Структурирование программы. Представление данных в программе

Курс «Технология программирования»

Кафедра управления и информатики НИУ «МЭИ»

Осень 2016 г.

Что такое указатель?

Все данные хранятся в памяти.

Переменная — именованная область памяти.

Можно считывать и записывать значения переменных.

Память — массив нумерованных ячеек-байт.

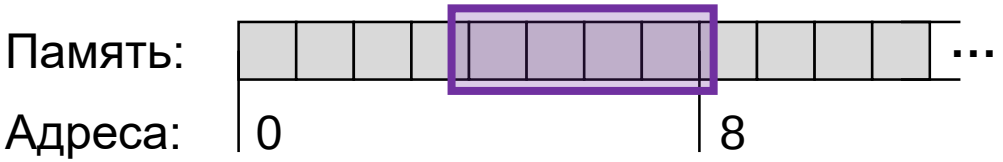
У переменных есть адреса.

Имея адрес ячейки, можно работать с её значением.

Номер называется адресом.

1) Как получить адрес?
2) Как хранить адрес?

Как?



Указатели

- Это переменные, содержащие адрес памяти.
- Указатель pointer:
  - pointer — адрес;
  - *pointer — значение по адресу (разыменование).
- Переменная mean (не указатель):
  - mean — значение;
  - &mean — адрес (взятие адреса).
- **nullptr** == NULL == 0:
  - нулевой указатель;
  - значение по нему получить нельзя.

Тип данных «указатель»

Звездочка перед переменной.

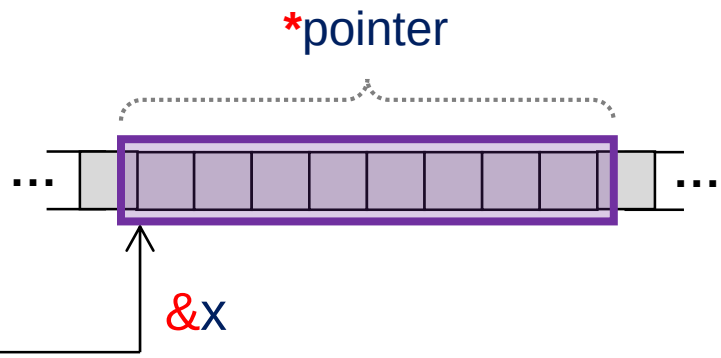
```
double x = 42;
double * unknown;
double* pointer = &x;
```

До присваивания значения указывает неизвестно, куда.

Указатель на **double**:

- может хранить адрес **double**
  - и только **double**;
- разыменование дает **double**.

```
x = *pointer;
```



Неизменяемость и указатели

Что неизменяемо?

- всё изменяемо: **int * p;**
  ***p**ointer = 0;
  pointer = **nullptr**;
- адрес: **int * const cp;**
  ***c**onst_**p**ointer = 0;
  ~~**c**onst_**p**ointer = **nullptr**;~~
- значение: **const int * pc;**
  ~~***p**ointer_to_**c**onst = 0;~~
  pointer_to_**c**onst = **nullptr**;
- и то, и другое: **const int * const cpc;**
  ~~***c**onst_**p**ointer_to_**c**onst = 0;~~
  ~~**c**onst_**p**ointer_to_**c**onst = **nullptr**;~~

Псевдонимы типов

- Способ дать типу новое имя:

```
typedef unsigned char Byte;
using Byte = unsigned char;   // То же для C++11 и выше.
```

- Используется для сокращения и упрощения записи сложных типов:

```
using Document = std::vector< std::vector< std::string > >;
```

- …или для декомпозиции записи:

```
using Word = std::string;
using Sentence = std::vector<Word>;
using Document = std::vector<Sentence>;
```

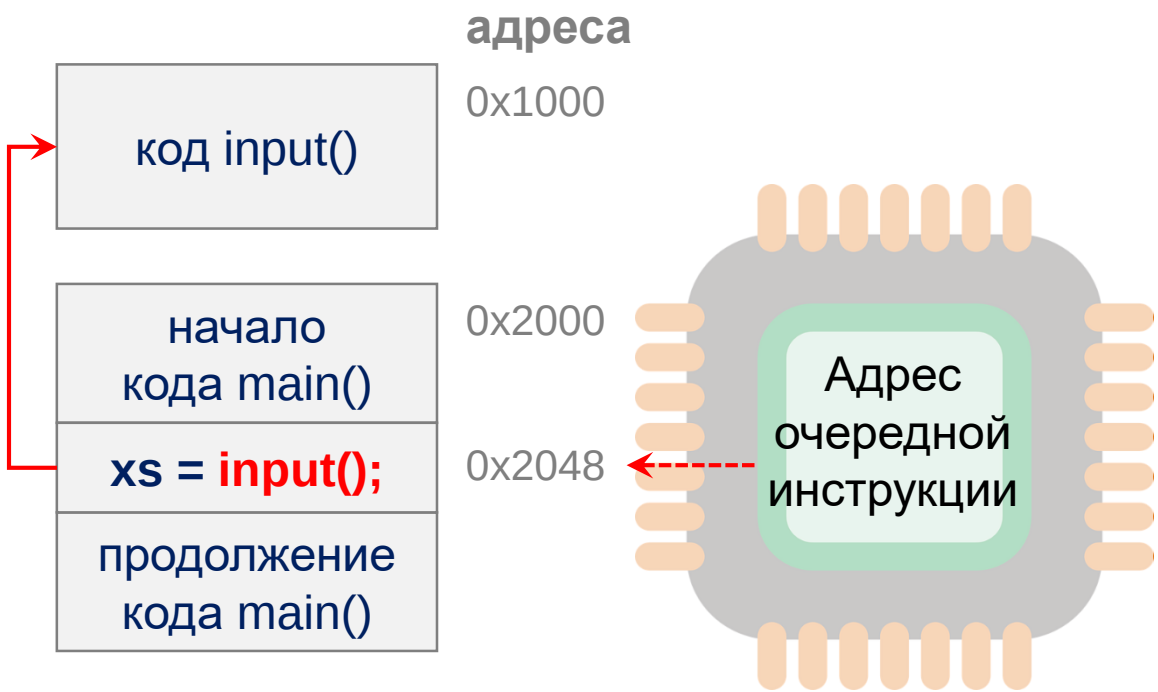
- Когда уже написан код, использующий псевдоним, можно поменять тип правкой всего одной строки.

Указатели на функции

- Код любой функции хранится в памяти.
- Суть вызова функции — передача управления её коду.
- **Вывод:** зная адрес кода функции, можно её вызвать.

Инструкция вызова функции:

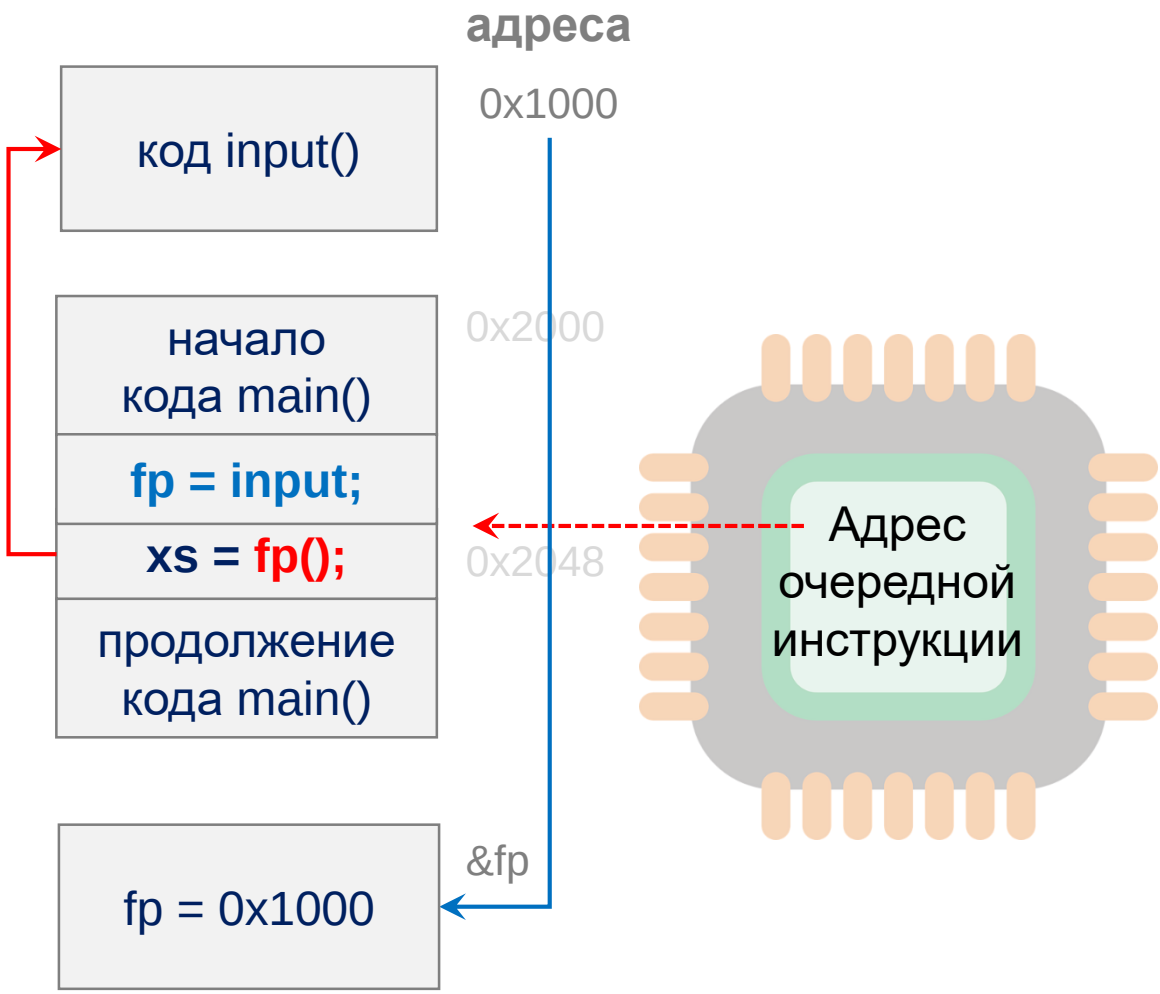
продолжить выполнение с адреса начала кода input()



Вызов функции по указателю

Инструкция косвенного вызова:

продолжить выполнение с адреса, хранящегося в переменной fp.



Пример: произвольный фильтр

```
using Predicate = bool(double);

vector<double> filter(
    const vector<double> data, Predicate satisfies)
{
    vector<double> result;
    for (const auto& item : data) {
        if (satisfies(item)) {
            result.push_back(item);
        }
    }
    return result;
}
```

Тип данных:
указатель на функцию,
которая принимает **double,**
а возвращает **bool**.

Передается указатель
на любую функцию-условие
для фильтрации

(на т. н. функцию обратного
вызова — **callback**).

Функция по указателю
вызывается обычным образом.

Пример: фильтрация температур

```
bool is_valid_kelvins(double temperature) {
    return temperature >= -273.15;
}

vector<double> temperatures = input_temperatures();

auto only_valid = filter(temperatures, is_valid_kelvins);
```

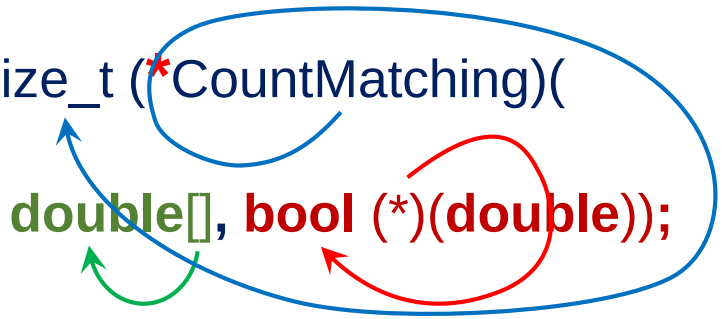
Можно динамически выбирать проверку:

```
Predicate check = are_we_using_celsius ?
        is_valid_centigrade : is_valid_kelvins;

auto only_valid = filter(temperatures, check);
```

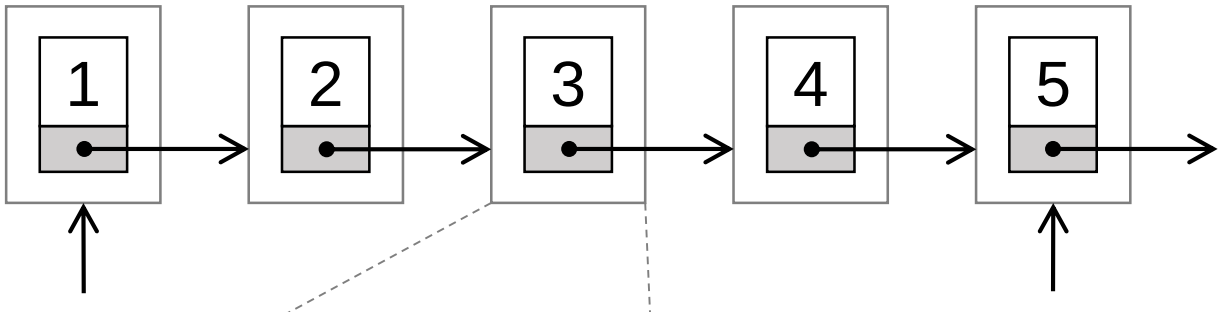
Правило спирали/улитки

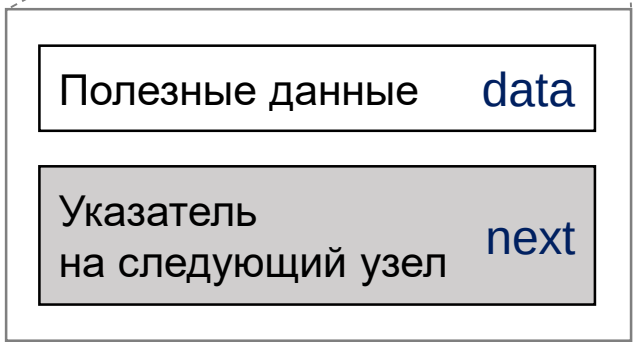
- Запись в форме **typedef** сложнее:
  - **typedef bool** (*Predicate)(**double**);
  - **typedef** size_t (*CountMatching)(



    - Функция, принимающая массив **double** и функцию, принимающую число и возвращающую **bool,** и возвращающая количество.
- Такой код на практике встречается в библиотеках.
- «Правило улитки»: определения читаются «по спирали» по часовой стрелке, начиная с самого вложенного.

Односвязный список (singly-linked list)





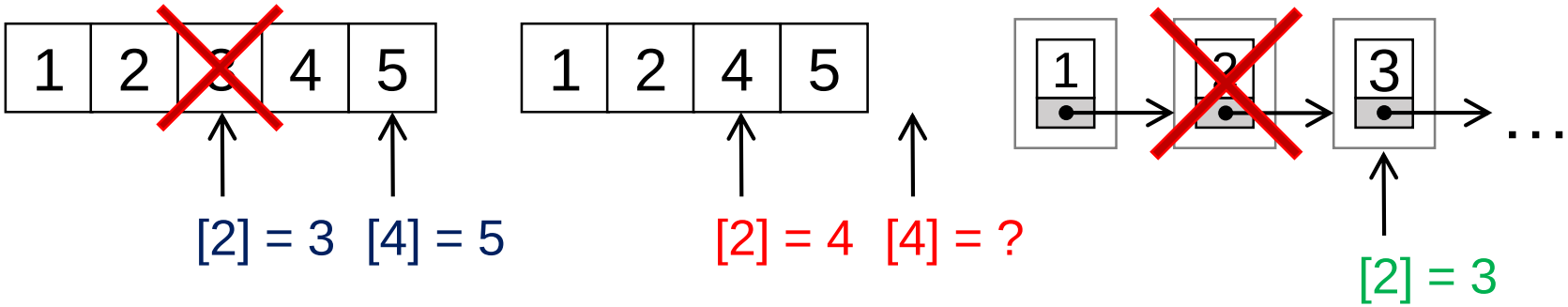
Элемент (узел, node)

```
struct Node
{
    double data;
    Node*  next;
};
```

```
struct List
{
    Node*  first;
    Node*  last;
    size_t  size;
};
```

Зачем нужен связанный список?

- Когда в связанном списке удаляется или добавляется элемент, прочие элементы не нужно перемещать в памяти.
- После любых операций над связанным списком адреса (сохранившихся) элементов не меняются.

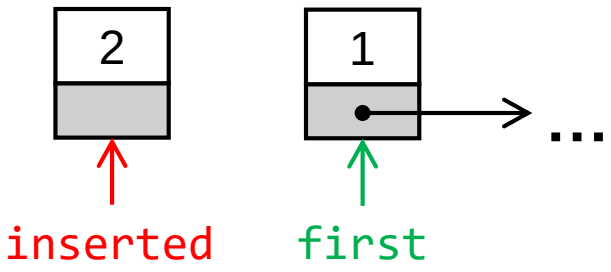


- Для доступа по индексу нужно обойти список — это медленно.
- Список удобен для случаев, когда:
  - нежелательно или нельзя копировать или смещать элементы;
  - скорость доступа по индексу несущественна.

Вставка элемента: а) в начало списка

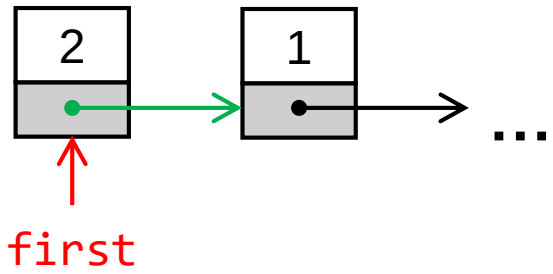
Было:

список и новый узел



Стало:

список с новым узлом в начале



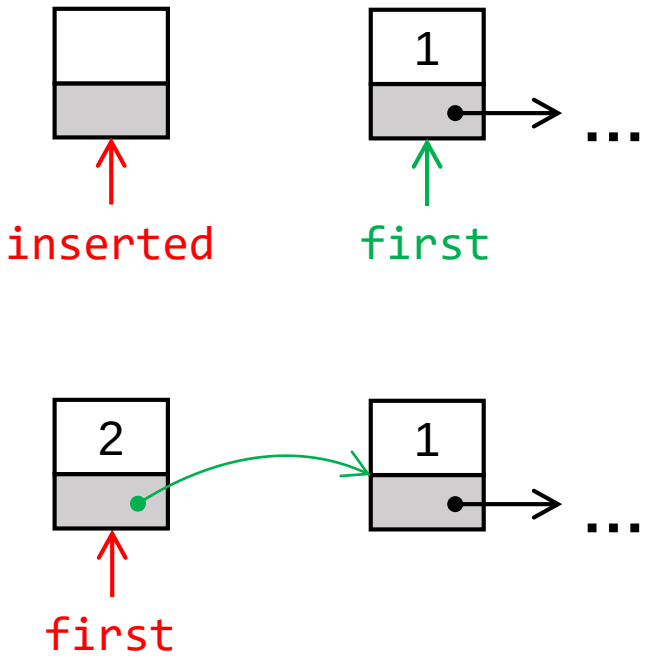

```
inserted->next = first;
first = inserted;
```

`pointer->field` — это `(*pointer).field`

- Один цвет — один адрес (значение указателя), кроме черных и серых (которые не важны).

Реализация вставки в список

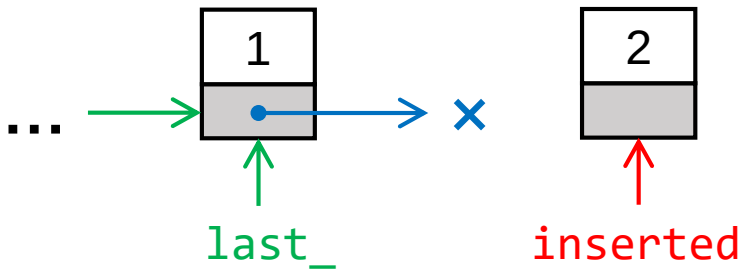
```
void list_push_front(List& list, double value)
{
    Node* inserted = new Node;
    inserted->value = value;
    inserted->next = list.first;
    list.first = inserted;
    if (list.last == nullptr) {
        list.last = inserted;
    }
    list.size++;
}
```



Вставка элемента: б) в конец списка

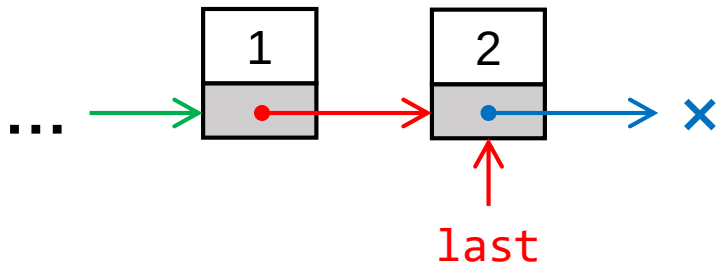
Было:

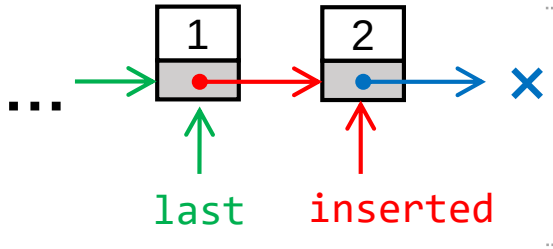
список и новый узел



Стало:

список с новым узлом в конце

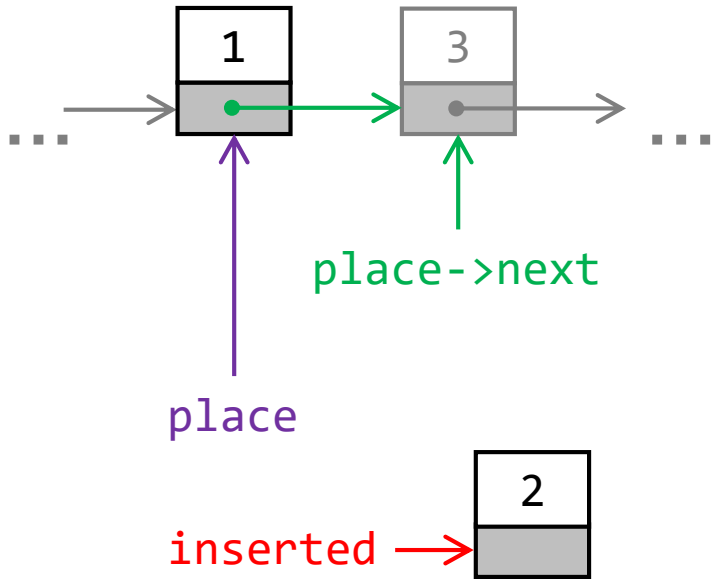




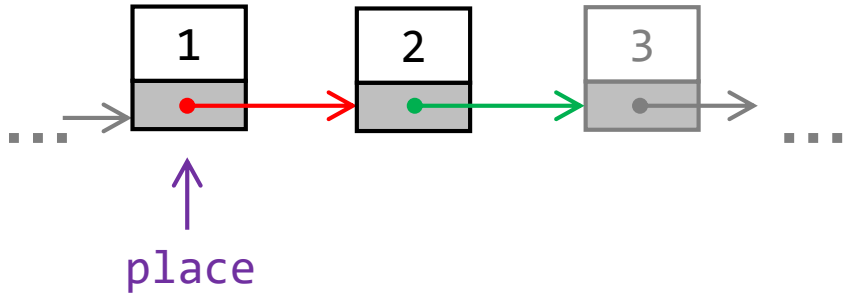

```
inserted->next = nullptr;
last->next = inserted;
last = inserted;
```

Вставка элемента: в) в произвольное место

Было:
список, новый узел
и указатель на узел в списке



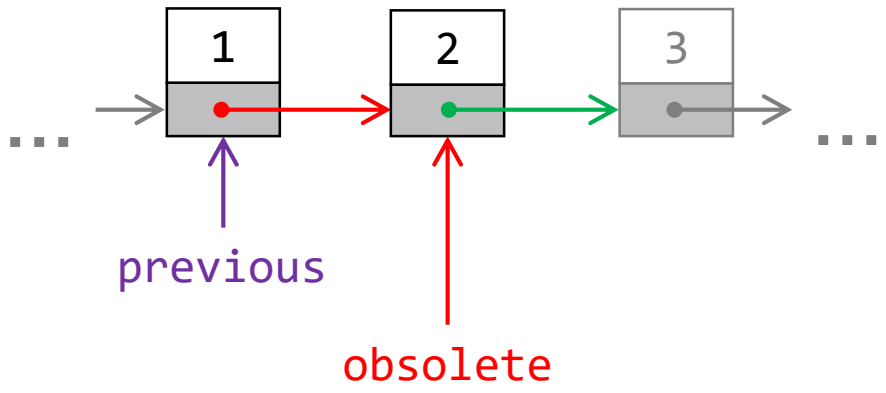
Стало:
список с новым узлом в конце



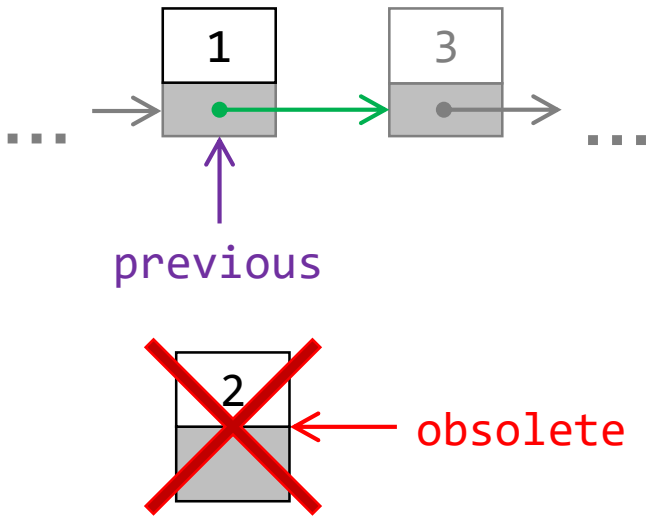

```
inserted->next = place->next;
place->next = inserted;
```

Удаление элемента

Было:
список и узел к удалению



Стало:
список без указанного узла

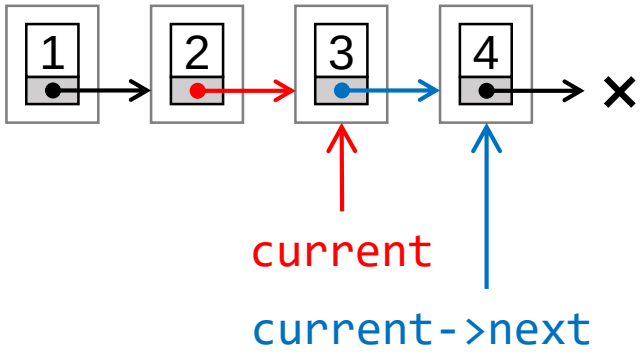



```
previos->next = obsolete->next;
delete obsolete;
```

Проход по списку (enumeration)

- Размер списка неизвестен:

```
Node* current = list.first;
while (current)
     current = current->next;
```



- До элемента № `index`:

```
Node* current = list.first;
for (i = 0; i < index; ++i)
     current = current->next;
```

- Интервал [`first`; `last`) или любой другой:

```
Node* current = list.first;
while (current != list.last)
   current = current->next;
```

  - Можно проверять `current`:

```
while (current && current != list.last) {
    …
}
```

Создание и очистка списка

```
void list_clear(List& list)
{
    Node* current = list.first;
    while (current) {
        Node* victim = current;
        current = current->next;
        delete victim;
    }
    list.first = list.last = nullptr;
    list.size = 0;
}
```

Изначально список пуст.

```
List list_create()
{
    List list;
    list.first = nullptr;
    list.last = nullptr;
    list.size = 0;
    return list;
}
```

При любых операциях не должно оставаться:

Корректность операций над связанным списком

- Операции должны поддерживать *согласованное состояние (consistent state):*
  - на первый элемент указывает только first;
  - last указывает на последний элемент;
  - size равно количеству элементов;
  - все элементы указывают на следующий,
    - нет «колец»;
    - последний элемент хранит указатель на **nullptr**.
- Не должно оставаться указателей на более не используемую память *(«висячих» указателей — dangling pointers).*
- Не должно возникать выделенной памяти, на которую нет указателей *(утечек памяти — memory leaks).*

Литература к лекции

- Programming: Principles and Practices Using C++:
  - пункт 27.2.5 — указатели на функции;
  - раздел 20.4 — связанный список.
- C++ Primer:
  - глава 2, раздел 2.3 — указатели и ссылки.
- *David Anderson.* The “Clockwise/Spiral” Rule (http://c-faq.com/decl/spiral.anderson.html)