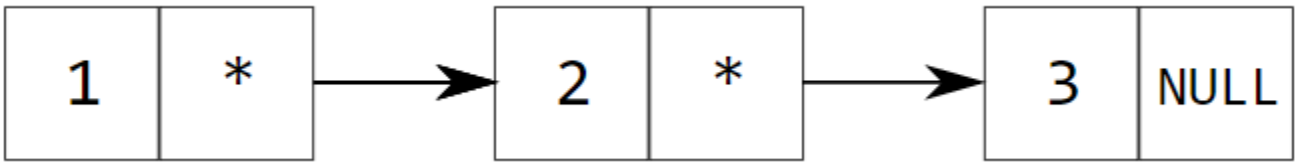
Связный список

Почти в любой программе есть необходимость хранить данные в каком-то контейнере, коллекции. Обычно нам необходимо производить с ней следующие операции:

- Доступ к элементам по индексу
- Добавление элемента:
  - в начало;
  - в конец;
  - на произвольную позицию по индексу;
- Сортировку в каком-то порядке.

Пока что мы работали только с одним типом контейнеров – массивом, и в C он единственный встроенный в сам язык; остальные нам нужно определять самостоятельно, или пользоваться сторонними библиотеками.

Связный список — это структура данных, один из возможных контейнеров для данных одинакового типа. Список состоит из цепочки элементов, каждый из которых хранит значение и адрес следующего элемента. Например, список (1, 2, 3) можно изобразить так:



За последним элементом списка ничего нет, поэтому его *указатель на следующий элемент* равен `NULL`.

Конечно, в связном списке можно хранить данные любых типов, но для примера мы будем работать со списками чисел типа `int64_t`.

Практическая значимость

Зачем нам связные списки, если есть массивы? Разные контейнеры делают одни операции лёгкими и быстрыми, а другие — медленными. Например, в массив неудобно добавлять элементы из-за его фиксированного размера, но можно быстро обратиться к уже существующему по индексу. В связный список, наоборот, удобно добавлять элементы в любое место, но доступ по индексу сложнее — нужно просмотреть весь список с самого начала.

Связные списки используются повсеместно: как основа для других структур данных (например, Хеш-таблица), в системном программировании (см. реализацию в ядре Linux), как основная структура данных в функциональных языках...

Кодирование связного списка

Пару из значения и адреса следующего элемента можно закодировать так:

```
// list это тип элементов списка


struct list {

   int64_t value;

   struct list* next;

};
```

Список задаётся указателем на его первый элемент. Если список пустой, то этот указатель устанавливается в NULL.

```
struct list* mylist = NULL ; // этот список пустой
```

Непустой список задаётся ненулевым указателем на первый элемент. Вернёмся к списку (1, 2, 3):

В коде его можно представить так:

```
struct list {

   int64_t value;

   struct list* next;

};


...


struct list x3 = { 3, NULL };

struct list x2 = { 2, &x3 };

struct list x1 = { 1, &x2 };


struct list* mylist = &x1; // это список (1,2,3)
```

Началом списка будет являться указатель `mylist` на `x1`.

Вот полный пример кода для определения списка и его вывода на экран.

```
#include <inttypes.h>

#include <stdio.h>


struct list {

  int64_t value;

  struct list* next;

};



void list_print(const struct list* l) {

  while (l) {

    printf("%" PRId64 " ", l->value);

    l = l->next;

  }

}


int main() {


  struct list x3 = { 3, NULL };

  struct list x2 = { 2, &x3 };

  struct list x1 = { 1, &x2 };


  list_print(&x1);
```

```
  return 0;
}
```

Создание элемента списка

Создавать связные списки так, как мы делали в примере, крайне неудобно:

```
void f() {

  struct list x3 = { 3, NULL };
  struct list x2 = { 2, &x3 };
  struct list x1 = { 1, &x2 };

}
```

Каждый элемент надо выделять на стеке отдельно. Когда функция `f` завершится, эти элементы уничтожатся, а нам не всегда этого бы хотелось.

Поэтому для начала научимся выделять элементы в куче с помощью `malloc`. Для этого определите функцию `node_create`, которая примет целочисленное значение и вернёт указатель на список из ровно одного элемента, выделенный в куче.

1task – program

Теперь нужно научиться добавлять элементы в список. Сначала будем добавлять элемент к началу списка, это проще и быстрее.

```
struct list {
   int64_t value;
   struct list* next;
};


// список
```

```
struct list* mylist = ... ;
```

Вспомним, что список задаётся указателем на его первый элемент. Чтобы добавить элемент в начало списка, нужно:

1. создать новый элемент;
2. прицепить старый список к нему;
3. перенаправить указатель на первый элемент так, чтобы он указывал на новое начало списка.

Чтобы иметь возможность изменить указатель на первый элемент, мы должны передать в функцию его (указателя) адрес, то есть *указатель на указатель* на первый элемент списка.

Определите функцию, которая добавит один элемент в начало списка.

2task – program

Посчитаем длину списка с помощью функции `list_length`. Для этого нужно пройтись по нему с начала и до конца, считая элементы. Напомним определение элемента списка:

```
struct list {
    int64_t value;
    struct list* next;
};
```

Чем длиннее список, тем дольше считается его длина. Время подсчёта длины зависит от длины списка линейно: если размер увеличивается в X раз, то и длина считается в X раз дольше. Напротив, для массива размер всегда фиксирован и потому считается моментально.

Обратите внимание, что подсчёт длины никак не изменяет список. Поэтому мы можем применять операцию "подсчитать длину" даже к неизменяемым спискам. Чтобы этого достичь правильно расставьте модификаторы в сигнатуре функции `list_length`.

Sample Input:

```
10 2 8 3
```

Sample Output:

```
4
```

3task – program

Теперь определим функцию `list_destroy`, которая освободит всю память, выделенную под элементы списка.

Вот пример неправильной реализации `list_destroy`. Почему?

```
void list_destroy( struct list* list ) {

   while (list) {

      free( list );

      list = list -> next;

   }

}
```

Проблема в том, что мы освобождаем память под элемент списка, а затем к ней обращаемся в выражении `list->next`. Стандарт языка напрямую запрещает обращаться к освобождённой памяти.

В самом деле, даже на практике мы можем представить ситуацию, когда между освобождением памяти и обращением к ней происходило выделение памяти в куче. Тогда память по адресу `list` может принадлежать уже другому блоку памяти и иметь другие значения.

Sample Input:

```
5 4 3 2 1
```

Sample Output:

```
5 4 3 2 1
```

4task – program

Полезно уметь находить последний элемент списка. Напишите функцию `list_last`, которая вернёт адрес последнего элемента списка.

Sample Input:

```
6 12 0893 1 2
```

Sample Output:

```
2
```

5task – program

Теперь на основе `list_last` мы можем реализовать функцию `list_add_back`, дописывающую элемент в конец списка. Не забудьте про пустые списки!

Вы можете вызывать функции:

- `struct list* list_last( struct list * list );`
- `void list_add_front( struct list** old, int64_t value );`
- `struct list* node_create( int64_t value );`

Sample Input:

```
1 2 3 4 5
```

Sample Output:

```
1 2 3 4 5 9 8 7
```

6task – program

Напишите функцию, проходящуюся по всему списку и считающую сумму всех его элементов. Помните, что она должна работать для неизменяемых списков.

Sample Input:

```
34 2 6
```

Sample Output:

```
42
```

7task – program

В массивах мы можем легко добраться до любого элемента, так как они лежат в памяти последовательно. Достаточно к адресу начала массива прибавить правильное смещение. Со связными списками сложнее: нужно пройтись по списку с начала, считая элементы.

Напишите функцию `list_at`, возвращающую экземпляр `maybe_int64`. Структуру мы берём [из предыдущего урока]:

```
struct maybe_int64 {

  bool valid;

  int64_t value;

};
```

```
struct maybe_int64 some_int64( int64_t i ) {
```

```
  return (struct maybe_int64) { .value = i, .valid = true };

}
```

```
const struct maybe_int64 none_int64 = { 0 };
```

Тесты для этого задания принимают сначала индекс в списке, затем список.

Sample Input:

```
3 0 9 8 7 6
```

Sample Output:

```
Some 7
```

8task – program

Используя уже реализованные функции не составит труда сделать функцию `list_reverse`, создающую перевернутую копию списка. Не забудьте отразить в коде, что она не изменяет сам список.

Вы можете пользоваться уже определённой функцией `list_add_front`.

Sample Input:

```
9 2 37 4 2
```

Sample Output:

```
2 4 37 2 9
```

9task – program

Наконец, научимся считывать список. Для этого сначала научимся читать одно число без гарантии успеха, то есть напишем функцию `struct maybe_int64 maybe_read_int64()`.
Структуру мы берём [из предыдущего урока]:

```
struct maybe_int64 {

  bool valid;

  int64_t value;

};
```

```
struct maybe_int64 some_int64( int64_t i ) {

  return (struct maybe_int64) { .value = i, .valid = true };

}


const struct maybe_int64 none_int64 = { 0 };
```

Функция `maybe_read_int64` будет вызывать `scanf` и возвращать экземпляр `struct maybe_int64`.

Помните, что вызов `scanf` с правильным спецификатором ввода -- это *попытка* прочитать число, которая не обязательно завершится успехом. Возвращаемое значение `scanf` показывает, получилось ли прочитать число или нет. Например, число не получится прочитать если ввод уже закончился.

Советуем прочитать: [справочную страницу для `scanf`].

Sample Input:

```
1
```

Sample Output:

```
Some 1
```

10task – program

И, наконец, реализуем чтение списка со входа. Читать числа будем пока это возможно.

Например, если на вход подаются числа 1, 3, 5, 8, нужно вернуть из функции список с этими числами в том же порядке.

Вы можете использовать функции `node_create` и `maybe_read_int64` из предыдущего шага (она и реализует чтение "пока возможно").

11task – program