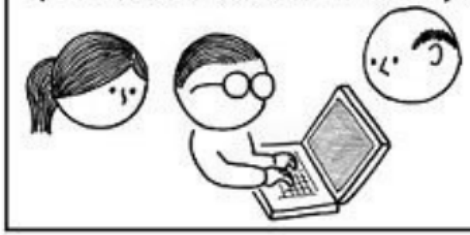
Дни **1 - 10**
Выучить перменные, констаты, массивы, строки, выражения, функции...

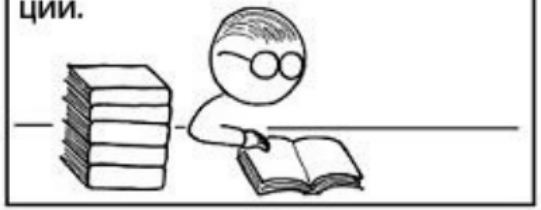
Дни **11 - 21**
Выучить потоки, указатели, ссылки, классы, объекты, наследование, полиморфизм...

Дни **22 - 697**
Много программировать для себя. Иногда взламывать что-то, но все время учится на ошибках.

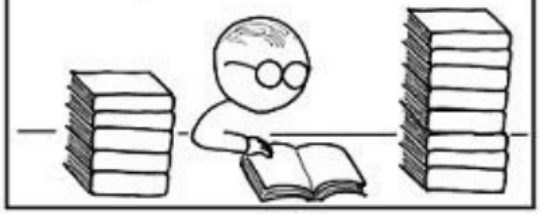
Дни **698 - 3648**
Общаться с другими программистами. Работать над проектами с ними. Учиться у них.



Дни **3649 - 7781**
Выучить продвинутую теоритическую физику и сформулировать теорию квантовой гравитации.

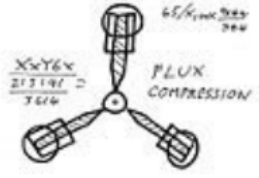


Дни **7782 - 14611**
Выучить биохимию, молекулярную биологию, генетику...



День **14611**
Использовать знания по биологии для создания омолаживающего зелья.

День **14611**
Использовать знания по физике для создания поточного конденсатора и вернуться в день 21.



День **21**
Заменить себя-из-прошлого.

Насколько я знаю,
это самый простой способ
"Выучить C++ за 21 день".

(*) На слайде есть опечатка

Введение в ООП, инкапсуляция

ООП/C++: Лекция 1

ака «Какие такие объекты, нормально же без них писали?»

О чём лекция сегодня

1. Немного оргмоментов
2. Послесловие прошлого семестра
3. Зачем вообще придумали ООП
4. ООП: первое демо
5. Инкапсуляция
6. Финальные ремарки

Немного оргмоментов

Оргмоменты

- Семестр посвящён объектно-ориентированному программированию вообще и на C++ в частности.
- Главная Книга Семестра: Стивен Прата - Язык программирования C++. Лекции и упражнения. - 2012.
- Будет одна контрольная. Вероятно, в конце апреля.

Послесловие прошлого семестра

Что стоит запомнить из прошлого семестра

Вечные ценности из прошлого семестра:

- Асимптотическая сложность алгоритмов
- Устройство памяти и работа с ней

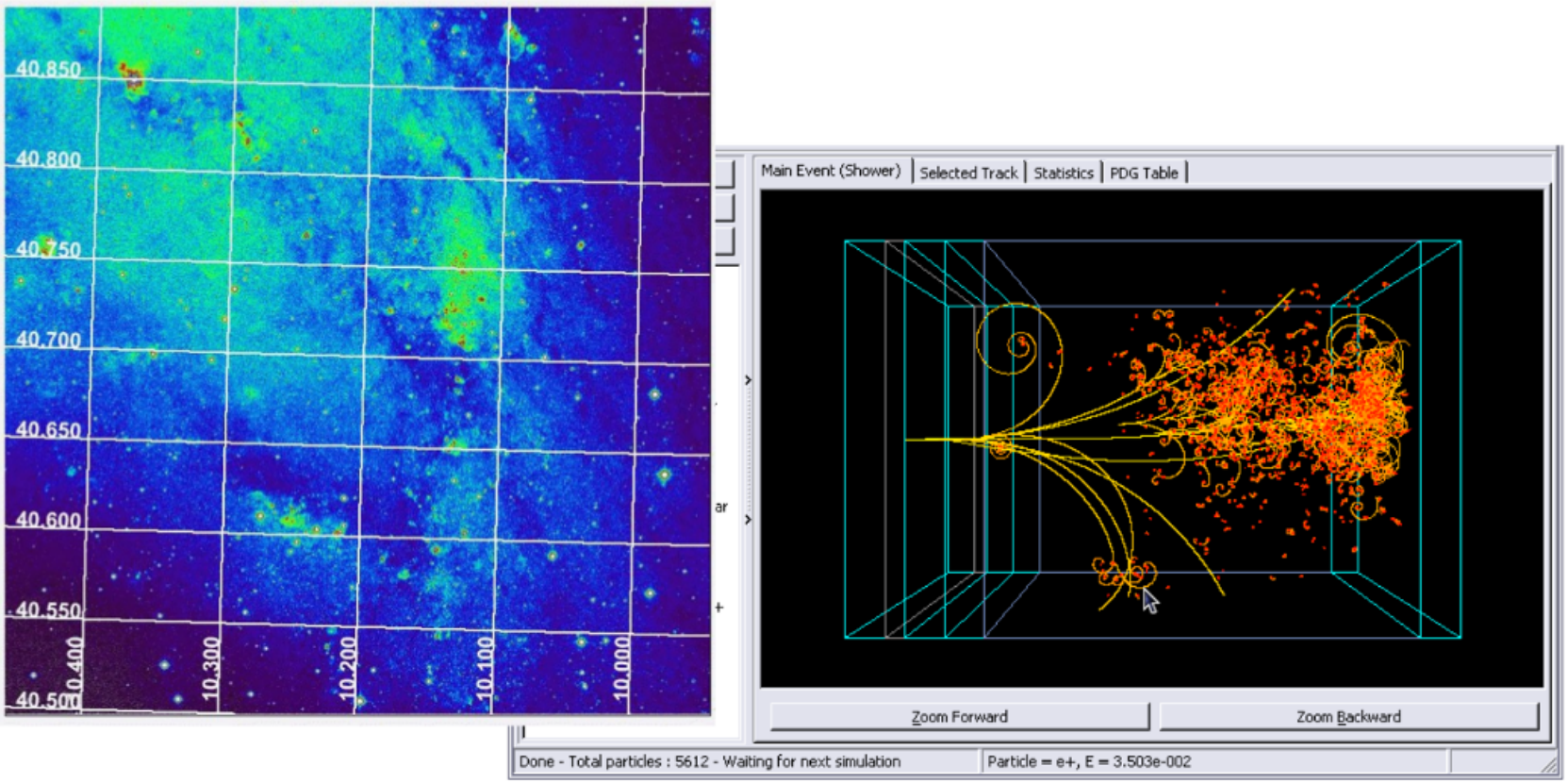
А ещё нам пригодится базовый синтаксис C++, хотя мы его сильно расширим в этом семестре.

Что пишут на C++

Встраиваемые системы (в том числе бортовые) – например, автопилот Curiosity

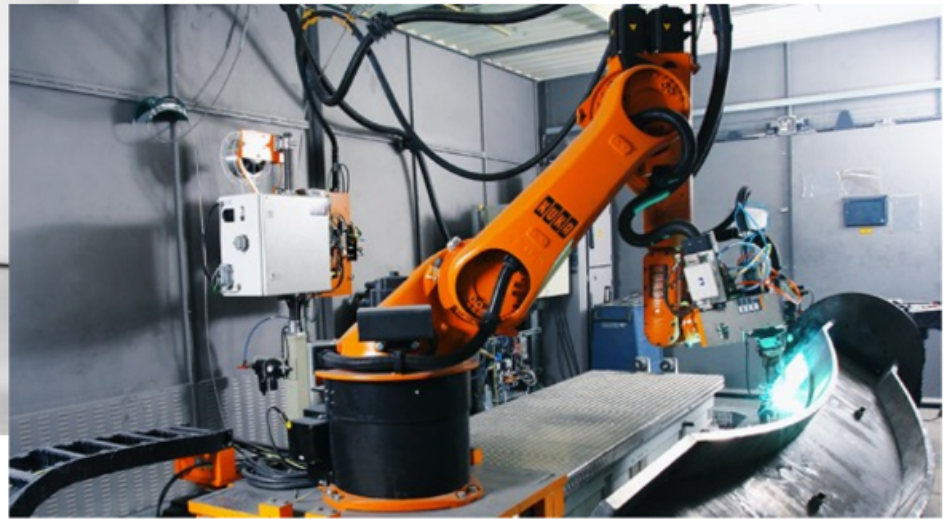
Что пишут на C++

Обработка данных больших экспериментов (например в CERN), моделирование физики (повсеместно)



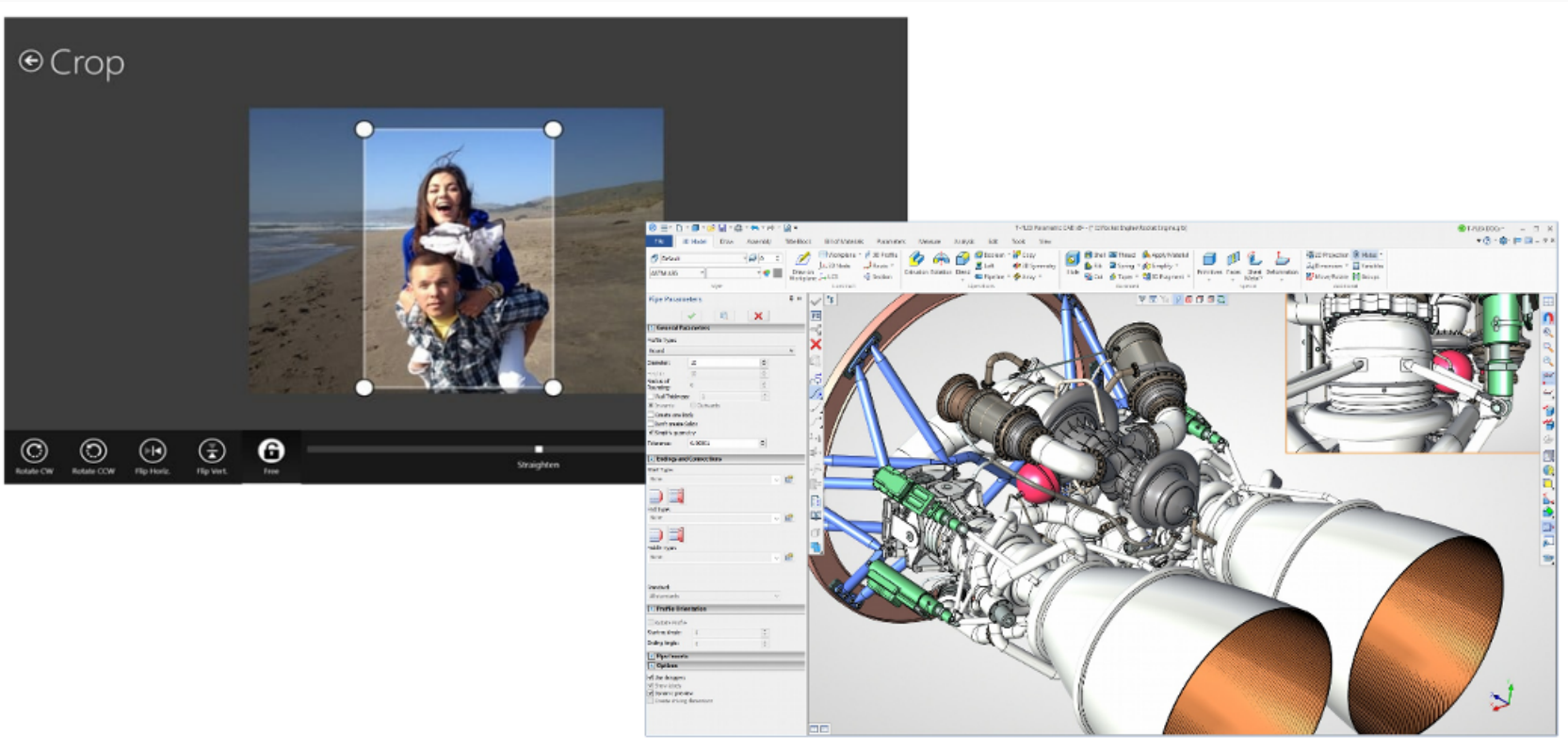
Что пишут на C++

«Мозги» систем, критичных ко времени выполнения (в самых разных областях)



Что пишут на C++

«Тяжёлые» приложения – CAD-системы, обработка видео, браузеры и т.д.



Что пишут на C++

Игры, спецэффекты, прочий рендеринг

Что НЕ пишут на C++

- Веб-приложения – в теории это можно, но в среднем очень неудобно
- Бизнес-логику корпоративных систем – есть более заточенные на это языки
- Прототипы чего бы то ни было – писать приложение на C++ в среднем небыстро

C++ : нюансы

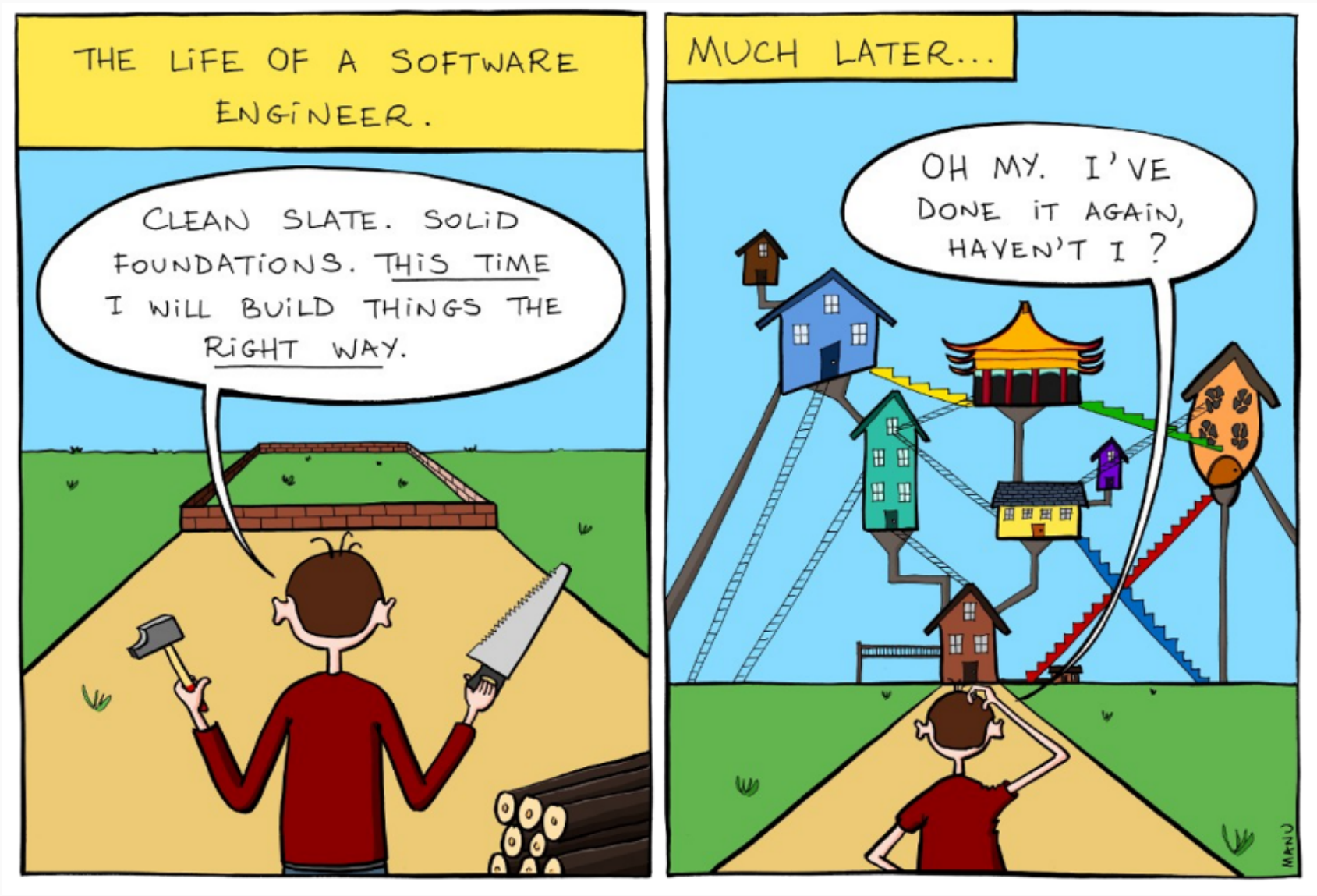
Можно сделать всё. Но для этого нужно знать очень много.

Причём не только язык, но и библиотеки к нему.

Зачем вообще придумали ООП

Откуда всё взялось

Цель ООП: нормально организовать код, когда его много



Нюансы

Возможные проблемы при столкновении с ООП:

- Совсем другой взгляд по сравнению с процедурным программированием
- Иногда придётся принимать на веру, что «так правда удобно, когда кода уже много»

ООП: первое демо

Первый пример

Disclaimer: Пример написан местами ужасно ради простоты изложения на данной лекции. Где именно ужас, и как его поправить – станет ясно в ходе дальнейших лекций.

Полный код к этой лекции:

https://github.com/avasyukov/oop-2nd-term/tree/master/2020/lection01

Было в процедурном стиле

```
// Представление стека в памяти
struct stack_handle
{
        int size;           // Размер стека
        int* data;          // Указатель на данные
        int top;            // Верхний элемент
};

// Функции для работы со стеком
void init(struct stack_handle* s, int size);
void finalize(struct stack_handle* s);
void push(struct stack_handle* s, int a);
int pop(struct stack_handle* s);
void clear(struct stack_handle* s);
```

Было в процедурном стиле

Ну так и прекрасно:

```
{
    struct stack_handle s1;
    init(&s1, 10);          // Инит стека на 10 элементов
    push(&s1, 1);           // Чего-нибудь сложим в стек
    push(&s1, 2);
    push(&s1, 3);
    cout << pop(&s1) << endl;    // Теперь достанем
    cout << pop(&s1) << endl;
    cout << pop(&s1) << endl;
    finalize(&s1);          // Завершение работы
}
```

Было в процедурном стиле

А если так?

```
{
    struct stack_handle s2;
    init(&s2, 10);
    push(&s2, 1);
    push(&s2, 2);
    push(&s2, 3);
    cout << pop(&s2) << endl;
    s2.top = 1;                    // Oooops
    cout << pop(&s2) << endl;
    finalize(&s2);
}
```

Было в процедурном стиле

Кто сказал «так нечестно»?

Проблемы

Очевидные неприятности:

- Вызывающий код может всё сломать
- Вызывающий код обязан знать внутреннюю логику – как минимум, вызывать init и finalize, которые вроде как внутреннее дело стека
- Вызывающий код зачем-то должен знать про устройство struct stack_handle со всеми его полями, хотя логически хочет видеть просто стек

Станет в объектном стиле

```
class stack
{
public:
        stack(int size);
        ~stack();
        void push(int a);
        int pop();
        void clear();
private:
        int size;
        int* data;
        int top;
};
```

Немного терминов (1/3)

- Класс – логическая сущность, содержит в себе поля и методы
- Поля – данные (переменные, массивы, прочее) «внутри» класса, примерно как в структуре
- Методы – функции, «логически прицепленные» к классу («входящие» в класс)

Немного терминов (2/3)

- Конструктор – служебный метод, вызывается «как-то сам» при создании экземпляра класса, отвечает за инициализацию всего
- Деструктор – служебный метод, вызывается «как-то сам» при удалении экземпляра класса, отвечает за очистку всего
- this – служебный указатель, доступен в методах, показывает на текущий экземпляра класса, для которого вызван метод

Немного терминов (3/3)

- Публичные методы и поля – те, к которым может обратиться код, внешний по отношению к классу
- Приватные методы и поля – те, которые доступны только из методов самого класса

Станет в объектном стиле

```
class stack
{
// Публичные методы класса, к ним можно обращаться извне
public:
        stack(int size);   // Конструктор
        ~stack();          // Деструктор
        void push(int a);          // <-|
        int pop();                 // <-|- Методы
        void clear();              // <-|
// Приватные поля класса, нет доступа извне
private:
        int size;          // Размер стека
        int* data;         // Указатель на данные
        int top;           // Верхний элемент
};
```

Станет в объектном стиле

Конструктор, инициализирует поля класса при его создании:

```
stack::stack(int stack_size)
{
    this->top = 0;
    this->size = stack_size;
    this->data = new int[stack_size];
}
```

Станет в объектном стиле

Деструктор, чистит память при удалении стека:

```
stack::~stack()
{
    delete[] this->data;
}
```

Станет в объектном стиле

Пример метода, складывает новый элемент в стек:

```
void stack::push(int a)
{
    if(this->top == this->size)
    {
        cout << "Stack overflow!" << endl;
        return;
    }
    this->data[this->top] = a;
    this->top++;
}
```

Станет в объектном стиле

Можно и не писать this, если это не вызывает проблем с именами полей и локальных переменных:

```
void stack::push(int a)
{
    if(top == size)
    {
        cout << "Stack overflow!" << endl;
        return;
    }
    data[top] = a;
    top++;
}
```

Станет в объектном стиле

```
{
    stack s1(10);           // Инит стека на 10 элементов
    s1.push(1);             // Чего-нибудь сложим в стек
    s1.push(2);
    s1.push(3);
    cout << s1.pop() << endl;    // Теперь достанем
    cout << s1.pop() << endl;
}
```

Станет в объектном стиле

```
{
    stack s2(10);
    s2.push(1);
    s2.push(2);
    s2.push(3);
    cout << s2.pop() << endl;
    s2.top = 1;                    // Так теперь нельзя
    cout << s2.pop() << endl;
}
```

Итоги

Чего мы добились:

- Вызывающий код не может ничего сломать (по крайней мере, очевидным образом)
- Аналоги init и finalize срабатывают «как-нибудь сами» в нужные моменты
- Вызывающий код не должен знать про изнанку реализации (и заодно становится чище и читаемее)

Инкапсуляция

Инкапсуляция

«Первое главное слово», один из базовых принципов ООП.

Варианты определения и трактовки:

- Упаковка данных и функций для работы с ними в единый компонент.
- Скрытие деталей реализации, ограничение доступа одних компонентов к другим.

В разных случаях могут иметь в виду только первое, только второе или оба пункта вместе.

Инкапсуляция

Базовый пример: стек, который мы только что разобрали.

Технически наш объектный код (класс стека):

- Упаковал данные и функции стека в одну сущность.
- Спрятал изнанку реализации от вызывающего.

Это и есть инкапсуляция.

Финальные ремарки

Что стоит запомнить из лекции

- ООП возникает тогда, когда кода становится много.
- Объектный код строится из классов. Класс – логическая сущность, содержит в себе поля и методы.
- Инкапсуляция – это когда вся логика некоторой сущности запаковывается внутрь этой самой сущности.

Где семинары?

- Группа 931 (Коротин) – 806 КПМ
- Группа 932 (Бирюков) – 324а ЛК
- Группа 933 (Лавриненко) – 319 ЛК
- Группа 934 (Васюков) – 706 КПМ
- Группа 935 (Бабичев) – 804 КПМ
- Группа 936 (Беклемышева) – 705 КПМ
- Группа 939 (Кулиев) – 807 КПМ

TO BE CONTINUED...