Записи

Трифон Трифонов

Обектно-ориентирано програмиране,
спец. Компютърни науки, 1 поток, 2021/22 г.

23 февруари – 2 март 2022 г.

Логическо описание

Записът е:

- съставен тип данни

Логическо описание

Записът е:

- съставен тип данни
- представя крайна редица от елементи

Логическо описание

Записът е:

- съставен тип данни
- представя крайна редица от елементи
- редицата е с фиксирана дължина

Логическо описание

Записът е:

- съставен тип данни
- представя крайна редица от елементи
- редицата е с фиксирана дължина
- елементите могат да са от различни типове

Логическо описание

Записът е:

- съставен тип данни
- представя крайна редица от елементи
- редицата е с фиксирана дължина
- елементите могат да са от различни типове
- произволен достъп до всеки елемент

Логическо описание

Записът е:

- съставен тип данни
- представя крайна редица от елементи
- редицата е с фиксирана дължина
- елементите могат да са от различни типове
- произволен достъп до всеки елемент

Дефиниция на запис

- `struct` <име> `{` <поле> { <поле> } `};`

```
int** p;          char& c;          int a[10];

double f(double);

double f(double x) { return x * x; }
```

Дефиниция на запис

- `struct` <име> `{` <поле> { <поле> } `};`
- <поле> ::= <тип> <идентификатор> {`,` <идентификатор> }`;`

Дефиниция на запис

- `struct` <име> `{` <поле> { <поле> } `};`
- <поле> ::= <тип> <идентификатор> {`,` <идентификатор> }`;`
- **Примери:**

Дефиниция на запис

- `struct` <име> `{` <поле> { <поле> } `};`
- <поле> ::= <тип> <идентификатор> {`,` <идентификатор> }`;`
- **Примери:**
- `struct Point { double x, y; };`

Дефиниция на запис

- `struct` <име> `{` <поле> `{` <поле> `}` `};`
- <поле> ::= <тип> <идентификатор> `{,` <идентификатор> `}`;
- **Примери:**
- `struct Point { double x, y; };`
- ```
  struct Student {
    char name[30];
    int year;
    double grade;
  };
  ```
  ```

Физическо представяне


```
struct Student {
  char name[30];
  int year;
  double grade;
};
```

- `sizeof(S)` — големина на записа `S`

Физическо представяне


```
struct Student {
  char name[30];
  int year;
  double grade;
};
```

- `sizeof(S)` — големина на записа S
- `sizeof(Point)` = ?

Физическо представяне


```
struct Student {
  char name[30];
  int year;
  double grade;
};
```

- `sizeof(S)` — големина на записа S
- `sizeof(Point)` = 16

Физическо представяне


```
struct Student {
  char name[30];
  int year;
  double grade;
};
```

- `sizeof(S)` — големина на записа `S`
- `sizeof(Point)` = 16
- `sizeof(Student)` = ?

Физическо представяне


```
struct Student {
  char name[30];
  int year;
  double grade;
};
```

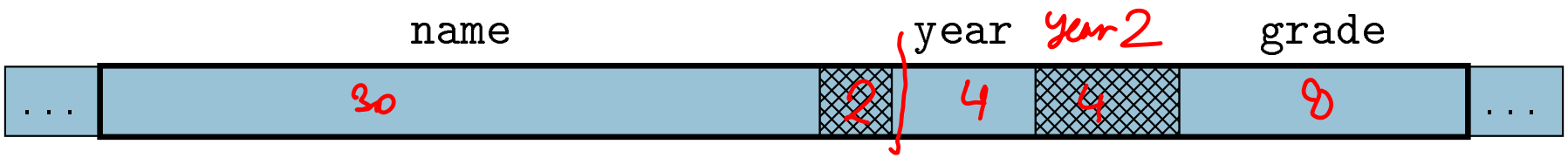
- `sizeof(S)` — големина на записа `S`
- `sizeof(Point)` = 16
- `sizeof(Student)` = 48

Физическо представяне


```
struct Student {
  char name[30];
  int year;
  double grade;
};
```

- `sizeof(S)` — големина на записа `S`
- `sizeof(Point)` = 16
- `sizeof(Student)` = 48
- Защо?

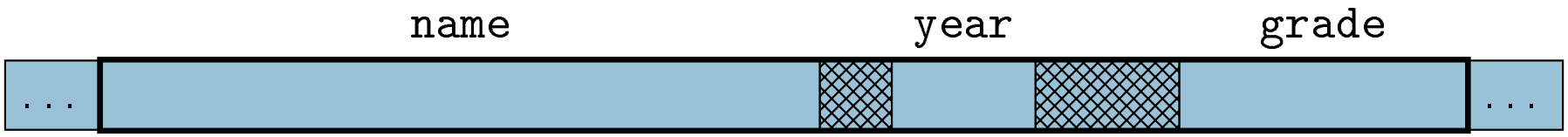
Физическо представяне




```
struct Student {
  char name[30];
  int year;
  double grade;
};
```

- `sizeof(S)` — големина на записа S
- `sizeof(Point)` = 16
- `sizeof(Student)` = 48
- Защо?
- Полетата в записите се подравняват до адрес кратен на големината им

Физическо представяне




```
struct Student {
  char name[30];
  int year;
  double grade;
};
```

- `sizeof(S)` — големина на записа `S`
- `sizeof(Point)` = 16
- `sizeof(Student)` = 48
- Защо?
- Полетата в записите се подравняват до адрес кратен на големината им
- Улеснява обработката от процесора

Променливи от тип запис

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

Променливи от тип запис

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

Примери:

- `Point p1, p2 = { 1.2, 3.4 };`

Променливи от тип запис

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

Примери:

- `Point p1, p2 = { 1.2, 3.4 };`
- `Student s1 = { "Иван Иванов", 1, 5.75 };`

Операции над записи

- присвояване (=)

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи от един и същи тип

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи от един и същи тип
  - `Point p3 = p1; p3 = p2;`

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи от един и същи тип
  - `Point p3 = p1; p3 = p2;`
  - ~~`Student s1 = p1;`~~

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи от един и същи тип
  - `Point p3 = p1; p3 = p2;`
  - ~~`Student s1 = p1;`~~
- достъп до поле (.)

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи от един и същи тип
  - `Point p3 = p1; p3 = p2;`
  - ~~`Student s1 = p1;`~~
- достъп до поле (.)
  - <променлива>.<име_на_поле>

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи 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;`

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи 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.year = 2; cout << s1.grade;`

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи 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.year = 2; cout << s1.grade;`
  - `cin.getline(s1.name, 30); s2 = s1;`

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи 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.year = 2; cout << s1.grade;`
  - `cin.getline(s1.name, 30); s2 = s1;`
  - `int* p = &s1.year;`

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи 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.year = 2; cout << s1.grade;`
  - `cin.getline(s1.name, 30); s2 = s1;`
  - `int* p = &s1.year;`
  - `char* s = s1.name;`

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи 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.year = 2; cout << s1.grade;`
  - `cin.getline(s1.name, 30); s2 = s1;`
  - `int* p = &s1.year;`
  - `char* s = s1.name;`
- няма операции за вход и изход

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи 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.year = 2; cout << s1.grade;`
  - `cin.getline(s1.name, 30); s2 = s1;`
  - `int* p = &s1.year;`
  - `char* s = s1.name;`
- няма операции за вход и изход
  - ~~`cin >> s1;`~~

Операции над записи

- присвояване (=)
  - могат да се присвояват само записи 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.year = 2; cout << s1.grade;`
  - `cin.getline(s1.name, 30); s2 = s1;`
  - `int* p = &s1.year;`
  - `char* s = s1.name;`
- няма операции за вход и изход
  - ~~`cin >> s1;`~~
  - ~~`cout << p1;`~~

Масив от записи

Можем да комбинираме свободно съставните типове данни, за да създаваме произволно сложни потребителски типове данни.

Масив от записи

Можем да комбинираме свободно съставните типове данни, за да създаваме произволно сложни потребителски типове данни.

- `Student s[10] = { { "Петър Петров", 2, 5.5},`
  `{ "Стефани Стефанова", 1, 6 } };`

Масив от записи

Можем да комбинираме свободно съставните типове данни, за да създаваме произволно сложни потребителски типове данни.

- `Student s[10] = { { "Петър Петров", 2, 5.5},`
  `{ "Стефани Стефанова", 1, 6 } };`
- `strcpy(s[2].name, "Иван Иванов");`

Масив от записи

Можем да комбинираме свободно съставните типове данни, за да създаваме произволно сложни потребителски типове данни.

- `Student s[10] = { { "Петър Петров", 2, 5.5},`
  `{ "Стефани Стефанова", 1, 6 } };`
- `strcpy(s[2].name, "Иван Иванов");`
- `cout << s[1].year;`

Масив от записи

Можем да комбинираме свободно съставните типове данни, за да създаваме произволно сложни потребителски типове данни.

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

Запис от записи

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

Запис от записи

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

- `Team team = { { "Диана", 2, 5 },`
  `{ "Радослав", 1, 6}, "Дислав"};`

Запис от записи

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

- `Team team = { { "Диана", 2, 5 },`
  `{ "Радослав", 1, 6}, "Дислав"};`
- `cout << team.name << ' ' << team.s2.name;`

Запис от записи

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

- `Team team = { { "Диана", 2, 5 },`
  `{ "Радослав", 1, 6}, "Дислав"};`
- `cout << team.name << ' ' << team.s2.name;`
- `double teamGrade = (team.s1.grade + team.s2.grade) / 2;`

Записи и функции

- записите като параметри

Записи и функции

- записите като параметри
  - предават се по стойност, като простите типове данни

Записи и функции

- записите като параметри
  - предават се по стойност, като простите типове данни
    - за разлика от масивите!

Записи и функции

- записите като параметри
  - предават се по стойност, като простите типове данни
    - за разлика от масивите!
  - промените във функциите са локални

Записи и функции

- записите като параметри
  - предават се по стойност, като простите типове данни
    - за разлика от масивите!
  - промените във функциите са локални
- записите като върнат резултат

Записи и функции

- записите като параметри
  - предават се по стойност, като простите типове данни
    - за разлика от масивите!
  - промените във функциите са локални
- записите като върнат резултат
  - връщат се по стойност, като простите типове данни

Записи и функции

- записите като параметри
  - предават се по стойност, като простите типове данни
    - за разлика от масивите!
  - промените във функциите са локални
- записите като върнат резултат
  - връщат се по стойност, като простите типове данни
  - връща се копие на записа

Задачи за записи

1. Да се въведе масив от студенти

Задачи за записи

1. Да се въведе масив от студенти
2. Да се изведат студентите в таблица

Задачи за записи

1. Да се въведе масив от студенти
2. Да се изведат студентите в таблица
3. Да се намери средния успех на всички студенти

Задачи за записи

1. Да се въведе масив от студенти
2. Да се изведат студентите в таблица
3. Да се намери средния успех на всички студенти
4. Да се подредят студентите по име

Указатели и препратки на записи

Можем да правим указатели и препратки на записите и техните полета

- `Student* ps1 = &s1, *ps2 = nullptr;`

Указатели и препратки на записи

Можем да правим указатели и препратки на записите и техните полета

- `Student* ps1 = &s1, *ps2 = nullptr;`
- `ps2 = ps1; *ps2 = s2;`

Указатели и препратки на записи

Можем да правим указатели и препратки на записите и техните полета

- `Student* ps1 = &s1, *ps2 = nullptr;`
- `ps2 = ps1; *ps2 = s2;`
- `Student& s3 = s1;`

Указатели и препратки на записи

Можем да правим указатели и препратки на записите и техните полета

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

Указатели и препратки на записи

Можем да правим указатели и препратки на записите и техните полета

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

Указатели и препратки на записи

Можем да правим указатели и препратки на записите и техните полета

- `Student* ps1 = &s1, *ps2 = nullptr;`
- `ps2 = ps1; *ps2 = s2;`
- `Student& s3 = s1;`
- `cout << s3.name;`
- `s3 = s2;`
- записите могат да се предават като параметри на функции по стойност, указател и препратка

Достъп до поле на запис чрез указател

<указател_към_запис> -> <поле>

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

Достъп до поле на запис чрез указател

<указател_към_запис> -> <поле>

- еквивалентно на (*<указател_към_запис>).<поле>
- ps1->grade += 0.5;

Достъп до поле на запис чрез указател

<указател_към_запис> -> <поле>

- еквивалентно на (*<указател_към_запис>).<поле>
- ps1->grade += 0.5;
- cout << ps2->year;

Достъп до поле на запис чрез указател

<указател_към_запис> -> <поле>

- еквивалентно на (*<указател_към_запис>).<поле>
- ps1->grade += 0.5;
- cout << ps2->year;
- Team* pteam = &team; cout << pteam->s1.name;

Рекурсивни записи

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

Рекурсивни записи

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

- записът се дефинира чрез себе си

Рекурсивни записи

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

- записът се дефинира чрез себе си
- `sizeof(Employee) = ?`

Рекурсивни записи

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

- записът се дефинира чрез себе си
- `sizeof(Employee)` = ?
- забранена рекурсия!

Рекурсивни записи — правилният начин

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

Рекурсивни записи — правилният начин

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

- записът се дефинира чрез себе си...

Рекурсивни записи — правилният начин

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

- записът се дефинира чрез себе си...
- ... но съдържа указател към себе си

Рекурсивни записи — правилният начин

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

- записът се дефинира чрез себе си...
- ... но съдържа указател към себе си
- `sizeof(Employee)` = ?

Рекурсивни записи — правилният начин

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

- записът се дефинира чрез себе си...
- ... но съдържа указател към себе си
- `sizeof(Employee)` = 72

Рекурсивни записи — правилният начин

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

- записът се дефинира чрез себе си...
- ... но съдържа указател към себе си
- `sizeof(Employee)` = 72
- ```
  Employee rector = { "Герджиков", nullptr },
           dean = { "Първанов", &rector },
           chair = { "Христова", &dean },
           lector = { "Трифонов", &chair };
  ```
  ```

Рекурсивни записи — правилният начин

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

- записът се дефинира чрез себе си...
- ... но съдържа указател към себе си
- `sizeof(Employee)` = 72
- ```
  Employee rector = { "Герджиков", nullptr },
           dean = { "Първанов", &rector },
           chair = { "Христова", &dean },
           lector = { "Трифонов", &chair };
  ```
- `cout << lector.boss->boss->boss->name;`

rector.boss = &lector;
  ```

Абстракция със структури от данни

Записите позволяват дефиниране на потребителски типове данни и операции над тях.

Абстракция със структури от данни

Записите позволяват дефиниране на потребителски типове данни и операции над тях.

Пример: Тип “рационално число”

Абстракция със структури от данни

Записите позволяват дефиниране на потребителски типове данни и операции над тях.

Пример: Тип “рационално число”

- **Логическо описание:** обикновена дроб

Абстракция със структури от данни

Записите позволяват дефиниране на потребителски типове данни и операции над тях.

Пример: Тип “рационално число”

- **Логическо описание:** обикновена дроб
- **Физическо представяне:** запис с числител и знаменател

Абстракция със структури от данни

Записите позволяват дефиниране на потребителски типове данни и операции над тях.

Пример: Тип “рационално число”

- **Логическо описание:** обикновена дроб
- **Физическо представяне:** запис с числител и знаменател
- **Базови операции:**
  - конструиране на рационално число
  - получаване на числител
  - получаване на знаменател

Абстракция със структури от данни

Записите позволяват дефиниране на потребителски типове данни и операции над тях.

Пример: Тип “рационално число”

- **Логическо описание:** обикновена дроб
- **Физическо представяне:** запис с числител и знаменател
- **Базови операции:**
  - конструиране на рационално число
  - получаване на числител
  - получаване на знаменател
- **Аритметични операции:**
  - събиране, изваждане
  - умножение, деление
  - сравнение

Абстракция със структури от данни

Записите позволяват дефиниране на потребителски типове данни и операции над тях.

Пример: Тип “рационално число”

- **Логическо описание:** обикновена дроб
- **Физическо представяне:** запис с числител и знаменател
- **Базови операции:**
  - конструиране на рационално число
  - получаване на числител
  - получаване на знаменател
- **Аритметични операции:**
  - събиране, изваждане
  - умножение, деление
  - сравнение
- **Приложни програми**

Абстракция със структури от данни

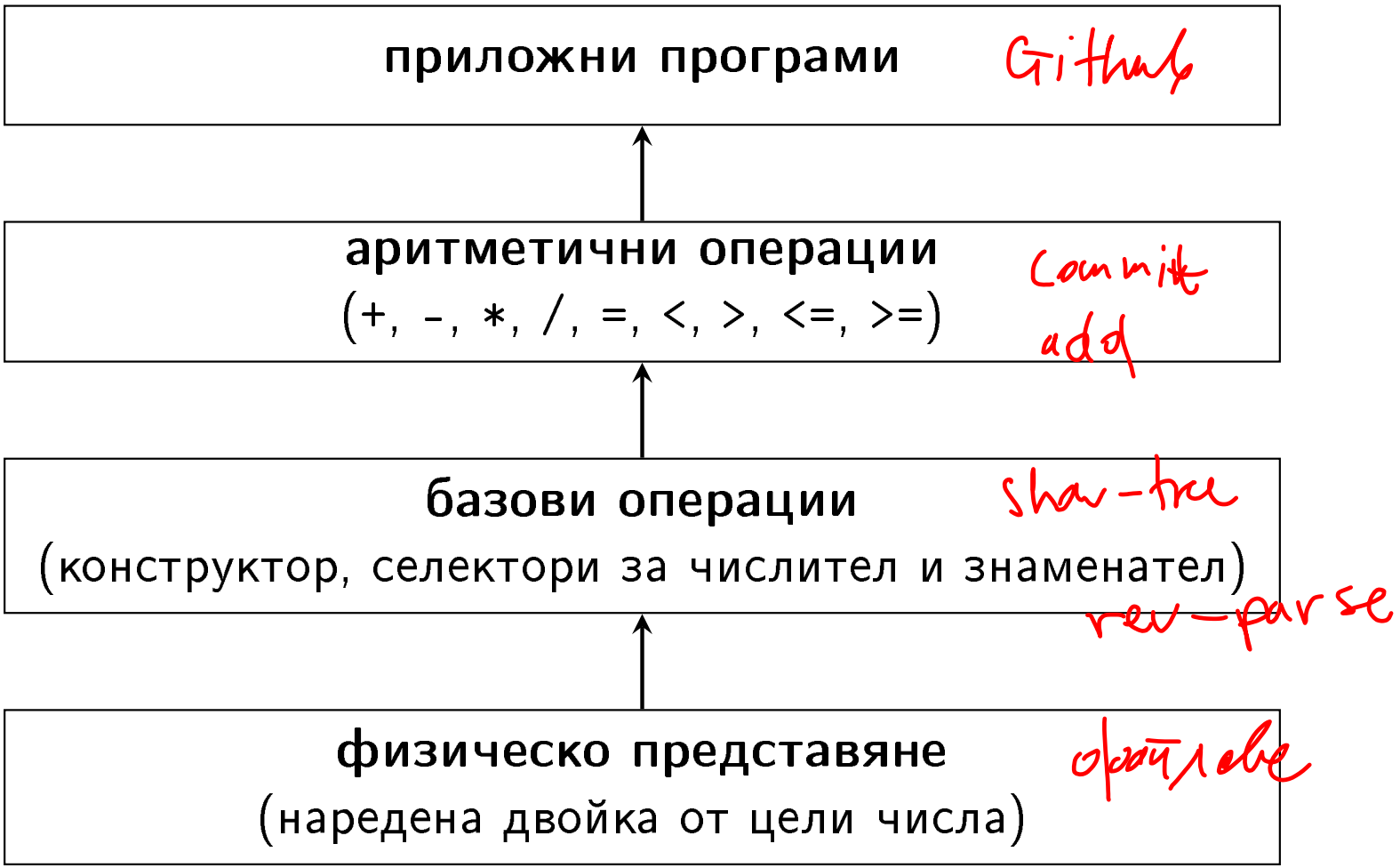
Записите позволяват дефиниране на потребителски типове данни и операции над тях.

Пример: Тип "рационално число"

- **Логическо описание:** обикновена дроб
- **Физическо представяне:** запис с числител и знаменател
- **Базови операции:**
  - конструиране на рационално число
  - получаване на числител
  - получаване на знаменател
- **Аритметични операции:**
  - събиране, изваждане
  - умножение, деление
  - сравнение
- **Приложни програми**

Идея: да изолираме вътрешното представяне на типа данни от операциите над него

Нива на абстракция



Обектно-ориентирано програмиране

- структуриране на данните на концептуално ниво като “обекти”

Обектно-ориентирано програмиране

- структуриране на данните на концептуално ниво като “обекти”
- обектите имат **свойства и методи** за работа с тях

Обектно-ориентирано програмиране

- структуриране на данните на концептуално ниво като “обекти”
- обектите имат **свойства и методи** за работа с тях
- могат да се дефинират **нива на достъп** до компоненти на обекта

Обектно-ориентирано програмиране

- структуриране на данните на концептуално ниво като “обекти”
- обектите имат **свойства и методи** за работа с тях
- могат да се дефинират **нива на достъп** до компоненти на обекта
- представянето на обекта обикновено се **капсулира**

Обектно-ориентирано програмиране

- структуриране на данните на концептуално ниво като “обекти”
- обектите имат **свойства и методи** за работа с тях
- могат да се дефинират **нива на достъп** до компоненти на обекта
- представянето на обекта обикновено се **капсулира**
- еднотипни обекти се описват чрез **класове** от обекти

Обектно-ориентирано програмиране

- структуриране на данните на концептуално ниво като “обекти”
- обектите имат **свойства и методи** за работа с тях
- могат да се дефинират **нива на достъп** до компоненти на обекта
- представянето на обекта обикновено се **капсулира**
- еднотипни обекти се описват чрез **класове** от обекти
- възможностите на обект могат да се разширяват (**наследяване**)

Обектно-ориентирано програмиране

- структуриране на данните на концептуално ниво като “обекти”
- обектите имат **свойства и методи** за работа с тях
- могат да се дефинират **нива на достъп** до компоненти на обекта
- представянето на обекта обикновено се **капсулира**
- еднотипни обекти се описват чрез **класове** от обекти
- възможностите на обект могат да се разширяват (**наследяване**)
- един обект може да има различни проявления (**полиморфизъм**)