Углублённое программирование на C++

Память в C++

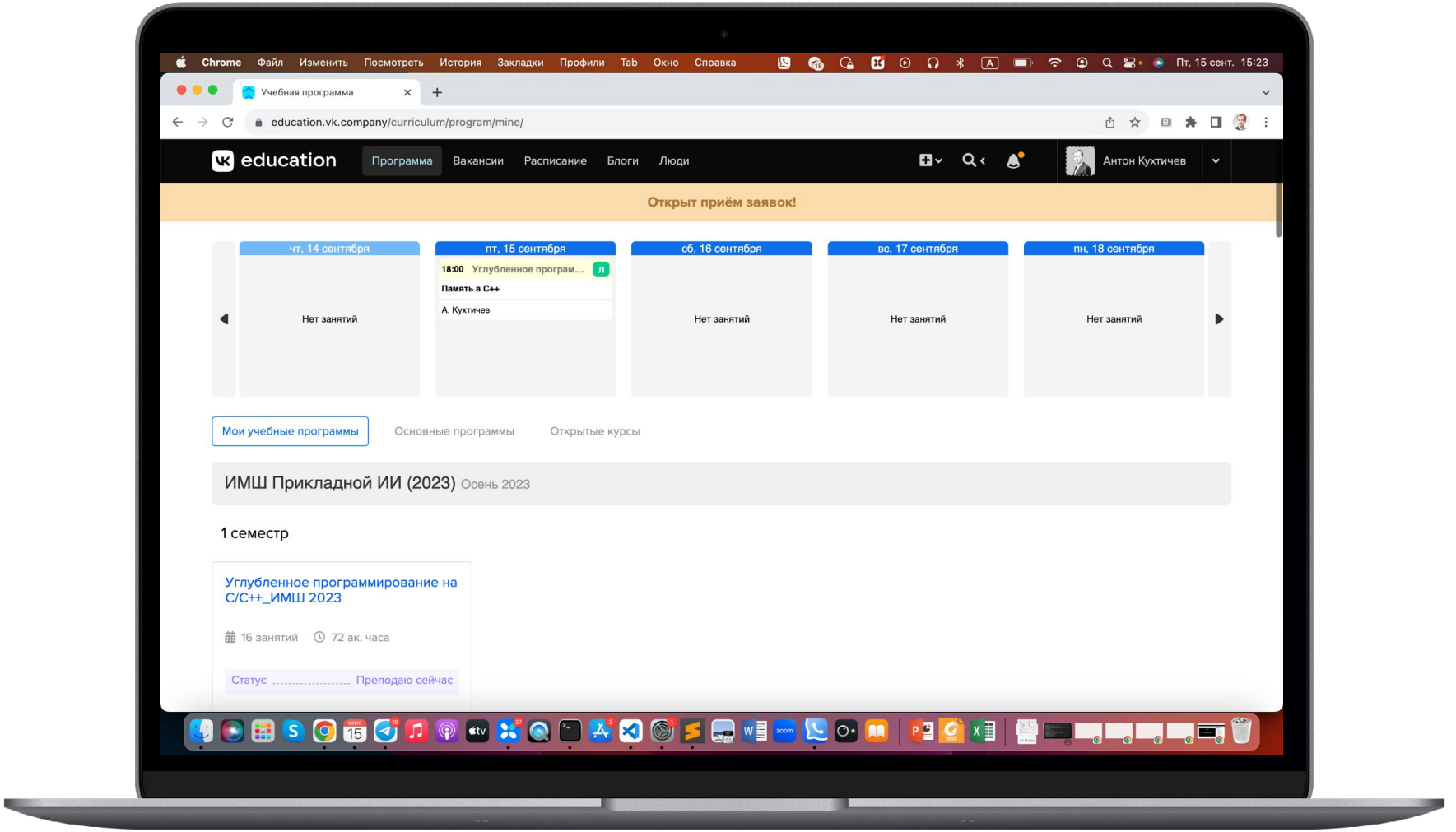
Кухтичев Антон

education

29 сентября 2025 года

Напоминание отметиться на портале

и оставить отзыв после лекции



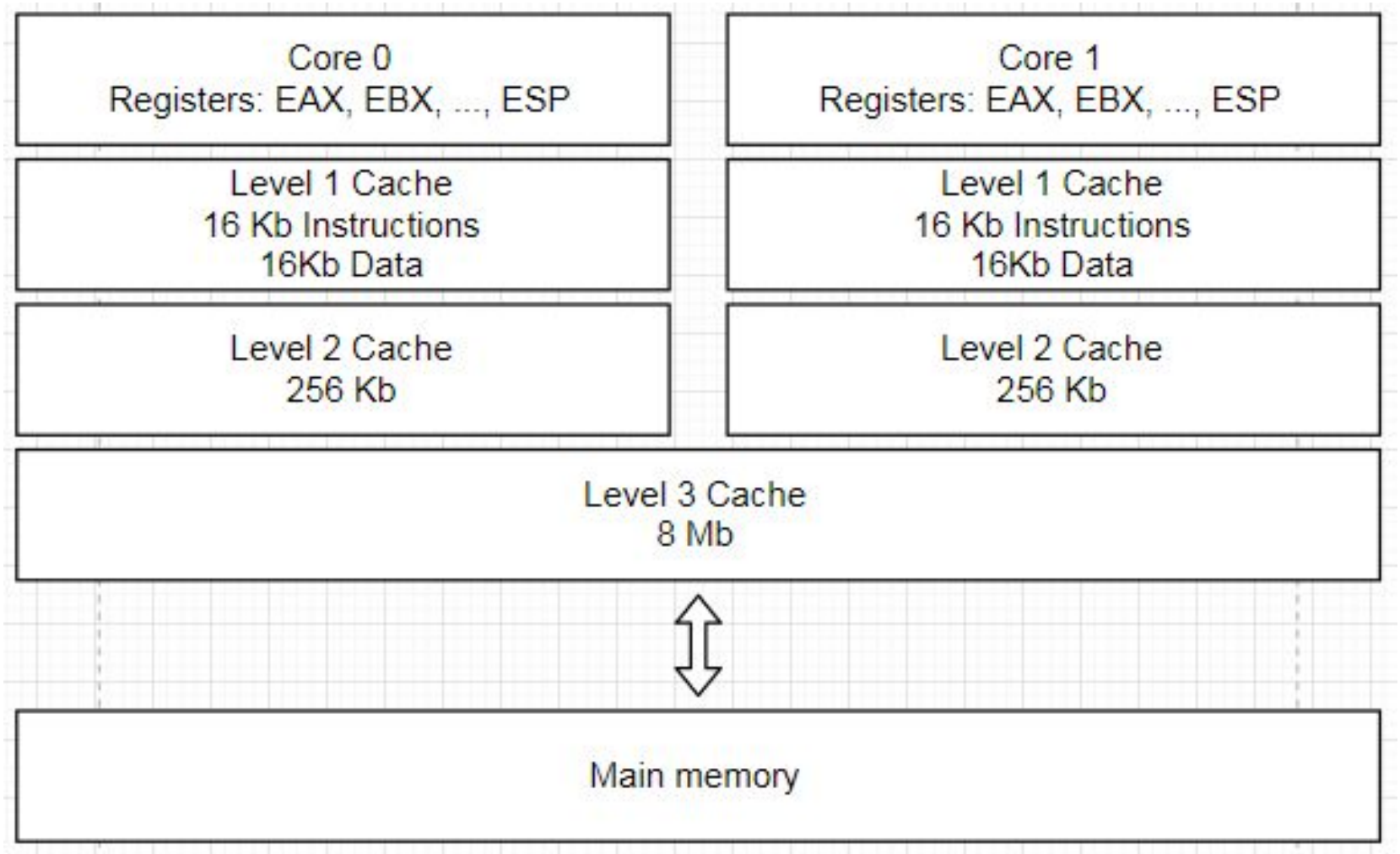
Содержание занятия

- Квиз #2
- Типичное размещение сегментов в памяти
- Запуск программы
- Кеш-память и оперативная память
- Виртуальная память
- Стек (stack)
- Куча (heap)
- Указатели
- Интеллектуальные указатели

Квиз #2

Размещение сегментов в памяти

Кеш-память и оперативная память



Что такое память?

- Память — часть компьютера, где хранятся программы и данные;
- Основной единицей памяти является двоичный разряд, который называется битом;
- Память состоит из ячеек;
- Каждая ячейка имеет номер, который называется адресом;
- В последние годы практически все производители выпускают компьютеры с 8-битными ячейками, которые называются байтами;
- Байты группируются в слова. Компьютер с 32-битными словами имеет 4 байта на каждое слово.

Запуск программы

- Программы на языке C/C++ начинают свою работу с вызова функции `main`

```
int main()
int main(int argc, char **argv)
int main(int argc, char **argv, char **env)
```

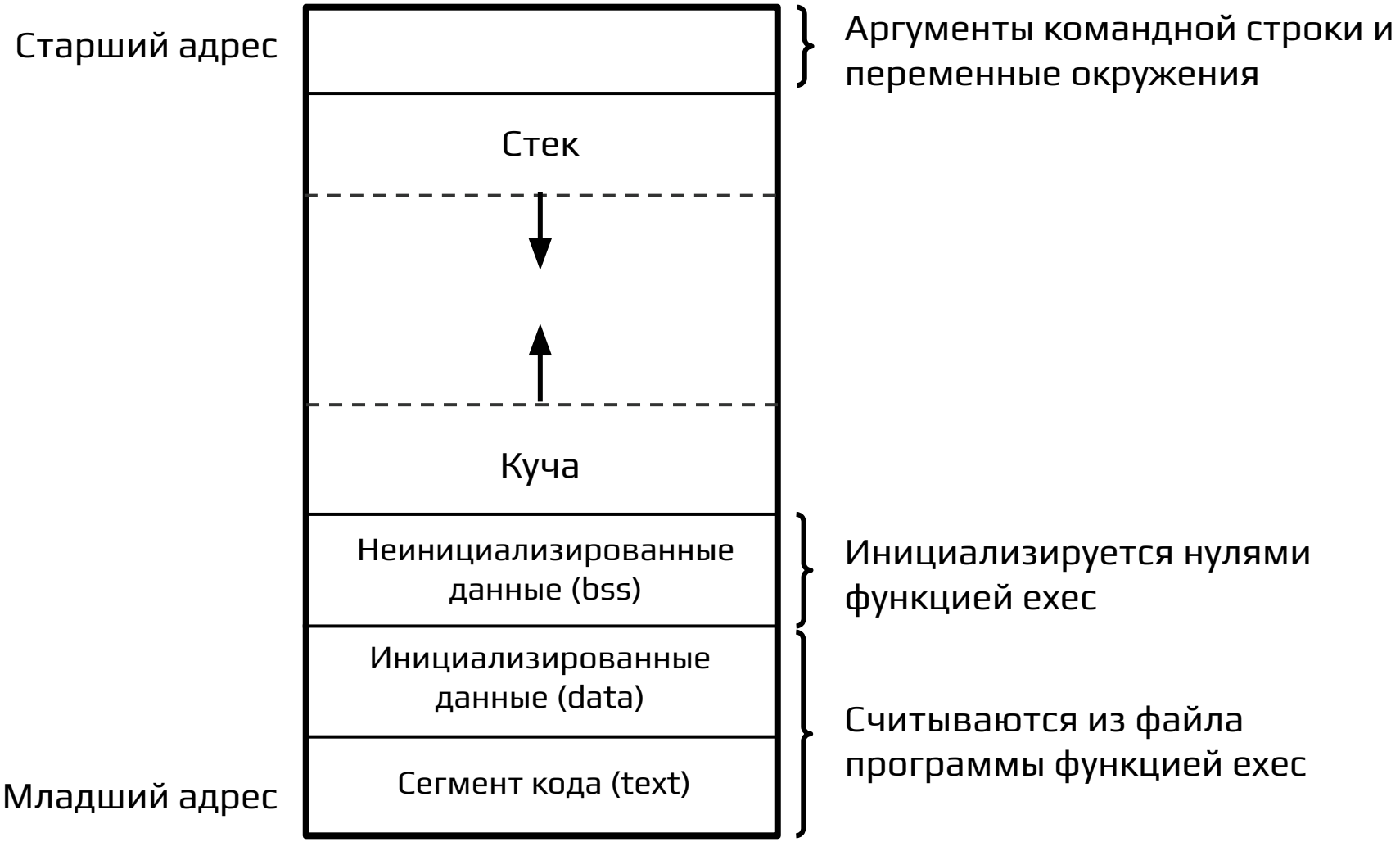
- Перед вызовом функции `main` выполняется специальная процедура начального запуска;
- Процедура начального запуска спроектирована так, что она вызывает функцию `exit`, когда происходит возврат из функции `main`.

```
exit(main(argc, argv));
```

Типичное размещение сегментов в памяти

- **Сегмент кода (text)**, машинные инструкции, которые выполняются центральным процессором;
- **Сегмент инициализированных данных (data)**, содержит переменные, которые инициализированы определёнными значениями в тексте программы;
- **Сегмент неинициализированных данных (bss)**, сегмент неинициализированных данных;
- **Сегмент стека (stack)**, где хранятся переменные с автоматическим классом размещения, а также информация, которая сохраняется при каждом вызове функции;
- **Куча (heap)**, или область динамической памяти.

Типичное размещение сегментов в памяти



Виртуальная память

Виртуальная память (1)

- Память делится на страницы;
- Страница может находится в оперативной памяти или на внешнем носителе;
- Трансляция из физического адреса в виртуальный и обратно выполняется через специальные таблицы: **PGD** (Page Global Directory), **PMD** (Page Middle Directory) и **PTE** (Page Table Entry). В PTE хранятся физические адреса страниц;
- Для ускорения трансляции адресов процессор хранит в кеше таблицу **TLB** (Translation lookaside buffer);

Виртуальная память (2)

- Если обращение к памяти не может быть оттранслировано через TLB, процессор обращается к таблицам страниц и пытается загрузить PTE оттуда в TLB. Если загрузка не удалась, процессор вызывает прерывание Page Fault.

- Обработчик прерывания Page Fault находится в подсистеме виртуальной памяти ядра ОС и может загрузить требуемую страницу с внешнего носителя в оперативную память.

Виртуальная память (3)

1 такт = 1 / частота процессора

1 / 3 GHz = 0.3 ns

	0.3 ns
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns

Compress 1K bytes with Zippy	3,000	ns
Send 1K bytes over 1 Gbps network	10,000	ns
Read 4K randomly from SSD	150,000	ns
Read 1 MB sequentially from memory	250,000	ns
Round trip within same datacenter	500,000	ns
Read 1 MB sequentially from SSD	1,000,000	ns
HDD seek	10,000,000	ns
Read 1 MB sequentially from HDD	20,000,000	ns
Send packet CA->Netherlands->CA	150,000,000	ns

Виртуальная память (4)

- Стараться укладывать данные в кеш;
- Минимизировать скачки по памяти;
- В условиях основной ветки делать ветку, которая выполняется с большей вероятностью.

Стек (stack)

Стек

- Непрерывная область памяти;
- Имеет методы: push и pop;
- Растёт от старших адресов к младшим;
- Имеет конечный размер (обычно около 2 MiB);
- Содержит локальные переменные из функций;
- Регистр ESP указывает на вершину стека. Меняется каждый раз, когда слово или адрес помещаются или удаляются из стека;
- Когда вызывается функция, то создаётся stack frame;
- Stack frame содержит локальные аргументы функции и возвращаемое значение;

Классы управления памятью и областью видимости в C++

- Характеризуются тремя понятиями:
  - Время жизни. Продолжительность хранения данных в памяти;
  - Область видимости. Части кода, из которых можно получить доступ к данным;
  - Связывание (linkage). Если к данным можно обратиться из другой единицы трансляции - связывание внешнее (external), иначе связывание внутреннее (internal);

Автоматический/регистровый (register)

- Время жизни: автоматическое (блок)
- Область видимости: Блок
- Связывание: отсутствует

```
{
    int i = 5;
}


if (true) {
    register int j = 3;
}
```

Статический без связывания

- Время жизни: статическое
- Область видимости: Блок
- Связывание: отсутствует

```
void foo()
{
    static int j = 3;
}
```

Статический с внутренним связыванием

- Время жизни: статическое
- Область видимости: файл
- Связывание: внутреннее

```
static int j = 5;
```

Инициализируется до входа в main.

Статический с внешним связыванием

- Время жизни: статическое
- Область видимости: файл
- Связывание: внешнее

```
// *.cpp
int i = 0;
// *.hpp
extern int i;
```

Куча (heap)

Heap

```
// Возвращают непустой указатель на
// выделенную область памяти или NULL.
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t new_size);

void free(void *ptr);
```

Heap

```
// можно (int*)(sizeof(int)), но лучше так не делать!
int *i = static_cast<int *>(malloc(sizeof(int));
std::string *name = new std::string();
char *data = new char[5];
...
free(i);
delete(name);
delete[] data;
```

1. Скотт Мейерс. Наиболее эффективное использование C++. Правило 2. Предпочитайте приведение типов в стиле C++.
2. Скотт Мейерс. Эффективное использование C++. Правило 16. Используйте одинаковые формы new и delete.

Heap

- `new` то же, что и `malloc()`, только дополнительно вызывает конструкторы;
- `new` возвращает определённый тип, в то время как `malloc()` возвращает `void *`;
- В случае ошибки, `malloc()` вернёт NULL, в то время как `new` выбросит исключение `bad_alloc` exception;
- Внутри `malloc` есть буфер, если в буфере есть место, ваш вызов может выполнится быстро;
- Если памяти в буфере нет, будет запрошена память у ОС (sbrk, VirtualAlloc) - это дорого;
- Для `malloc()` должны автоматически определять размер данных (sizeof * количество данных), в то время как для `new` мы указываем количество объектов;
- ОС выделяет память страницами от 4 KiB, а может быть и все 2 MiB;
- Стандартные аллокаторы универсальные, то есть должны быть потокобезопасны, быть одинаково эффективны для блоков.

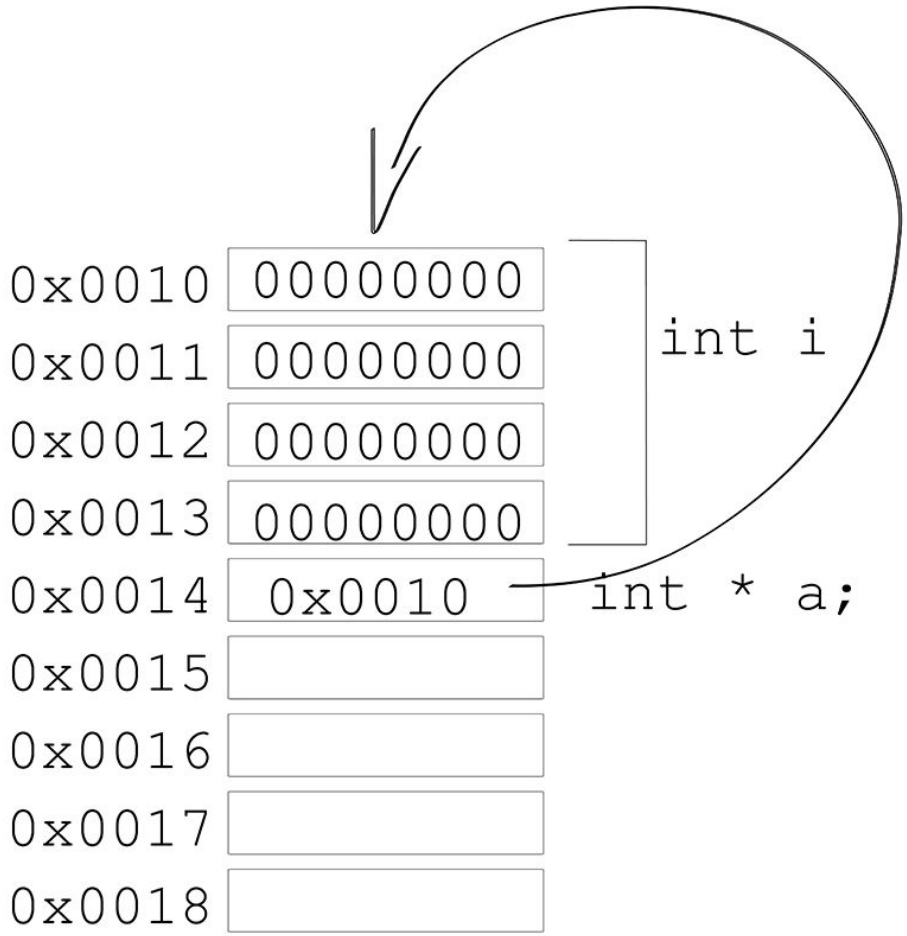
Указатели

Указатели

- Указатель — это переменная, содержащая адрес другой переменной.
- Для нулевого указателя используется одно из: `0`, NULL (C-style) или `nullptr` (современный C++);
- `nullptr` — можно рассматривать как указатель *любого* типа;
- Основной операцией над указателями является разыменование — `*` — получение объекта, на который указывает указатель.
- Другая операция — `&` — взятие адреса

Скотт Мейерс. Эффективный и современный C++. Пункт 3.2 Предпочитайте nullptr значениям 0 и NULL.

Указатели


0x0010
0x0011
0x0012
0x0013
0x0014
0x0015
0x0016
0x0017
0x0018
00000000
00000000
00000000
00000000
0x0010
int i
int * a;

Указатели

- Для любого объекта `x`, выражение `&x` возвращает адрес `x`;
- Если у `x` тип `T`, то у `&x` будет тип “указатель на T”;

```
int i;

unsigned long ul;

int *pi = &i;

unsigned long *pul = &ul;
```

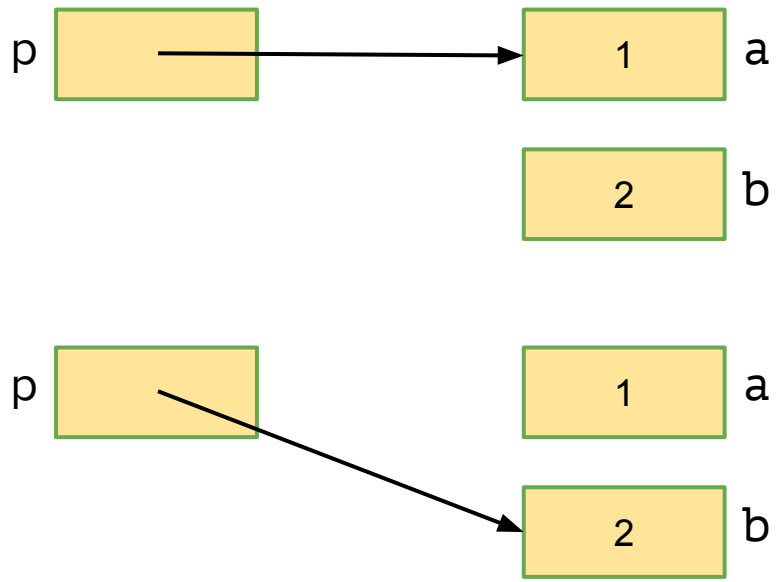
Указатели

- Указатель может указывать на разные объекты в разное время в отличие от ссылок

```
int a = 1, b = 2;

int *p = &a;


p = &b;
```



Указатели

- Ссылка всегда определена в отличие от указателя;

```
char *ch = nullptr; // Хорошо или плохо?;
char &ch_ref;       // Хорошо или плохо?
char &ch_ref = *ch; // Хорошо или плохо?
```

Указатели

- Ссылка всегда определена в отличие от указателя;

```
char *ch = nullptr; // Всё хорошо, указатель ссылается на nullptr;
char &ch_ref;       // Ошибка, ссылка не инициализирована!
char &ch_ref = *ch; // Так можно, но если у вас возникает желание
писать такой код, то лучше отказаться от идеи использовать ссылки.
```

Скотт Мейерс. Наиболее эффективное использование C++. Правило 1. Различайте указатели и ссылки.

Ссылки и указатели

- `int *x[10];` `// ???`
- `int (*y)[10];` `// ???`
- `const int *z1;` `//`
- `int const *z2;` `//`
- `int * const z3;` `//`

Упражнение

1. Какой результат сравнения?

```
NULL == nullptr
```

2.
```
Obj *op = new Obj;
Obj *op2 = op;
delete op;
delete op2;
```

3.
```
void f(int i);
void f(char *s);
f(NULL);
f(nullptr);
```

Ответы

1. `true`, гарантировано стандартом

   `Two operands of type std::nullptr_t or one operand of type std::nullptr_t and the other a null pointer constant compare equal.`

2. Неопределённое поведение, но скорее всего runtime error;

3. ```
   f(NULL);    // Вызовется f(int)
   f(nullptr); // Вызовется f(char *)
   ```
   ```

Интеллектуальные указатели

Интеллектуальные указатели (smart pointers)

- Объявления указателя не даёт информации о том, указывают ли они на один объект или на массив;
- Объявление указателя ничего не говорит о том, должны ли вы уничтожить то, на что он указывает, когда завершите работу;
- Если требуется вызвать `delete`, то причина 1 означает, что нет никакого способа узнать, следует ли использовать оператор для удаления одного объекта (`delete`) или для удаления массива (`delete []`);
- Обычно нет способа выяснить, не является ли указатель висячим.

Скотт Мейерс. Эффективный и современный C++.
Глава 4.

Виды интеллектуальных указателей

- `std::auto_ptr`
  - `удалён в C++17;`
- `std::unique_ptr`
  - Воплощает в себя семантику исключительного владения;
  - Копирование `std::unique_ptr` не разрешается;
- `std::shared_ptr`
  - Воплощает в себя семантику совместного владения;
  - Все указатели, указывающие на объект, сотрудничают для обеспечения гарантии, его уничтожение произойдёт в точке, где он станет более ненужным.
- `std::weak_ptr`
  - Используется для разрешения проблемы циклической зависимости.

std::shared_ptr

```
// До C++17
std::shared_ptr<char> ptr(new char[size_], std::default_delete<char[]>());

// Начиная с C++17
std::shared_ptr<char[]> ptr(new char[size_]);
```

Домашнее задание

Домашнее задание #1 (1)

Написать свой аллокатор со стратегией линейного выделения памяти со следующим интерфейсом:

```
struct Allocator;

void init_allocator(Allocator *alloc, size_t maxSize);
char* alloc(Allocator *alloc, size_t size);
void reset(Allocator *alloc);
void clear(Allocator *alloc)
```

Домашнее задание #1 (2)

При вызове `init_allocator` аллоцируется указанный размер, после чего при вызове `alloc` возвращает указатель на блок запрошенного размера или `nullptr`, если места недостаточно. После вызова `reset` аллокатор позволяет использовать свою память снова.

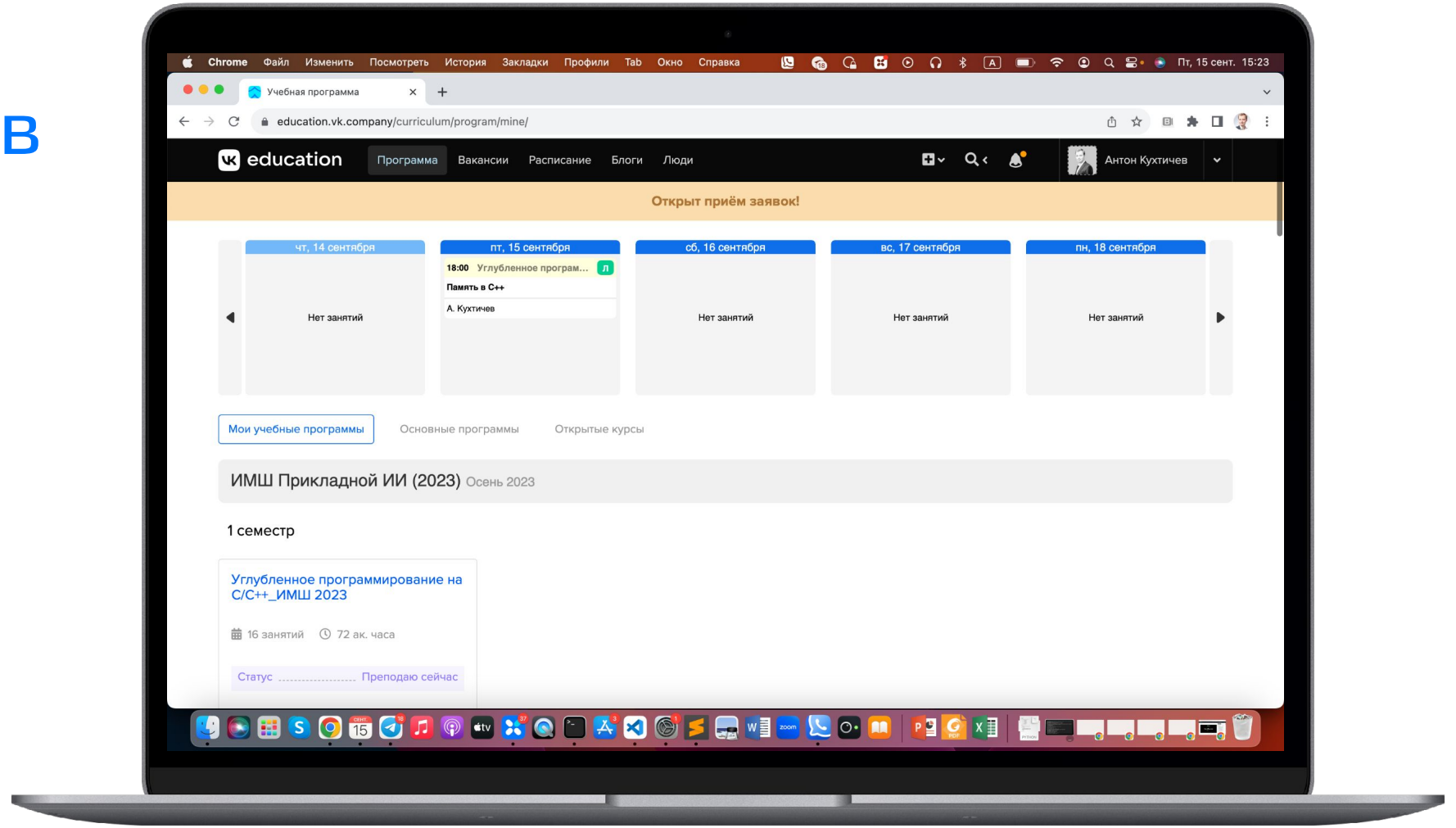
Написать юнитесты при помощи gtest. В Makefile должна быть цель test.

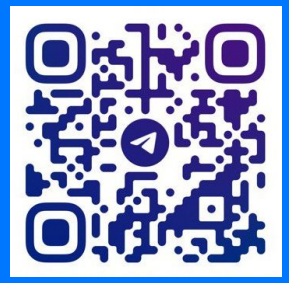
Полезная литература в помощь

- Э. Таненбаум. «Архитектура компьютера»
- Э. Таненбаум, А. Вудхалл. «Операционные системы. Разработка и реализация»
- У. Ричард Стивенс, Стивен А. Раго «UNIX. Профессиональное программирование»

Напоминание оставить отзыв

Это правда важно





Спасибо
за внимание!