

# Увод в програмирането

Сложни типове данни. Работа с динамичната памет

2017-2018 г.

ФМИ, специалност „Софтуерно инженерство“

# Съдържание

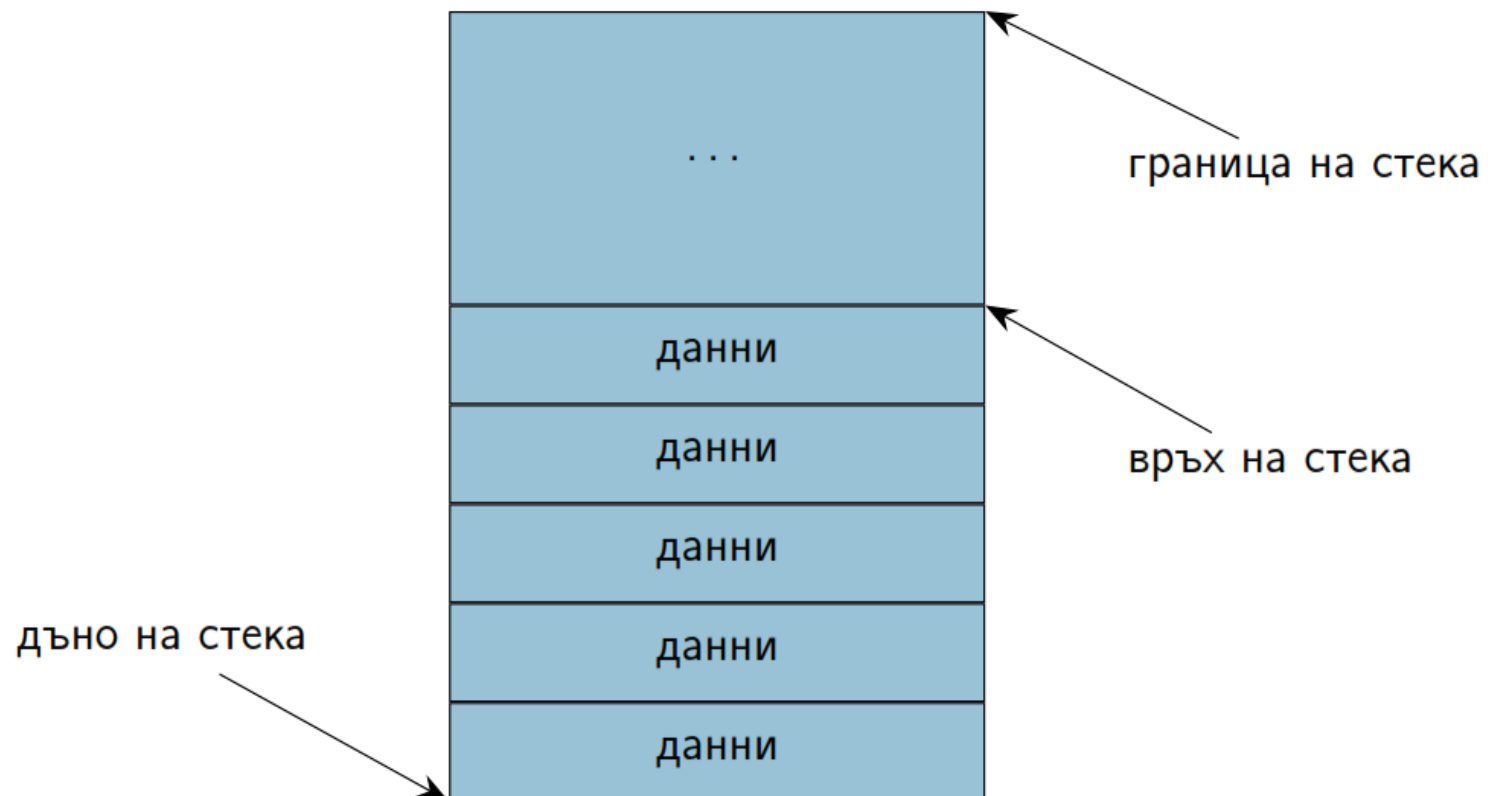
- Динамична памет (heap)
- Работа с динамичната памет
- Тип запис/структура (struct)

# Разпределение на Оперативната Памет (ОП)

- Най-общо се състои от:
  - програмен код,
  - област на статичните данни,
  - област на динамичните данни
  - програмен стек

Програмен стек
Динамична памет
Статична памет
Програмен код
ОП за работа с ОС

# Програмен стек



# Свойства на програмния стек

- Паметта се заделя в момента на дефиниция
- Всеки заделен блок памет носи името на променливата
- Паметта се освобождава при изход от блока (или функцията), в който е дефинирана променливата
- Последно заделената памет се освобождава първа
- Програмистът няма контрол над управлението на паметта
  - Паметта не може да се запази за по-дълго (след края на блока)
  - Паметта не може да се освободи по-рано (преди края на блока)
- Количеството заделена памет до голяма степен е определено по време на компилация
  - При какви случаи заделената памет може да варира по време на изпълнение?

# Област за динамична памет (heap)

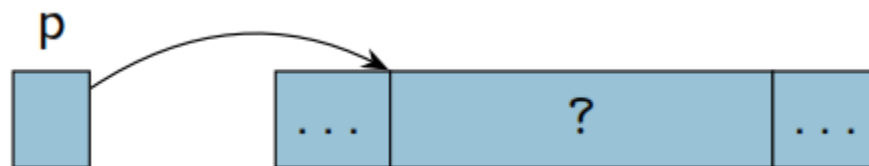
- Областта за динамична памет е набор от свободни блокове памет
- Динамична памет може да бъде заделена и освободена по всяко време на изпълнение на програмата
- Програмата може да заяви блок с различна големина
- Операционната система се грижи за управлението на динамичната памет
  - Поддържа “карта” кои клетки са свободни и кои заети
  - Контролира коя част от паметта от коя програма се използва (защитен режим)
  - Освобождава програмиста от задължението да знае къде физически се намира заделената от него памет

# Заделяне на динамична памет

- Заделянето на динамична памет става с операциите **new** и **new[]**
  - **new** <тип> – заделя блок от памет за една променлива от <тип>
  - **new** <тип>(<инициализация>) заделя блок от памет за една променлива от <тип> и я инициализира със зададените един или повече параметри
  - **new** <тип>[<брой>] заделя блок от памет за масив от <брой> елемента от <тип>
- Връща се указател към новозаделения блок от съответния <тип>

# Примери за заделяне на памет

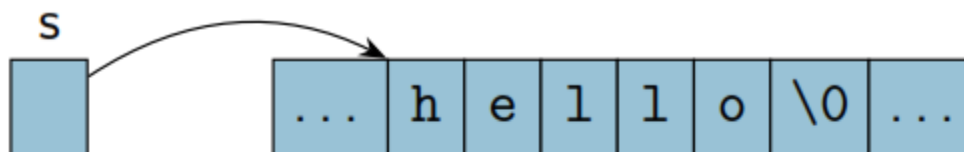
```
int* p = new int;
```



```
float* q = new float(1.23);
```



```
char* s = new char[6];  
strcpy(s, "hello");
```



```
char** ss = new char*(s);
```





# Освобождаване на памет

- Предварително заделена динамична памет се освобождава с операциите `delete` и `delete[]`
- `delete` <указател> освобождава блок от памет с начало, сочено от <указател>
- `delete`[<брой>] <указател> освобождава блок от памет, съдържащ масив от <брой> обекти, първият от които е сочен от <указател>
  - указването на <брой> не е задължително, понеже операционната система “знае” колко е голям заделения блок

# Особености при работа с динамичната памет

- На `delete` може да се подаде само указател, върнат от `new`
- Не е позволено освобождаването на памет в програмния стек или областта за програмен код
  - `int x; int* p = &x; ... delete p;`
  - ~~`delete sin; delete main;`~~
- Не е позволено частично освобождаване на памет
  - `int* a = new int[10]; ... delete (a+2);`
- Не е позволено използването на памет след като е освободена
- Не е позволено повторното освобождаване на една и съща памет
  - `int* p = new int[5], *q = p; delete p; q[1] = 5; delete q;`

# Особености при работа с динамичната памет

- Програмистът има контрол над заделянето на памет
- Програмистът носи отговорност за правилната работа с динамичната памет
- Заделената динамична памет остава непокътната до освобождаването ѝ с `delete` или до завършване на програмата
- След приключване на програмата, цялата заделена от нея памет се освобождава от операционната система

# Особености при работа с динамичната памет

- Честото заделяне и освобождаване на малки блокове памет води до т.нар. фрагментация



# Често срещани грешки

- Работа с указател към незаделена или освободена памет
- Освобождаване на произволна памет
- “Загубване” на указател към заделена памет
- Неосвобождаване на неизползвана памет
- Изтичане на памет (memory leak)

# Тип запис

# Структура от данни запис

- Логическо описание
  - Съставен тип данни,
  - Представа крайна редица от **фиксиран брой елементи**
  - Елементите може да са от **различни типове**
  - Достъпът до всеки елемент от редицата е пряк и се осъществява чрез име, наречено поле на записа.
- Физическо представяне
  - Полетата на записа се представят последователно в паметта.

# Дефиниране и използване на записи

- Един запис се определя чрез имената и типовете на съставлящите я полета.
- **Дефиниция на запис**

<дефиниция\_на\_запис> ::=

**struct** <име\_на\_запис>

{ <дефиниция\_на\_полета>;

{<дефиниция\_на\_полета>;}

};

<име\_на\_запис> ::= <идентификатор>

<дефиниция\_на\_полета> ::=

<тип> <име\_на\_поле>{, <име\_на\_поле>}

<тип> ::= <име\_на\_тип> | <дефиниция\_на\_тип>

<име\_на\_поле> ::= <идентификатор>



# Примери

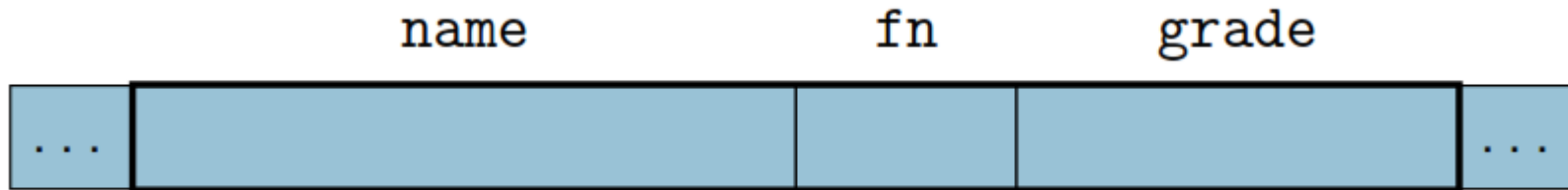
1. Координатите на точка в координатната система може да се зададат чрез запис с две полета.

```
struct Point{ double x, y; };
```

2. Данните за студент от една група (име, факултетен номер, среден успех) може да се зададат чрез запис с три полета.

```
struct Student {  
    char name[45];  
    int fn;  
    double grade;  
}
```

# Физическо представяне

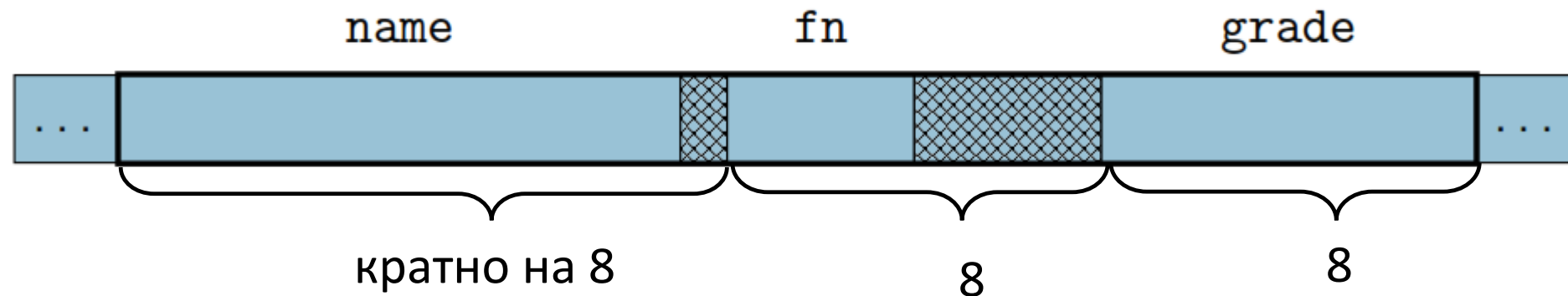


А дали е точно така?

`sizeof(Student)` = ?

# Подравняване (padding)

- Полетата в структурите се подравняват до адрес, кратен на големината им

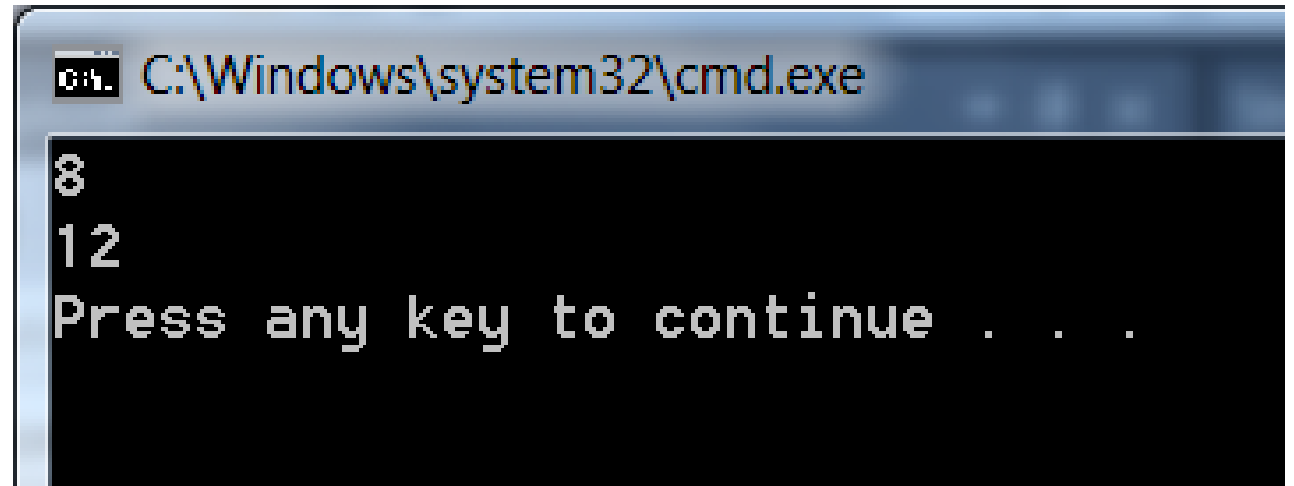


```
struct Student {  
    char name[45];  
    int fn;  
    double grade;  
}
```

`sizeof(Student) = 64`

# Малък експеримент

```
struct Point{ double x, y; };  
  
struct A {char c; char d; int i; };  
  
struct B {char c; int i; char d; };  
  
int main() {  
    cout << sizeof(A) << endl;  
    cout << sizeof(B) << endl;  
  
    return 0;  
}
```

A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Windows\system32\cmd.exe". The command prompt shows the output of a program: the number "8" on the first line, the number "12" on the second line, and the text "Press any key to continue . . ." on the third line.

```
C:\Windows\system32\cmd.exe  
8  
12  
Press any key to continue . . .
```

# Дефиниране на променливи от тип запис

```
[struct] <тип_запис> <име> [ = { <израз> {, <израз> } } ]  
{, <име> [ = { <израз> {, <израз> } } ] };
```

```
Student s1 = { "Петър Петров", 66666, 5.75 };
```

```
Point p1 = { 3.0, -0.6 };
```

# Операции над записи

- Присвояване (=)
  - Може да се присвояват само структури от един и същи тип
  - `Point p3 = p1; p3 = p2;`
  - ~~`Student s1 = p1;`~~
- Достъп до поле (.)
  - `<променлива>.<име_на_поле>`
  - `p1.x = 1.3; p2 = p1; p2.y = -p2.y;`
  - `s1.fn = 41000; cout << s1.grade;`
  - `char* s = s1.name;`
  - `int* p = &s1.fn;`
- Няма операции за вход и изход
  - ~~`cin >> s1;`~~
  - ~~`cout << p1;`~~

# Операции над записи

- Операциите над записи зависят от реализацията на езика. По стандарт за всяка реализация са определени следните операции и вградени функции:
- Над полетата на променливи от тип запис
  - Всяко поле на променлива от тип запис е от някакъв тип. Всички операции и вградени функции, допустими за данните от този тип, са допустими и за съответното поле.
- Над променливи от тип запис
  - Възможно е на променлива от тип запис да се присвои стойността на вече инициализирана променлива от същия тип запис или стойността на израз от същия тип.

# Масив от записи

```
Student s[10] = { { "Петър Петров", 80000, 5.5},  
                  { "Стефани Стефанова", 60000, 6 } };  
strcpy(s[2].name, "Иван Иванов");  
cout << s[1].fn;  
for(int i = 0; i < n; i++)  
    cin >> s[i].grade;
```



# Запис от записи

```
struct Team {  
    Student s1, s2;  
    char name[30];  
};
```

```
Team team = { { "Диана", 80003, 5 },  
              { "Радослав", 60004, 6}, "Дислав"};  
  
cout << team.name << ' ' << team.s2.name;  
  
double teamGrade = (team.s1.grade + team.s2.grade) / 2;
```

# Записите като полета

```
struct Employee {  
    char name[64];  
    Employee boss;  
};
```

```
Employee rector = { "Герджиков", NULL },  
dean = { "Първанов", &rector },  
dep_chair = { "Георгиева", &dean },  
lector = { "Димов", &dep_chair };
```

```
cout << lector.boss->boss->boss->name;
```

# Дефиниране и използване на структури ...

- **Указатели към записи**

Дефинират се по общоприетия начин.

- **Указател към запис**

<указател\_към\_запис> ::=

[**struct**] <име\_на\_запис> \* <променлива\_указател>  
[**= &** <променлива>];

където

<променлива> е от тип <име\_на\_запис>.

# Записи и указатели

```
Student* ps1 = &s1, *ps2 = NULL;  
ps2 = ps1; *ps2 = s2;  
Student& s3 = s1;  
cout << s3.name;  
s3 = s2;
```

# Достъп до поле на запис чрез указател

`<указател_към_запис> -> <поле>`

еквивалентно на `(*<указател_към_запис>).<поле>`

```
ps1->grade += 0.5;
```

```
cout << ps2->fn;
```

```
Team* pteam = &team; cout << pteam->s1.name;
```

# Записи и функции

- Записите като параметри
  - Предават се по стойност, като простите типове данни
  - Промените във функциите са локални
  - Записите може да се предават като параметри на функции също и по указател и псевдоним
- Записите като върнат резултат
  - Връщат се по стойност, като простите типове данни
  - Връща се копие на записа

- За подготовката на тази презентация са използвани слайдове на:
  - Доц. Александър Григоров
  - Доц. Трифон Трифонов