Семинар #6: Динамический массив.

Разные варианты массивов в языке C

Статический массив

Статический массив – это массив, размер которого фиксирован. В языке C такой массив создается так:

```
int a[3] = {10, 20, 30};
```

У такого статического массива есть 2 проблемы:

- Нельзя поменять размер, то есть нельзя добавить или удалить элемент.
- Он выделяется на стеке и его максимальный размер сильно ограничен.

Массив на стеке с переменным размером, но фиксированной вместимостью

Первую проблему можно частично решить, если создать массив больше, чем нужно на данный момент:

```
int a[100] = {10, 20, 30};
size_t size = 3;
```

В этом примере мы создали статический массив, который может хранить 100 элементов, но в данный момент используем только первые 3 элемента. Чтобы помнить, сколько элементов используется в данный момент мы завели переменную `size`. Введём следующие определения:

- *Размер массива (англ. size)* – количество элементов массива, которые доступны для использования.
- *Вместимость массива (англ. capacity)* – количество элементов, под которые в массиве выделена память.

То есть, для массива из примера выше размер равен 3, а вместимость равна 100. В такой массив мы можем добавлять элементы, но только до тех пор пока размер меньше, чем вместимость. Например, добавить новый элемент в конец массива `a` можно так (но только если `size < 100`):

```
a[size] = 60;
size += 1;
```

Несмотря на то, что в такой массив можно добавлять и удалять элементы, у такого подхода также есть недостатки. Вместимость не может меняться во время выполнения программы. Её нужно указать заранее. Если указать слишком маленькую вместимость, то этого может не хватить, а если указать слишком большую, то будет напрасно потрачено слишком много памяти. К тому же этот массив всё так же создаётся на стеке, поэтому его размер ограничен.

Выделение динамического массива в куче

Динамический массив – это массив, размер и вместимость которого может меняться во время выполнения программы. Такой массив, можно создать, выделив память в куче:

```
int* p = (int*)malloc(sizeof(int) * 3);
```

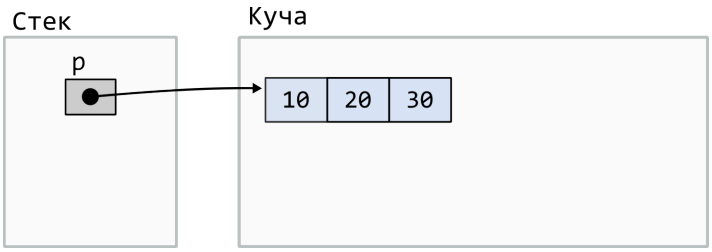
Указатель `p` указывает на массив из трёх элементов, созданный в куче. После того, как такой массив был создан, можно изменить его размер. Для этого нужно сделать следующее:

1. Выделить в куче ещё один участок памяти под новый массив большего размера.
2. Скопировать данные из старого массива в новый.
3. Освободить память старого массива.

Процесс увеличения размера массива, созданного в куче

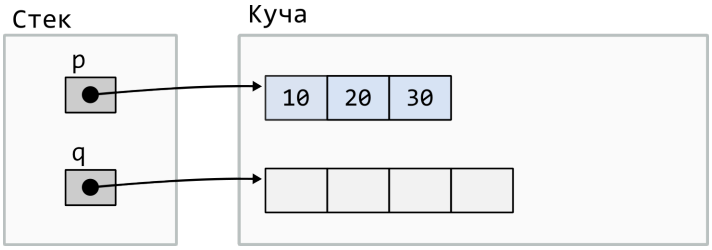
1) Исходное положение

```
int* p = (int*)malloc(sizeof(int)*3);
p[0] = 10;
p[1] = 20;
p[3] = 30;
```



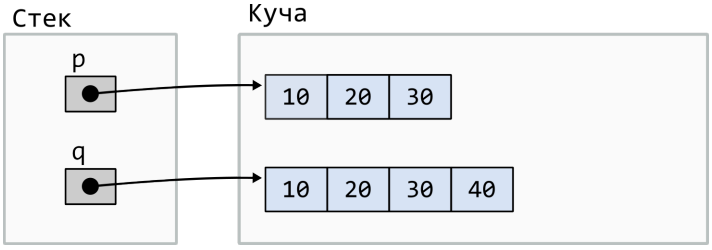
2) Выделяем новый массив большей памяти

```
int* q = (int*)malloc(sizeof(int)*4);
```



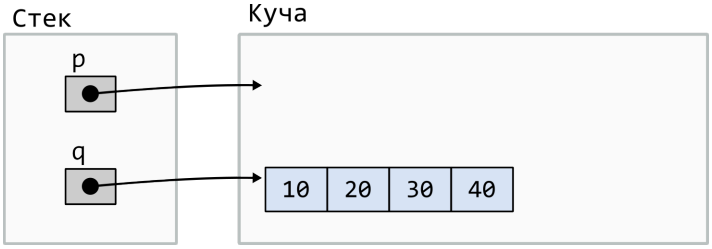
3) Копируем элементы из старого массива
   и добавляем новый элемент

```
for (size_t i = 0; i < 3; ++i)
    q[i] = p[i];
q[3] = 40;
```



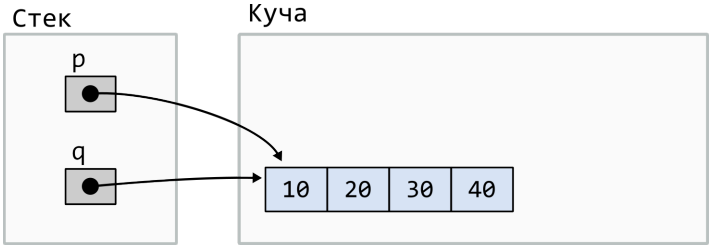
4) Освобождаем память старого массива

```
free(p);
```



5) Изменяем значение старого указателя

```
p = q;
```



Такой подход позволяет создавать массивы с изменяемым размером, не ограниченным размером стека. Однако, и этого подхода есть недостатки:

- Медленное добавление элементов. Для того, чтобы добавить один элемент в массив, нам пришлось вызвать `malloc`, а также скопировать весь массив.
- Придётся делать все операции по выделению/освобождению памяти и копированию массива каждый раз, когда нужно изменить размер. Это неудобно.

Эти проблемы мы решим при написании своего динамического массива.

Создаём свой динамический массив

В отличии от многих других языков, в языке C нет удобного динамического массива, поэтому нам придётся написать свой. Попробуем написать наш массив так, чтобы он удовлетворял следующим требованиям:

1. В массив должно быть возможным добавление и удаление элементов. Его размер может меняться во время выполнения программы.
2. Размер массива не должен быть ограничен размером стека.
3. Массив должен быстро работать. Добавление и удаление элементов в конец массива должно работать за $O(1)$ в среднем.
4. Массив не должен занимать слишком много памяти. А именно, общее количество выделенной для массива памяти должно быть не более чем в 2 раза превышать суммарный размер всех элементов массива.
5. С нашим массивом должно быть удобно работать.
6. Тип хранимого элемента массива должен быть настраиваемым.

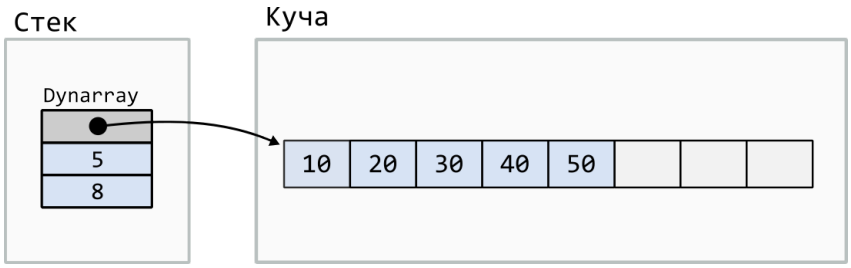
Структура для динамического массива будет выглядить следующим образом:

```
struct dynarray
{
    int* data;
    size_t size;
    size_t capacity;
};
typedef struct dynarray Dynarray;
```

Поля этой структуры:

- `data` – указатель на элементы массива, которые выделяются в куче.
- `size` – текущий размер массива. Столько элементов содержится в массиве.
- `capacity` - текущая вместимость массива. Под столько элементов в массиве выделена память. В отличии от статического массива эта величина может меняться. Количество памяти, выделенной в куче, будет равна `capacity * sizeof(int)`.

В памяти динамический массив с размером 5 и вместимостью 8 будет выглядеть следующим образом:



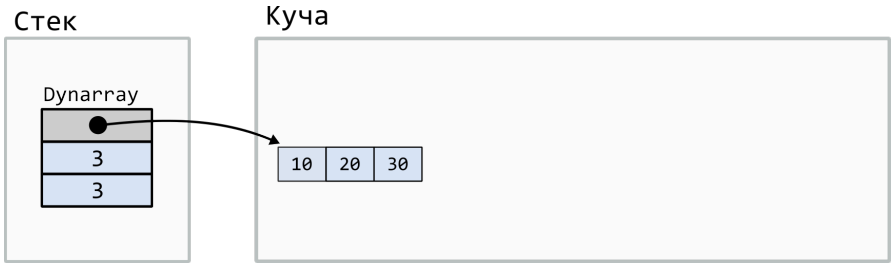
Напишем следующие функции для работы с нашим массивом:

```
void init(Dynarray* pd, size_t n)               // Задаёт массив из n нулевых элементов
int  get(const Dynarray* pd, size_t i)          // Получает значение i - го элемента
void set(Dynarray* pd, size_t i, int value)     // Задаёт значение i - го элемента
void reserve(Dynarray* pd, size_t new_capacity) // Увеличивает вместимость массива
void push_back(Dynarray* pd, int value);        // Вставляет элемент в конец массива
void print(const Dynarray* pd)                  // Печатает массив на экран
void destroy(Dynarray* pd)                      // Уничтожает наш массив
```

Процесс увеличения размера динамического массива

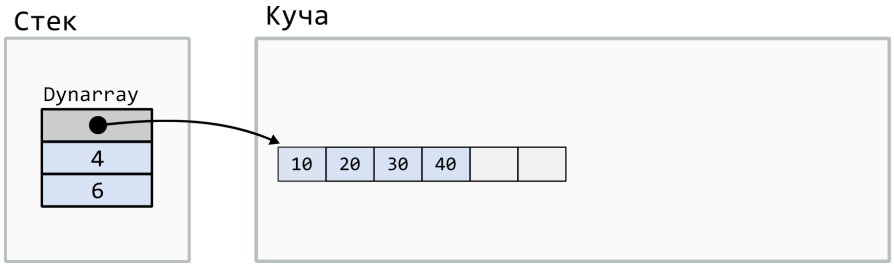
1) Исходное положение

```
Dynarray a;
init(&a, 3);
set(&a, 0, 10);
set(&a, 0, 20);
set(&a, 0, 30);
```



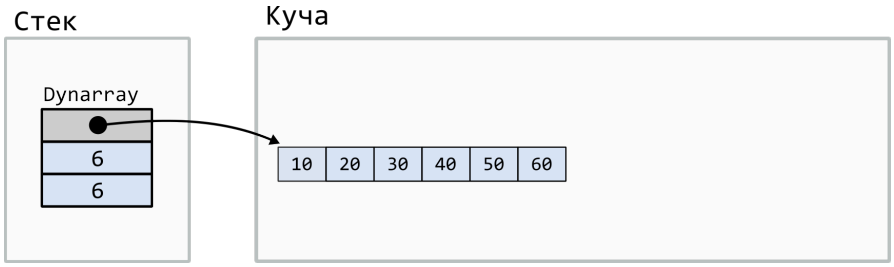
2) Добавляем один элемент,
увеличивая размер массива,
созданного в куче

```
push_back(&a, 40);
```



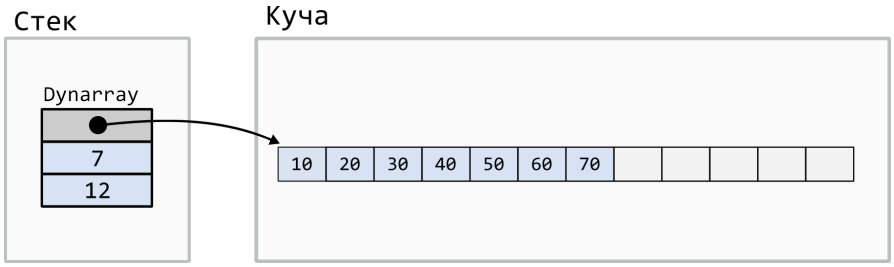
3) Добавляем ещё два элемента, на
этот раз перевыделять память
не нужно

```
push_back(&a, 50);
push_back(&a, 60);
```



4) Добавляем ещё один элемент,
увеличивая размер массива,
созданного в куче

```
push_back(&a, 70);
```



Исходный код нашего динамического массива

Код также находится в файле `code/0dynarray/dynarray.c`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
// Error checked malloc
void* ecmalloc(size_t n)
{
    void* p = malloc(n);
    if (p == NULL)
    {
        fprintf(stderr, "Memory allocation error.\n");
        exit(1);
    }
    return p;
}

struct dynarray
{
    int* data;
    size_t size;
    size_t capacity;
};
typedef struct dynarray Dynarray;

void clear(Dynarray* pd)
{
    for (size_t i = 0; i < pd->size; ++i)
        pd->data[i] = 0;
}

void init(Dynarray* pd, size_t initial_size)
{
    pd->size = initial_size;
    pd->capacity = initial_size;
    if (pd->size == 0)
        pd->data = NULL;
    else
        pd->data = (int*)ecmalloc(pd->capacity * sizeof(int));
    clear(pd);
}

int get(const Dynarray* pd, size_t index)
{
    assert(index >= 0 && index < pd->size);
    return pd->data[index];
}

void set(Dynarray* pd, size_t index, int value)
{
    assert(index >= 0 && index < pd->size);
    pd->data[index] = value;
}
```

```
void reserve(Dynarray* pd, size_t new_capacity)
{
    if (new_capacity <= pd->capacity)
        return;

    int* new_data = (int*)ecmalloc(new_capacity * sizeof(int));
    for (size_t i = 0; i < pd->size; ++i)
        new_data[i] = pd->data[i];

    free(pd->data);
    pd->data = new_data;
    pd->capacity = new_capacity;
}

void push_back(Dynarray* pd, int x)
{
    static const double growth_factor = 2;
    if (pd->size == pd->capacity)
    {
        size_t new_capacity = (size_t)(growth_factor * pd->capacity);
        if (new_capacity <= pd->size)
            new_capacity = pd->size + 1;

        reserve(pd, new_capacity);
    }
    pd->data[pd->size] = x;
    pd->size += 1;
}

void print(const Dynarray* pd)
{
    printf("dynarray: ");
    for (size_t i = 0; i < pd->size; ++i)
        printf("%i ", pd->data[i]);
    printf("\n");
}

void destroy(Dynarray* pd)
{
    free(pd->data);
    pd->data = NULL;
}

int main()
{
    Dynarray a;
    init(&a, 0);
    for (int i = 0; i < 100; ++i)
        push_back(&a, i);
    print(&a);
    destroy(&a);
}
```

Пояснение по исходному коду динамического массива

- Нами была написана функция `ecmalloc`. Это просто `malloc` с проверкой на ошибки. В случае если `malloc` вернёт `NULL`, эта функция напечатает сообщение об ошибке и выйдет из программы. Функция печатает в стандартный поток `stderr`, предназначенный для вывода сообщений об ошибках.

- Динамический массив передаётся в функции по указателю или по константному указателю в зависимости от того будет ли меняться массив в функции или нет. При этом, тут мы передаём по неконстантному указателю даже если поля структуры `Dynarray` не меняется, но меняются элементы в куче. Например, в функцию `set` динамический массив передаётся не по константному указателю, а по обычному.

- Макрос `assert` или библиотеки `assert.h` – это простой, но эффективный способ для обнаружения ошибок. Assert с английского переводится как "утверждать". Макрос `assert` принимает на вход условие. Если условие истинно, то `assert` ничего не делает, но если условие ложно, то `assert` печатает сообщение об ошибке и выходит из программы. Например, следующая строка:

  ```
  assert(index >= 0 && index < pd->size);
  ```

  завершит программу с ошибкой, если `index` не принадлежит отрезку `[0, pd->size - 1]`.

- Функция `init` создаёт массив из нулевых элементов размера `initial_size`. Отдельно обрабатывается случай массива размера `0`. В этом случае память в куче не будет выделяться, а поле `data` будет равно `NULL`.

- Функция `reserve` увеличивает вместимость массива до значения `new_capacity`. Если вместимость уже больше или равна `new_capacity`, то это функция ничеко не делает.

- Функция `push_back` добавляет один элемент в конец массива. Если вместимости массива не хватает, то эта функция вызывает `reserve`, чтобы увеличить вместимость в 2 раза (за исключением случая, когда вместимость равна 0, в этом случае новая вместимость будет равна 1).

Стратегии роста вместимости динамического массива

Если в нашем массиве перестёт хватать места, то его вместимость увеличивается в 2 раза. В принципе, можно было выбрать и другую стратегию выделения памяти.

- **Аддитивная стратегия** – при нехватки места вместимость увеличивается на фиксированное количество элементов, например, на 100 элементов.

- **Мультипликативная стратегия** – при нехватки места вместимость увеличивается в фиксированное число раз, например, в 2 раза. Коэффициент увеличения вместимости массива называют фактором роста (англ. *growth factor*). Обычно фактор роста выбирают равным 1.5 или 2.

Посчитаем среднюю вычислительную сложность добавления элемента в конец массива при использовании той или иной стратегии. Предположим, что изначально массив пуст, а затем мы добавляем в него N элементов по одному, где N очень большое. При каждом перевыделении памяти нам нужно скопировать элементы из старого массива в новый.

В случае аддитивной стратегии с увеличением на 100 нам придется сделать:

$$100 + 200 + 300 + ... + N = \frac{N^2}{200}$$

копирований элементов. В среднем, на одно добавление элемента придётся $\frac{N}{200}$ копирований. То есть, средняя вычислительная сложность добавления элемента в данном случае будет равна $O(N)$.

В случае мультипликативной стратегии с фактором роста 2 нам придется сделать:

$$1 + 2 + 4 + 8 + ... + N = 2 \cdot N - 1$$

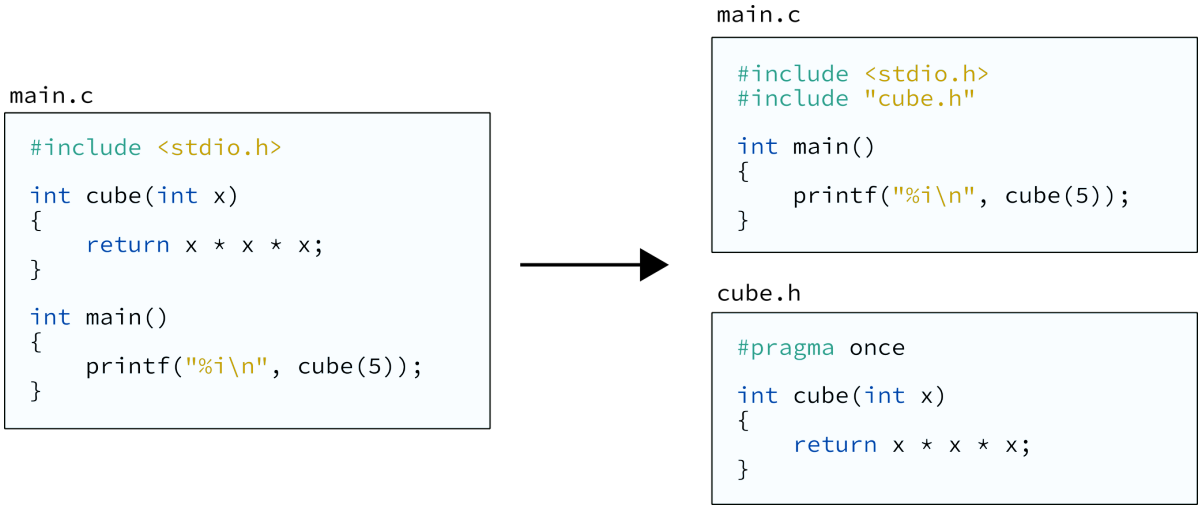
копирований элементов. В среднем, на одно добавление элемента придётся 2 копирования. То есть, средняя вычислительная сложность в данном случае будет равна $O(1)$.

Мультипликативная стратегия лучше в общем случае. Аддитивную стратегию можно использовать, если известно, что массив не будет часто расширяться и его размер всегда будет находится в некотором заранее известном интервале. В этом случае, использование аддитивной стратегии будет более эффективно по памяти.

Заголовочные файлы. Директива `#include`.

Ранее написанный код динамического массива работает корректно, но хотелось бы сделать его пригодным для повторного использования в других программах. Для этого необходимо вынести реализацию динамического массива в отдельный файл и затем подключать его в тех проектах, где требуется такая структура данных.

Простейший (но не совсем корректный) способ сделать нечто подобное – это выделить весь код реализации в отдельный файл, а подключать этот код к программе с помощью директивы `#include`. Рассмотрим, как происходит такое простейшее разделение программы на части, на примере простейшей программы, в которой мы хотим выделить одну функцию `cube` в отдельный файл.



- Скомпилировать такую, разделённую на две части программу, можно следующим образом:

  ```
  $ gcc main.c
  ```

  Передавать компилятору дополнительный файл `cube.h` не нужно. Он сам его найдёт, прочитав имя в директиве `#include`.

- В языке C принято давать файлам, подключаемым с помощью директивы `#include` расширение `.h`. Такие файлы называются *заголовочными файлами* (*header files*).

- Директива `#include` делает очень простую вещь – она просто ищет файл и вставляет всё содержимое этого файла на своё место. То есть в данном примере файл `cube.h` просто вставится в файл `main.c` за место строки `#include "cube.h"`.

- При использовании угловых `<>` скобок в `#include` компилятор будет искать файл в системных путях, а при использовании кавычек `""` сначала поищет в текущей директории.

- Заголовочные файлы стандартной библиотеки, такие как `stdio.h` должны храниться где-то в системе. В начале компиляции эти файлы ищутся и подставляются за место соответствующй директивы `#include`. Чтобы посмотреть все директории, в который компилятор `gcc` ищет заголовочные файлы, можно вызвать компилятор с опцией `-v` (от слова *verbose* - многословный): `gcc -v main.c`

- Как уже было сказано, директива `#include` не очень интеллектуальна – она просто вставляет содержимое файл за место себя. Но что произойдёт, если мы два раза включим один заголовочный файл `cube.h`. В этом случае у нас в программе будет два определения функции `cube`, что приведёт к ошибке.

  Случайно допустить ошибку двойного включения очень просто, особенно если программа состоит из множества файлов. Предположим, что мы написали ещё один файл `utils.h`, который внутри себя включает `cube.h`. Тогда, если мы напишем так:

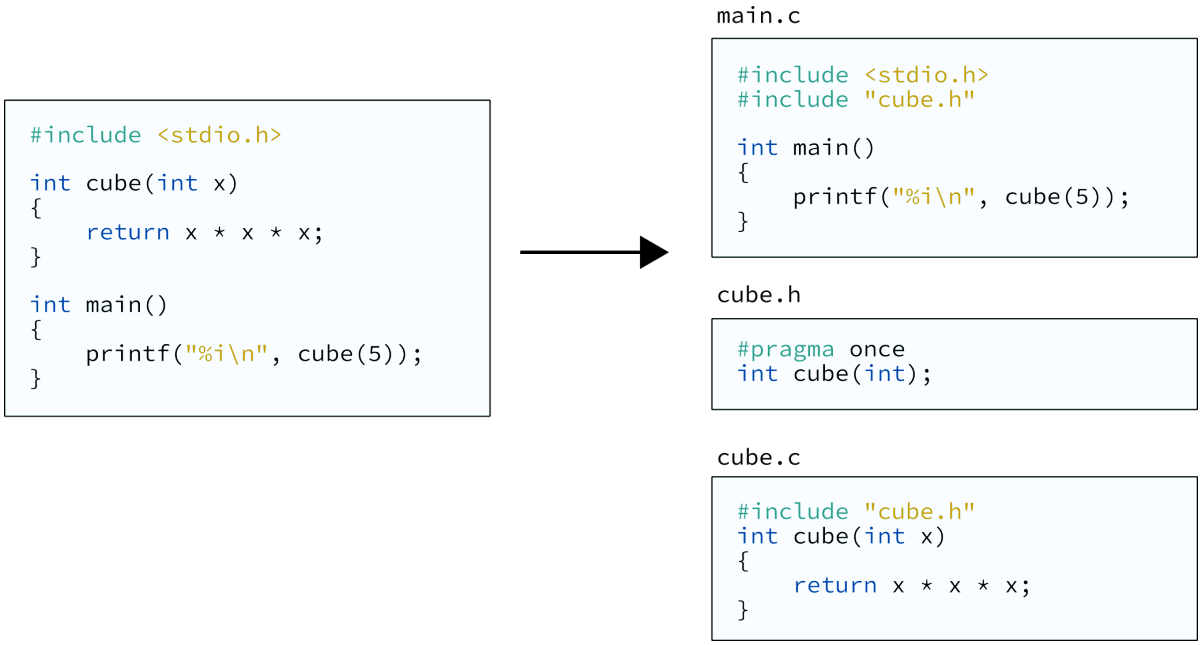
  ```
  #include "utils.h"  // utils.h тоже включает внутри себя cube.h
  #include "cube.h"   // Ошибка. Двойное включение!
  ```

  Директива `#pragma once` нужна для защиты от повторного включения. Пропишите эту директиву в начале заголовочного файла, и он будет включаться только один раз.

- Данный способ разбиения программы на файлы, хоть и является простейшим, **не является правильным**. Он будет работать только при подключении файла только в одну *единицу трансляции*.

Раздельная компиляция

Раздельная компиляция – это подход, при котором программа разбивается на несколько отдельных файлов исходного кода (`.c`), которые компилируются независимо друг от друга. В отличии от предыдущего способа при раздельной компиляции части программы будут компилироваться отдельно и объединяться вместе только на этапе линковки.
Единица трансляции – это исходный `.c` файл после того, как все директивы (`#include`, `#define` и другие) были обработаны. Одна программа может состоять из множества единиц трансляции.



- Скомпилировать такую программу можно следующей командой:

  ```
  $ gcc main.c cube.c
  ```

  Файл `cube.h` в аргументах компилятора указывать не нужно – он найдёт его через директиву `#include`. А вот файл `cube.c` обязательно нужно указать, иначе компилятор никак не сможет его найти.

- При раздельной компиляции подключаемый код состоит из двух файлов: заголовочного файла (`.h`) и файла исходного кода (`.c`). В заголовочных файлах обычно содержатся только объявления функций, структур, переменных. А в файлах исходного кода определения функций и переменных.

- В предыдущем способе в заголовочном файле `cube.h` содержалось определение функции. Это приводило к тому, что такой заголовочный файл нельзя было подключить в больше чем одну единицу трансляции одной программы. Всё потому, что в одной программе на языке C не может быть двух определений одной функции (даже если эти определения находятся в разных единицах трансляции). Более подробно почему это происходит мы будем проходить в дальнейшем. В случае раздельной компиляции заголовочный файл будет содержать только объявления и его можно будет без проблем подключать к разным единицам трансляции.

- Несмотря на то, что заголовочный файл, состоящий только из объявлений, можно подключать к различным единицам трансляции, двойное подключение такого файла *в одну единицу трансляции* по прежнему может привести к ошибкам. Например, ошибка случится, если в заголовочном файле содержится объявление структуры. Поэтому директиву `#pragma once` необходимо использовать и в этом случае.

- Заголовочный файл необходимо всегда подключать к соответствующему файлу исходного кода. Как в нашем примере файл `cube.h` был подключён к файлу `cube.c`. Даже если в некоторых случаях программа скомпилируется и без этого (как, например, в нашем), такое подключение является обязательным правилом хорошего тона для проверки согласованности.

Директивы препроцессора

Директивы макросов `#define`

Директивы условной компиляции

`#if`, `#else`, `#elif`, `#ifdef`, `#ifndef`, `#endif`.

Флаг `-D` для компилятора `gcc`

Стандартные макро-константы

```
__FILE__
__LINE__
__DATE__
__TIME__
__cplusplus
_WIN32
_WIN64
_MSC_VER
__MINGW32__
```

```c
#include <stdio.h>

int main()
{
#ifdef _WIN32
    printf("Windows\n");
#elif __linux__
    printf("Linux\n");
#elif __APPLE__
    printf("Apple\n");
#endif
}
```

Функциональные макросы

Многострочные макросы. Типичные ошибки, которые могут возникнуть при работе с функциональными макросами. Макросы `SUM`, `MULT`. Передача `i++` в макрос `SQUARE`. Макросы, содержащие другие макросы.

Использование оператора `do while` в многострочных функциональных макросах. Операция стрингификация (`#`). Склейка строковый макросов. Операция конкатенация (`##`). Макрос `assert` из библиотеки `assert.h`. Написание макроса, аналогичного макросу `assert`. Флаг `-E` компилятора `gcc`.

Макросы для объявления/определения новых структур. Динамический массив с разными типами.

Односвязный список

Односвязный список – базовая динамическая структура данных, состоящая из узлов, содержащих указатели на следующий узел списка. В отличии от массива элементы связного списка не лежат в памяти вплотную друг к другу. В качестве узла выступает структура, которая хранит элемент и указатель на следующий узел, например, вот такая:

```
struct node
{
    int value;
    struct node* next;
};
typedef struct node Node;
```

Для того, чтобы было удобно работать со связным списком также создадим структуру для самого списка. В простейшем случае она будет хранить указатель на первый узел:

```
struct forwardlist
{
    Node* head;
};
typedef struct forwardlist ForwardList;
```

В программе будем называть односвязный список как `ForwardList`, так как по нему можно перемещаться только вперёд. В памяти односвязный список будет выглядеть так: