;
```

9 Объявления указателей, ссылок и массивов*

9.1 Общие принципы

Модификатор указателя, ссылки, массива распространяется только на ближайшую переменную:

```
int* a, b, c; // int*, int, int
int x, &y, z; // int, int&, int
int p, q, r[10]; // int, int, int[10]
```

В том числе поэтому многими кодстайлами (в том числе нашим) рекомендуется писать по одному объявлению на строку:

```
int* a;
int b;
int c;
```

9.2 Указатели и ссылки на массивы

Чтобы задать указатель/ссылку на массив необходимо "привязать"модификатор `*` / `&` к имени переменной:

```
int (*a)[10]; // pointer to array
int (&a)[10]; // array reference
```

Это правило распространяется и на объявления "в одну строку":

```
int a, *b[10], (&c)[20];
```

9.3 Сложные комбинации

Чтобы читать сложные типы, состоящие из комбинации ссылок, указателей и массивов, прибегнем к методу улитки:

- Находим имя переменной
- Двигаемся направо до первой закрывающейся скобки
- Двигаемся налево до первой открывающейся скобки
- Читаем подряд все встретившиеся типы и модификаторы

Так, например, если нас просят задать переменную типа "ссылка на массив из 15 указателей на массивы `int` 'ов размера 10 то получим
`int (*(&a)[15])[10];`

10 С-строки

10.1 Строковые литералы

Строковыми литералами называются объекты, которые находятся внутри ". Немного о строковых литералах:

1. Представляют собой массивы символов с 0 на конце
2. Могут быть скопированы при инициализации (исключение из общего правила)
3. Сравнение происходит непоэлементно, а сравниваются адреса первых элементов как с массивами. Но у строковых литералов есть особый способ работы:
    - Массив, с которым связан строковый литерал, лежит в статической (глобальной) области памяти (в таблице строковых литералов).
    - Компилятор, анализируя исходный код, помещает каждый попавшийся строковый литерал в отдельный буфер (отдельная строка таблицы).
    - Как правило, одинаковые литералы ссылаются на одну и ту же область памяти, поэтому их сравнение путем сравнения указателей может давать верный результат (но это не гарантировано стандартом!).

10.2 Строки

- Строка — часть массива элементов char, ограниченная символом 0
- Правила работы со строками такие же, как и с обычными массивами