Аксиома № 1:

Передача осуществ-ся всегда по значению: (передача POD-ресурсов тривиальна) (типы данных языка)

```
int foo(int i, double d){
    return ec;
}
```

компилятор в месте вызова этой ф-ии обязан снять копию параметра и создать новый экз-р для вызывающего модуля (ф-ии)

```
int i = foo(j, f)
```

] j – double => компилятор кастерит int

аналогично для return

```
long l = foo(j, f)
```

└ будет создан экз-р по копии, которая вернулась по завершении ф-ии.

Передача ресурса — агрегата нетривиальна.

① ресурс уже проинициализирован в вызывающей стороне.

```
int foo (int * pInt , size_t _size ){
...
}
```

т.е. Аксиома: инварианта (!pInt && !_size) мы имеем доступ к памяти, но не знаем размер, поэтому _size. Но проверки на валидность — потеря времени.

```
// где-то в коде
int ia[3];
...
int ec = foo(ia)
```

```
int foo (int ia[3]) {
}
/* здесь получили int *  */
т.е. sizeof(ia) не сработает (потеря положения)
```

так не будет. Кто-нибудь поменяет размерность а компилятор этого не контролирует.

Более функ-ая аксиома: ВС не резиновая

- ```
  int foo (int* &ia){
  ...
  }
  ```

  данный модуль будет пользоваться ресурсом, принадлежащим вызыв-й стороне (имеет неограниченно делать, что угодно с перед-м ресурсом)

  ia = new int [10]; (утечка будет)
  решение: int foo (int * const & ia) { ... }

- ```
  int foo (int const & i) [... ]
  ```

  будет создаваться копия указателя на передаваемый объект

  это не передача через стек.

- ```
  int& foo (int i) {
      int * pInt = new int [10];
      return *pInt;
  }
  ```

  возврат по ссылке: выделена память на куче. Будет утечка памяти

  ```
  int i= foo(...);
  ```
  ```

```
int& foo (int i)
    int j;
    return j;
}
```

на стеке: нашего то уже почищена. что мы вернём? ↓.
ссылка на локальную переменную — ссылка на невалидный экз-р.

при перегрузке оператора operator [], нужно было бы возвращать по ссылке. (Но не будем, вдруг передастся число, такое, что мы выйдем из размерности, как мы предупредим пользователя)

В языке есть заготовка под UC:

```
// в uc_t.h
union uc_t {
        int * pInt;
        float * pFlt;
        double * pDBl;
};
```

```
...
int foo(uc_t uc){
...
}
```

```
// где-то в main
uc_t uc;
int rc = foo(uc);
```

Ошибки: 1) передача по значению
Решение: `int rc= foo(&uc);`
или:
```
uc_t foo (uc_t * uc){
    uc_t uc;
    /* инициализ */
    return uc;
}
```

Но лучше всего так:
```
int rc = foo (&uc);
int foo (uc_t *uc);
```

2) мы не знаем, какие данные хотсат в контейнере.

Решение: создать свой тип данных:

```
// uc-t.h

enum TYPES types {
    SHORT,
    INT,
    LONG
}
+ uc_t...
```

```
int foo (uc_t * uc, enum TYPES *t, size_t * p_size){
...
}
```

// uc_t.h

```
typedef struct tag_UC {

    size_t uc_size;
    enum TYPES uc_t;

    void *pData;
} UC;
```

```
int foo (struct UC * uc) {
...
}
```

потом компилятор подстроит указатели.

```
// в main

struct UC uc;

int rc = foo (&uc);
```

нет гарантии, что контейнер не исп-ся (как-то нужно определить поле, чтобы понять потом)

/* по мере работы с uc могут быть нарушены инварианты */
в конечный код хотелось бы перенести данные.

надо сократить реализацию. В публичном доступе остаётся только:

```
typedef struct tag_UC UC;
```

уже нельзя создавать экз-ры на стеке.

Фабричный метод

// uc.c

```
struct UC * create_UC (enum TYPES t, size_t _size,
                       void * pData);
```

// main

```
struct UC *pUC;

pUC = create_UC (INT, 10, NULL)
```

не знаем, где хотели инициализировать.

// uc.c

+ удаление

```
void delete_UC (struct UC **pUC)
```

Чтобы копии не хранились

(в C++ было бы struct UC &pUC)