

1. ~~Програмата~~ от имена (Namespace). Видове експресии и разлики.

Работа с инициализации, достъп до елементите, влагане, работа с функции, работа с масиви. Размер на обекти/инстанции. Подавяне/изваждане и отместване. Обединение. Endianess и проверка за little/big endian.

Namespace

- инструмент за избягване на конфликт на имена
- наредена двойка от скоби с дефиниции на тях

Тип: namespace ns {
 void f();
 int global=5;
}

main
ns::f();
↑
Оператор за резолюция

- using - ключова дума за добавяне на namespace в настоящия ~~current~~ scope

⚠ Конфликт

namespace A
f()

namespace B
f()

using A;
using B;
f(); // Compile time
A::f(); // v
B::f(); // v

⚠ анонимен namespace
- механизъм за рестриционе на видимостта на променливи, обекти, функции

Тип: namespace {
 void internal();

Енумератори

- тун, който може да има стойности със
от предварително ~~специални~~ ^{дифиниранни} константи (енумера-
тори)
- Енумератор свояства, като число
 - ако не е указано се избира предимо +1
 - ако на първи не е зададена стойност то тога е 0

Нп. 1) enum class

red, 110

blue, 111

orange, 112

3

2) enum + 2

a, 110

b, 111

c = 5, 115

d, 116

e = 4, 114

f, 115

g = a + b 111

3

Има 2 рода енумератори

I Unscoped

- енумераторите са глобални ~~на този~~ промен-
ливи, имената им са заборавени

Нп.: enum color

red,

orange

3

enum ~~fruit~~
~~orange~~

int ~~orange~~;

- имплицитно преобразование от енумератор
към чука

Up: enum color {
 red, 110
 orange 111};
enum animal {
 dog, 110
 cat 111};

3

if (color == red == animal::dog)

// то са юе е true

// имена числова розка

II scoped (enum class)

- променливите не са заоблачи
- имена имплицитно преобразуване

Up: enum ^{class} person {
 male,
 female};
enum ^{class} subject {
 oop,
 SDA,};

3

int n = person::male;

if (person::male == subject::oop) {

// compile time error

0

Разота с инициализации

- user defined
 - ноще обектност от имена, която се използва
- б определен пред

struct A {

 char a[10];

 int b;

 double c;

иинициализации

A obj;

A * obj; ptr = new A();

delete obj; ptr;

Дочети до елементите

obj-> +;

obj; ptr-> +;

3

Взаимодействие

```
struct B {
    double va;
    A a;
} int c;
```

3

Работа с ф-циями

сигнатура на ф-ции f (ф-ция, которую можно вызвать от где-либо)

1. конк - $f1(A \& obj)$	1 2 3 4 5 6
2. конст конк - $f2(const A \& obj);$	1 2 3 4 5 6
3. перф - $f3(A& obj);$	1 2 3 4 5 6
4. конст перф - $f4(const A& obj);$	1 2 3 4 5 6
5. указатель - $f5(A^* obj);$	1 2 3 4 5 6
6. конст указатель - $f6(const A^* obj);$	1 2 3 4 5 6

- Время жизни инициализации от ф-ций

- 1) $A f() \dots 3 \quad \checkmark$
 - 2) $A& f() \dots A \& obj; \quad \overbrace{\dots 3}^{\text{выход из края}} \text{ на scope-е} \quad \times$
 - 3) $A^* f() \dots A \& obj; \text{return } A \& obj; \quad \times$
- (2,3 работы за статичные объекты)
- 4) $A^* f() \dots A^* ptr = new A(); \quad \overbrace{\dots 3}^{\text{назначение}} \sim \checkmark$
(погрешно генерируется
заранее за избранным)
 - 5) $const A& f() \dots A \& obj; \quad \overbrace{\dots 3}^{\checkmark} \quad \text{(запись не вносится
на объект)}$

Работа с массивами

```
A arr[10];
A^* ptr = new A[n];
do Refe[3] ptr;
```

Размер на обекти / инициализация

Паметта, която можем да използваме:

1. Глобал (static) - записване на статичните глобални и константни променливи, една за употреба в програма

2. Stack - памет съдържаща локалните променливи
- малка, бърза, мененирана от OS (имаме контрол върху нея), LIFO, locality

3. Heap (динамичка) - заделя се по време на програмата
- нико locality (разположена из паметта), но дава, позволява заделящего на голями парчета памет, мененира се от потребителя (В С++)

sizeof

показва само byte-a в паметта където са хукни за да запазим посочения обект

`tp: struct T {
 int a;
 int b;}`

3

`sizeof(T) = 8;`

принципи на sizeof у нас са следните данни:
int - 4 bytes, double - 8 bytes, char - 1 byte, bool - 1 byte
pointer - 8 bytes (for x64) and 4 bytes for x86(32)

Инициализации на stack и heap

	stack	heap
int a = 1;	4	0
char b = 'a';	1	0
bool arr = new bool[1];	8	1
int m = new int*[10];	8	10 x 8 (pointer)
int arr = new int[100];	8	100 x 4 (int)
char test[3] = "abc";	4	0
int arr[40] = 203;	40 x 4 (int)	0

Подравняване и от欠缺

- В паметта ще бъде място да се дадат подравнявано
- В паметта съответната на структурата са предстаниени
така както са написани в кода.

- Структурата бива подравнена до най-добрата си
граница (или бърежка) структура (OS specific)

alignment requirement

- разницата на 2 последователни адреса на които можем да разположим една и съща

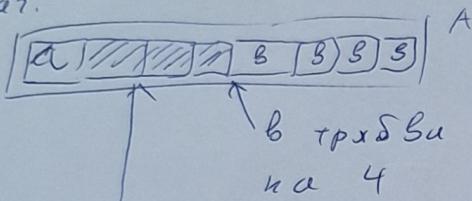
16p:
 $\text{alignof}(\text{int}) = 4$; // за примитивни типове alignment
 $\text{alignof}(T) = \text{sizeof}(T)$

16p:

```
struct A {
    char a;
    int b;
```

$\text{alignof}(A) = 4$;

△ Кодредба от машинната памет на компютър и съдържанието
 Думи:



В тяхното започва от адрес кратен
на 4

Образува се padding, когто е неизползван
空间 от кой задължен от програмата с
такъв структурата да е правилно подравнена

За да не оставим ненужна неизползвана
памет можем да:

△ char arr[] in struct

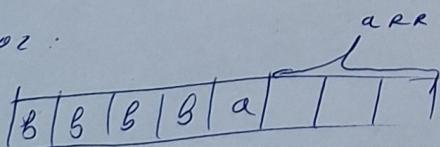
struct A Optimize {

int b;

char a;

char arr[]; // касае също също
и когто бъзможна място
в така:

Думи:

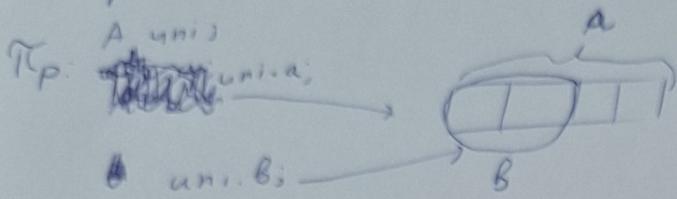


△ struct B 23

$\text{sizeof}(B) = 0$

Обединение

- последовательност от полета, използвайки една и съща памет
- user defined
- при работе от 1 деск. структуре



union A {

int a;

short b;

};

Endianess

- начин за подадене дане на бързотоце в DC

int

1	2	3	4
---	---	---	---

- big endianess

1	2	3	4
---	---	---	---

- little endianess - малък - младшият байт е във върх

4	3	2	1
---	---	---	---

Тип: проверка за little/big endian b increasing

union A {

int a;

short b;

};

main.cpp

~~union A;~~

A uni;

uni.a = 4;

if (uni.b == 4)

// 13 little endian

4	0	0	0
---	---	---	---

"при взимане на неправилни
112 трябва да е 4"

else

// 13 big endian

// ако е big endian uni ще е правилна

така

b	0	0	0	4
---	---	---	---	---

и uni.b = 0