ООП

гл.ас. д-р. Нора Ангелова

АЛГОРИТЪМ

Механизъм за намиране на решение, който е еднозначен, изпълним и завършващ, се нарича алгоритъм.

* *Съществуват вероятностни алгоритми, съдържащи елемент на случайност*

Поредица от стъпки, които водят до решаването на даден проблем.

АЛГОРИТЪМ

- Последователност от стъпки
- Диаграма (flowchart)

Step 1: Start

Step 2: Create a variable to receive the user's email address

Step 3: Clear the variable in case it's not empty

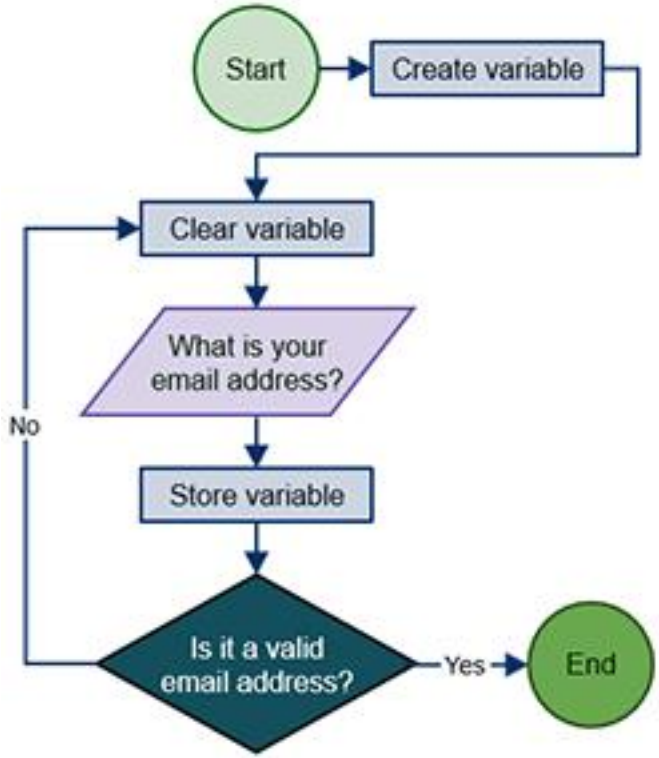
Step 4: Ask the user for an email address

Step 5: Store the response in the variable

Step 6: Check the stored response to see if it is a valid email address

Step 7: Not valid? Go back to Step 3.

Step 8: End



ПАРАДИГМА

Образец, ключов модел или метод за постигане на някаква цел

Парадигма най-общо означава модел на мислене.

ООП

Обектно-ориентираното програмиране (ООП) е парадигма в компютърното програмиране, при която една програмна система се моделира като набор от обекти, които взаимодействат помежду си.

ООП

Опишете обектите

Структура от данни

Под структура от данни се разбира организирана информация, която може да бъде описана, създадена и обработена с помощта на програма.

Определяне на структура от данни:

- логическо описание - описание на базата на декомпозицията ѝ на по-прости структури, а също на декомпозиция на операциите над структурата на по-прости операции.
- физическо представяне - дава методи за представяне на структурата в паметта на компютъра.

ЗАПИС

- Логическо описание

Съставна, статична, хетерогенна структура от данни.
Определя се като крайна редица от фиксиран брой елементи, които могат да са от различни типове.

Достъпът до всеки елемент е пряк и се осъществява чрез име, наречено **поле** на записа.

Запис

- Физическо представяне

Полетата на записа се представят последователно в паметта.

ДЕФИНИРАНЕ

```
struct <име_на_структура> {
      <дефиниция_на_полета>;
      {<дефиниция_на_полета>;}опц
};

<дефиниция_на_полета> ::= <тип><име>{,<име>}
<име_на_структура>, <име> ::= <идентификатор>
<тип> ::= <име_на_тип>|<дефиниция_на_тип>
```

ДЕФИНИРАНЕ

Примери

Множество от стойности

Всички крайни редици от по толкова елемента, колкото са полетата ѝ, като всеки елемент е от тип, съвместим с типа на съответното поле на структурата.

```
struct structName {
        int name1;
        double name2;
};
```

Всички двойки от вида:
{ int, double }

ДЕФИНИЦИЯ НА ПРОМЕНЛИВА

<дeф_променлива_от_тип_структура> ::=

<име_на_структура> <променлива>
[={<редица_от_изрази>}]опц

{,<променлива> [={<редица_от_изрази>}]опц}опц

<редица_от_изрази> ::=
<израз>|<израз>,<редица_от_изрази>

Пример
structName var1 = {10, 14.5}

ДОСТЪП

- Достъпът до полетата на структура е пряк
- Може да се осъществи чрез променлива от типа на структурата.
  Променлива и името на полето се разделят с оператора точка.

```
structName var1;
cout << var1.name1 << var1.name2;
```

* променливи от типа на полето и се наричат полета на променливата от тип структура или
член-данни на структурата!

ПАМЕТ

- Дефиницията на променлива от тип структура предизвиква заделяне на памет за всяко поле на променливата.

Всяко поле трябва да се разположи в паметта на адрес, който е равен на число кратно на размера на полето.

Пример:

```
struct example {
    char a;
    short int b;
    int c;
    char d;
};
```

Очакваме заделената памет за структурата да изглежда по следния начин:

Size of 1 block = 1 byte

Size of 1 row = 4 byte

a	b	b	c
c	c	c	d

ПАМЕТ

- Процесорът не може да достъпи паметта с равен размер на думите (4B - max size на полето)

Пример:

```
struct example {
    char a;
    short int b;
    int c;
    char d;
};
```

Реалното представяне

Size of 1 block = 1 byte

Size of 1 row = 4 byte

a	padding	b	b
c	c	c	c
d	padding	padding	padding

ПАМЕТ - ПОДРЕДБА

```
struct X {
  short s; /* 2 bytes *//* 2 padding bytes */
  int i; /* 4 bytes */
  char c; /* 1 byte */ /* 3 padding bytes */
};

struct Z {
  int i; /* 4 bytes */
  short s; /* 2 bytes */
  char c; /* 1 byte */ /* 1 padding byte */
};

const int sizeX = sizeof(struct X); /* = 12 */
const int sizeZ = sizeof(struct Z); /* = 8 */
```

Памет - подредба

- Полетата се сортират по тяхната големина в низходящ ред.
- Отстъпите (paddings) са позволени между данните за отделните полета и след последното поле.

Запис

- Имената на полетата в рамките на една дефиниция на структурата трябва да са различни идентификатори

```
struct structName {
      int name1;
      double name2;
};
```

ЗАПИС

- Възможно е име на запис, на негово поле и на произволна променлива в програмата да е един и същ идентификатор.
  НЕ ГО ИЗПОЛЗВАЙТЕ!

Запис

- Възможно е влагане на структури, т.е.
  поле на структура може да е от тип структура

```
struct Student {
      …
};

struct ClassRoom {
      Student maria;
};
```

Запис

- Не е възможно поле на структурата да е от тип, съвпадащ с името на структурата

```
struct Student {
      Student maria; // НЕ!!!
};
```

Какъв е размерът на Student ?

Запис

- Възможно поле на структурата да е от тип указател към името на структурата

```
struct Student {
      Student* maria; // OK
};
```

Какъв е размерът на Student ?

Запис

- Ако две структури трябва да се обръщат една към друга, е необходимо пред дефинициите им да се постави декларацията на втората по ред структура

```
struct StudentList;
struct Student {
    …
    StudentList* stl;
};

struct StudentList {
    …
    Student st;
};
```

Операции

- Над член-данните
- Над променливи от тип структура - присвояване на друга променлива или израз

УКАЗАТЕЛИ

<указател_към_структура> ::=

<име_на_структура>* <променлива_указател> [=&{<променлива>}]опц;

<променлива> ::= <име_на_структура>

```
structName var1;
structName* stPointer = &var1;
```

ДОСТЪП

```
structName var1;
structName* stPointer = &var1;

(*stPointer).name1

stPointer->name1
```

Абстракция

Идея: методите за използването на данните се разделят от тяхното представяне.

1. Всяка програма се проектира така, че да работи с „абстрактни данни“ – данни с неясно представяне.
2. Представянето на данните се конкретизира с помощта на множество функции – конструктури, селектори (гетъри), мутатори (сетъри), предикати.

Абстракция

Проектиране за работа с абстрактни данни

Задача:

Калкулатор за рационални числа.

n1/d1 * n2/d2 = n1*n2/d1*d2

1. Резултатът от умножението на рационални числа?
2. Създаване на рационално число
3. Извличане на числител на рационално число
4. Извличане на знаменател на рационално число
5. Функция за умножение на рациални числа

Абстракция

1. Създаване на рационално число
   `void makerat(rat& result, int n, int d);`
2. Извличане на числител на рационално число
   `int numerator(rat& r);`
3. Извличане на знаменател на рационално число
   `int denominator(rat& r);`
4. Функция за умножение на рациални числа

```
rat multRats(rat& r1, rat& r2) {
    rat r;
    makerat(
        r,
        numerator(r1)*numerator(r2),
        denominator(r1)*denominator(r2)
    );
    return r;
}
```

АБСТРАКЦИЯ

Конкретизация на представянето

```
struct rat {
    int num;
    int denom;
};

void makerat(rat& r, int n, int d) {
    r.num = n;
    r.denom = d;
}

int numerator(rat& r) {
    return r.num;
}

int denominator(rat& r) {
    return r.denom;
}
```

Абстракция

Абстрактен тип данни – тип данни, за който се изисква скриване на реализацията на типа и неговото „поведение“ се дефинира от множество от данни и множество от операции.

rat абстрактен тип данни ли е ?

АБСТРАКЦИЯ

```
struct rat {
    int num;
    int denom;

    // член-функции на rat
    void makerat(int n, int d);
    int numerator();
    int denominator();
    void printRat();
};
```

Достъп до член-функции

```
rat r;
r.makerat(1,5);
```

Абстракция

```
void rat::makerat(int n, int d) {
    num = n;
    denom = d;
}

int rat::numerator() {
    return num;
}

int rat::denominator() {
    return denom;
}
```

Достъп до член-данни

- Функцията за умножение на рационални числа не достъпва num && denom
- Ако се опитаме да ги достъпим

```
rat r;
cout << r.num; // опитът за достъп е успешен
```

- Може да се забрани чрез спецификатори за достъп

Спецификатори за достъп

- public – член-данните и член-функциите са достъпни за всяко функция, която е в обрастта на структурата.

  default
- private – член-данните и член-функциите са достъпни само за член-функциите.
- protected

Спецификатори за достъп

```
struct rat {
    private:
    int num;
    int denom;

    public:
    // член-функции на rat
    void makerat(int n, int d);
    int numerator();
    int denominator();
    void printRat();
};
```

ДОСТЪП

```
rat r;
r.makerat(1,5); // OK
cout << r.num; // NO
```

ООП

Това са идеите, които са в основата на ООП