СЕМИНАР IV

Разбор проблем с прошлой дорешки и контрольной

си-строки и массивы char

- массив не обязан заканчиваться на nul (`'\0'`), си-строка - обязана
- `strlen` считает количество символов, за исключением nul. Память надо выделять +1
- `strcpy` копирует строку, включая nul, `strncpy` копирует указанное количество символов (или до nul)
- `memcpy` копирует указанное количество байтов

IntVector

- На всякий `new[]` нужен свой `delete[]`
- Формулируйте контракты как можно полнее, - это позволит избежать глупых ошибок

Контракты IntVector

- структура принимает значения
  - `{nullptr, 0, 0}` (исходное)
  - `{new int[C], S, C}`, где `S <= C`
- pushBack
  - `{new int[C], S, C}` ⇒ `{new int[C1], S+1, C1}`
  - ничего не забыли?
- popBack
  - `{new int[C], S, C}` ⇒ `{new int[C], S-1, C}`
- deallocate
  - ⇒ исходное состояние

Stack & Heap Growth

- Нет смысла исследовать один кадр стека

```
void grows_wrong() {
    int x, y, z;
    int a[3];
}

void grows_right(int& outer) {
    int inner;
}
```

Работа с односвязными списками

- Ветвление: удаляем/добавляем голову, промежуточный или последний элемент
- Локальные переменные: (указатель на новую голову, указатель на предыдущий элемент)
- Эврика №1: new_head — указатель на новую голову + pprev — указатель на указатель текущего элемента (&new_head или &prev→next)
  - current == *pprev
- Эврика №2: фиктивный головной элемент + указатель на предыдущий (&dummy_head или prev)
  - current == prev→next

Поиск/замена

- Повторная замена должна начинаться с конца заменённого фрагмента
  - `replace("hello aaaa* world a* !", "a*")`
  - `replace("hello *aaaa world *a !", "*a")`

Примеры классов — string

- контейнер над строками
- хранят данные в куче (или SOO[*])
- удобный интерфейс: доступ по индексу, find…
- данные копируются (в конструкторе/операторах) и очищаются (в деструкторе)

string_view

- легковесные представления строчек
- не владеют данными:
  - нужно передать их при инициализации
  - они неизменяемы внутри view
- оберткой можно пользоваться до тех пор, пока данные доступны
- быстро работает копирование и методы `substr`, `remove_prefix` и т.д.

```cpp
std::string s = "foobar";
std::string subStr  = s.substr(3);      // copy

std::string_view sv = s;                // no copy
std::string_view subSv = sv.substr(3); // no copy
```

Системы сборки

Мотивация:

- Большой проект — много зависимостей
- C++ позволяет компилировать единицы трансляции отдельно + слинковывать их позднее
- ⇒ на изменениях можно перекомпилировать только нужное [и другие бонусы]
- Нужно описание целей и зависимостей, и команды сборки: make, ninja, MS VC++, CodeBlocks…
- `CMake` — генератор (разных) конфигов сборки, проектов IDE

Makefiles

Декларативное описание целей и зависимостей:

```
<target>: <dep1> <dep2> ...
[tab] <command>

main.o: main.cpp
        g++ -c main.cpp

hello.o: hello.cpp
        g++ -c hello.cpp

all: main.o hello.o
        g++ main.o hello.o -o hello
```

Задачи

#1 Немного композиций

Функции, у которых аргумент и возвращаемое значение совпадают по типам прекрасны тем, что их можно комбинировать!

Реализуйте функцию `Compose` со следующей сигнатурой [**0.5 балла**]:

```
using FuncT = double(*)(double);

// n - число композированных функций
// Пример: Compose(2, f, g, 3.1415) вычисляет f(g(3.1415
double Compose(size_t n, ...);
```

Проблема предыдщуей реализации в том, что её не очень удобно использовать: каждый раз надо передавать все функции для композиции. Как решить эту проблему? Добавить состояние.

Реализуйте класс `Composer`, сохраняющий внутри себя композируемые функции [**0.5 балла**]:

```
class Composer {
 public:
  // Сигнатура как у функции Compose, но аргумент не пер
  Composer(size_t n, ...);
  // Применяет композицию сохраненных функций к arg
  double operator()(double arg) const;
// Добавляйте всё, что необходимо (но лишнего не надо!)
};
```

#2 C from C++

- `sum.h`
  - объявить функцию `int sum(int, int)`
- `sum.c`
  - определить функцию `int sum(int, int)`
- `test.cpp`: подключить `sum.h`
- написать достаточный `Makefile` с таргетом `all` (для linux [*])

[*]: компилировать необходимо соответствующими компиляторами в отдельные объектные файлы (`gcc`/`clang`/`cl` для комиляции `C`)

#3 Enumerations

Любой современный человек жить не может без календарей. Тем не менее, удобных бибилиотек для работы даже с такой простой вещью как дни недели всё ещё очень мало. Ну разве что стандартная библиотека c++ приходит в голову…

Давайте попробуем чуть улучшить ситуацию!

Реализуйте в пространстве имён `wdu` (week day utils) перечисление `WeekDay`, определяющее дни недели, и три функции для работы с ним

```
// Возвращает имя дня недели на английском с большой бук
const char *GetDayOfWeekName(WeekDay w);
// Возвращает true для субботы и воскресенья (да простят
bool IsWeekend(WeekDay w);
// По году, месяцу и дню возвращает день недели. Если да
WeekDay GetDayOfWeek(size_t year, size_t month, size_t d
```

#4 StringView

Реализуйте класс `StringView` (для элементов типа `char`), `npos` и методы:

1. конструкторы: от си-строки, от `std::string`, от `char* + size_t` (указатель на начало + размер)
2. операторы присваивания: от си-строки, от `std::string`, от `StringView`
3. `data()`, `size()`
4. `substr` — возвращает `StringView`, принимает стартовый индекс + опциональный конечный индекс (default: `npos`)

1. `operator[]` — достает соответствующий индексу элемент
2. `find` — возвращает индекс начала подстроки или `npos`, принимает в аргументе `StringView` или `char`
3. `startsWith`, `endsWith` — принимает в аргументе `StringView` или `char`
4. `findFirstOf`, `findFirstNotOf` — *same* + опциональный стартовый индекс (default: `0`)
5. `removePrefix` — сдвигает начало на `size_t` (аргумент), в самом объекте stringView