*МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2010*
*MATHEMATICS AND EDUCATION IN MATHEMATICS, 2010*
*Proceedings of the Thirty Ninth Spring Conference of*
*the Union of Bulgarian Mathematicians*
*Albena, April 6–10, 2010*

# ВЪВЕЖДАНЕ НА РЕКУРСИЯТА ЧРЕЗ АБСТРАКЦИЯ И РЕДИЦИ ОТ ЗАДАЧИ

**Павел Азълов**

Рекурсията е мощно средство за описание на алгоритми. Тя е основна тема в почти всички уводни курсове по програмиране. Опитът показва, че рекурсията е трудна за обучаемите и това налага допълнително внимание при преподаването ѝ. В това е и акцентът на статията. Върху описания подход се предлага: (1) рекурсивните дефиниции да се записват чрез базови операции, които предварително се дефинират от преподавателя като множество от базови функции, и (2) упражненията върху рекурсията да се извършват с подходящи редици от задачи. Базовите операции са специфични за всяка конкретна редицата от задачи и с тях се осъществява плавен преход от рекурсивна дефиниция към рекурсивна функция. Използването на базови функции е вид функционална абстракция, ф която се скриват подробностите на изучавания език за програмиране. По този начин се създава възможност обучаемите да се концентрират основно върху формулирането на рекурсивната дефиниция.

**1. Въведение.** Рекурсивната природа на много от понятията в информатиката я прави важна тема с множество практически приложения. Ето защо, тя не случайно присъства в учебната програма на профил информатика [1, 2, 3] на средното училище, а и във всички университески програми на специалностите, свързани с информатиката [4, 5]. Важността на рекурсията не само като метод за програмиране, но и като начин на мислене, прилаган при дефининиране на нови понятия и алгоритми, е провокирала някои автори да направят предложения за изучаването ѝ преди масиви [7], дори и преди въвеждането на управляващите структури [8].

**Идея на подхода.** Накраткор идеята на подхода може да се опише така. При изграждане на рекурсивна функция най-напред се записва съответната ѝ рекурсивна дефиниция на език, близък до естествения. Словесното изразяване на рекурсивно решение на една задача се различава от записването ѝ на езика за програмиране. Тази разлика може по-лесно да се преодолее чрез въвеждане на "*междинен език*". Той представлява множество от функции, подходящо подбрани за конкретното множество от задачи. С помощта на тези функции, които тук се наричат *базови*, рекурсивната дефиниция се записва формално, лесно се разбира нейният смисъл и тя може непосредствено да се запише като рекурсивна C/C++ функция. Така чрез предварително изградено множество от операции, учениците изцяло се концентрират върху рекурсивната дефиниция, без да се разсейват от детайлите на нейната реализация. В случая се прилага *функционална абстракция*, с което се постига плавен преход от рекурсивна дефиниция към рекурсивна функция.

**2. Плавен преход от рекурсивна дефиниция към рекурсивна функция.** Да започнем с една примерна задача: *Да се напише рекурсивна програма, която проверява дали даден знаков низ е палиндром.*

*Палиндромът* е знаков низ, който, "прочетен" отляво надясно и от дясно наляво е един и същ. Палиндроми са например следните низове: `"abcdcba"`, `"ABBA"`, `""`, `"a"`. Последните два низа са съответно с дължини 0 (празният низ) и 1 (низ с един знак) и не са включени случайно. Това са примери за най-простите палиндроми, които представляват базовите (тривиалните) случаи на рекурсивната дефиниция. Съгласно по-горната дефиниция е ясно, че палиндромът е низ, чиито знаци са симетрични спрямо средата на низа. От тук следва, че ако първият и последният знак на даден низ $s$ са различни, то $s$ не е палиндром. Ако тези знаци са равни, низът $s$ ще бъде палиндром, тогава и само тогава, когато низът $s'$, получен от $s$ чрез отстраняването на първия и последния знак, е палиндром. Това подсказва рекурсивния характер на понятието и едновременно с това дава идея за формулиране на съответната рекурсивна дефиниция. Ето и самата дефиниция, записана с изрази от функции на класа string в C++:

**Дефиниция 1.**

$$\texttt{pal(s)} = \begin{cases} \texttt{true}, & \text{ако } \texttt{s.length( )} \leq 1; \\ \texttt{false}, & \text{ако } \texttt{s[0]} \neq \texttt{s[s.length( )} - 1\texttt{]}; \\ \texttt{pal(s.substr(1,s.length( )} - 2\texttt{))}, & \text{в останалите случаи.} \end{cases}$$

Записването на функцията в този вид изисква определено ниво на познаване на C++ класовете и специално на класа string. Ако един ученик може да запише Дефиниция 1, то той лесно ще запише и съответната C++ функция (Фиг. 1):

```
bool pal(string s)
{
  if (s.length() <= 1)
    return true;
  else if (s[0] != s[s.length() -1])
    return false;
  else
    return pal(s.substr(1, s.length() - 2));
}
```

Фиг. 1. C++ код на рекурсивната функция от Дефиниция 1

Не е реалистично да се очаква, че учениците ще запишат рекурсивната Дефиниция 1, когато те все още изучават езика за програмиране. Ето защо, със следващия вариант на формална дефиниция на палиндром скриваме (абстрахираме се от) тези подробности и я записваме на език близък до естествения:

**Дефиниция 2.**

$$\texttt{pal(s)} = \begin{cases} \texttt{true}, & \text{ако низът } \texttt{s} \text{ е празен или с дължина един знак} \\ & \text{(базови случаи);} \\ \texttt{false}, & \text{ако първият и последния знак на низа са различни;} \\ \texttt{pal(s}'\texttt{)}, & \text{където } \texttt{s}', \text{ е низът който се получава от } \texttt{s} \text{ чрез} \\ & \text{отстраняване на първия и последния знак;} \end{cases}$$

Очевидно този нов вариант на дефиницията е по-прост и по-ясен. Но ясно е също, че тази дефиниция се отличава твърде много от записа ѝ на C++ (Фиг. 1). По тази причина да запишем трети вариант, в който изрази като "*е празен*", "*първи елемент*", и "*последен елемент*" са заменени с обръщения към съответни функции, опериращи със знаци и низове. Така се достига до следната дефиниция:

**Дефиниция 3.**

$$\texttt{pal(s)} = \begin{cases} \texttt{true}, & \text{ако } \texttt{isEmpty(s)} \text{ или } \texttt{isEmpty(tail(s))} \\ & \text{е със стойност } \texttt{true} \\ \texttt{false}, & \text{ако } \texttt{first(s)} \neq \texttt{last(s)}; \\ \texttt{pal(body(s))}, & \text{в останалите случаи.} \end{cases}$$

където функцията:
`first(s)` – връща първия знак на низа `s`;
`last(s)` – връща последния знак на низа `s`;
`tail(s)` – връща низа `s` без първия му знак;
`body(s)` – връща низа `s` без първия и без последния знак;
`isEmpty(s)` – връща `true`, ако `s` е празен низ и `false` – в противен случай.

Този трети вариант на дефиницията има само положителните страни на предишните две дефиниции, които са следствие на въведената функционална абстракция. Това е така, защото:

• детайлите на езика C++ са скрити чрез въвеждането на петте функции, използвани в Дефиниция 3;

• от Дефиниция 3 лесно се преминава към C++ код, както се вижда от Фиг. 2.

```
bool pal(string s)
{
  if (isEmpty(s) | | isEmpty(tail(s)))
    return true;
  else if (first(s)!= last(s))
    return false;
  else
    return pal(body(s))
}
```

Фиг. 2. C++ код на рекурсивната функция от Дефиниция 3

Цялата процедура за преход от рекурсивна дефиниция към рекурсивна функция може да се резюмира в три стъпки:

(1) [БАЗОВИ ФУНКЦИИ ] Най-напред преподавателят създава модул от функции [4], който включва множеството от *базови функции*.

(2) [РЕКУРСИВНА ДЕФИНИЦИЯ] Решението на задачата се записва като рекурсивна дефиниция от типа на ДЕФИНИЦИЯ 3, в която се използват базови операции.

(3) [РЕКУРСИВНА ФУНКЦИЯ] Записване на рекурсивната дефиниция от тип ДЕФИНИЦИЯ 3 като C++ рекурсивна функция.

**3. Редици от задачи.** След създаването на базовите функции, с които се реализира функционална абстракция, може да се премине към формулиране на задачи, за които се предполага, че са в сила следните две условия:

• решенията на всички задачи изискват операции върху една и съща структура от данни. Това изискване е следствие от факта, че за решаването на задачите се използва едно и също множество от базови функции, които оперират върху една и съща структура от данни; • ако решението или идеята за решаването на задача $x$ може да се използва при решаването на задача $y$, задача $x$ трябва да прехожда задача $y$.

Множество от задачи с по-горните свойства наричаме *редица от задачи* [6]. Предимството на използването на редици от задачи е, че след решаване на една или няколко задачи, по аналогия учениците биха могли да продължат сами с останалите задачи. По-долу това е показно с конкретна редица от задачи.

Като пример да представим редицата от задачи: "*Низове и рекурсия*". Има поне две съображения за да се използват знакови низове, а именно:

• знаковият низ е структура от данни, която се възприема лесно от обучаемите;

• повечето от задачите за знакови низове могат лесно да се преформулират за масиви, линейни списъци и двоични дървета. Последните две структури от данни имат естествена рекурсивна природа.

**3.1. Множество от базови функции на редицата от задачи *Низове и рекурсия*.** Започваме с множеството от базови операции на редицата от задачи, в които се оперира със знакови низове. Всяка операция е записана с нейния прототип:

```
char first(string s);     // Връща първия знак на низа s
char last(string s);      // Връща последния знак на низа s
string head(string s);    // Връща низа s без неговия последен знак
string tail(string s);    // Връща низа s без неговия първи знак
string body(string s);    // Връща низа s без неговите първи и последен знак
bool isEmpty(string s);   // Проверява дали низът s е празен
char toChar(unsigned n);  // Трансформира десетична цифра в знак
unsigned toDigit(char c); // Трансформира знак в десетична цифра
```

Множеството от тези функции не е единствено. То не е нито минимално, нито независимо. Например функцията `body(s)` може да се изрази чрез две базови функции по следния начин: `tail(head(s))`. Ако въведем функцията *дължина* или използваме съответната функция от класа string, множеството от базови функции ще е друго.

Изборът на базовите функции е въпрос на лично предпочитание, но те трябва да са достатъчно прости и лесни за използване от обучаемите. Множеството от базови функции зависи до известна степен и от езика за програмиране. Предложенията по-горе примерни функции за езика C++ са почти същите и за езиците Pasca, Java и C#. С множеството от базови функции се разширява съответният език с няколко абстрактни операции върху знаци и знакови низове. Ние няма да коментираме тяхната реализация, която в този случай е тривиална. Ефективността на самите базови операции, както и ефективността на рекурсивните функции, които се очаква да бъдат написани, не са приоритет в тази тема.

**3.2. Конкретна редица от задачи на тема *Низове и рекурсия.*** По-долу е представена една конкретна редица от задачи за съставяне на рекурсивни функции, опериращи със знакови низове. За да се илюстрира подходът, описан по-горе, за три от задачите са дадени рекурсивните дефиниции и съответните им рекурсивни C++ функции. Предполага се, че най-напред се формулира рекурсивната дефиниция (лявата колонка), която след това се трансформира в рекурсивна функция (дясната колонка).

**S1.** Да се напише рекурсивна функция, с която се пресмята дължината на знаков низ.

$$\texttt{len(s)} = \begin{cases} 0, \text{ако } \texttt{isEmpty(s) = true} \\ 1 + \texttt{len(tail(s))}, \\ \quad \text{в противен случай} \end{cases}$$

```
int len(string s)
{
  if (isEmpty(s))
    return 0;
  else
    return 1+len(tail(s));
}
```

**S2.** Да се напише рекурсивна функция, с която се отпечатва даден знаков низ.

**S3.** Да се напише рекурсивна функция, с която се търси подниз в даден знаков низ.

**S4.** Да се напише рекурсивна функция, която проверява дали всички знаци на знаков низ са десетични цифри.

**S5.** Да се напише рекурсивна функция, с която в даден знаков низ се намира знакът с най-голям ASCII код.

**S6.** Да се напише рекурсивна функция, която инвертира (обръща) даден знаков низ.

$$\texttt{reverse(s)} = \begin{cases} \texttt{s}, \text{ ако } \texttt{len(s)} \leq 1 \\ \texttt{last(s)+reverce(body(s))+} \\ \texttt{first(s)}, \text{ в противен случай} \end{cases}$$

```
string reverse(string s)
{
  if (len(s) <= 1)
    return s;
  else
    return last(s) + reverse(body(s))
    + first(s);
}
```

**S7.** Да се напише рекурсивна функция, която реализира операцията *конкатенация* на два знакови низа.

**S8.** Да се напише рекурсивна функция, която копира подниз на даден знаков низ.

**S9.** Да се напише рекурсивна функция, която трансформира цяло число без знак в знаков низ.

**S10.** Да се напише рекурсивна функция, която трансформира даден знаков низ в цяло число без знак. Приема се, че всички знаци на низа са десетични цифри.

**S11.** Да се напише рекурсивна функция, с която се проверява дали даден знаков низ е идентификатор. Приема се, че идентификаторът е низ, чиито първи знак е буква, а останалите, ако има такива, са букви и/или цифри.

```
isIde(s)= ideBegin(s) && ideTail(s)
```

```
bool isIde(string s)
{
  return ideBegin(s) && ideTail(s);
}
```

$$\texttt{ideBegin(s)} = \begin{cases} \texttt{false}, \text{ ако } \texttt{isEmpty(s) = true}; \\ \texttt{true}, \text{ ако } \texttt{first(s)} \text{ е буква}; \\ \texttt{false}, \text{ ако } \texttt{first(s)} \text{ не е буква} \end{cases}$$

```
bool ideBegin(string s)
{
  if(Empty(s))
    return false;
  else if (islower(first(s)) | |
           isupper(first(s)))
    return true;
  else
    return false;
}
```

$$\texttt{ideTail(s)} = \begin{cases} \texttt{true}, \text{ ако } \texttt{isEmpty(s) = true}, \\ \texttt{false}, \text{ ако } \texttt{first(s)} \text{ не е буква} \\ \qquad\qquad \text{нито цифра}; \\ \texttt{ideTail(tail(s))}, \\ \qquad\qquad \text{в противен случай} \end{cases}$$

```
bool ideTail(string s)
{
 if (isEmpty(s))
   return true;
 else if (!isalnum(first(s)))
   return false;
 else
   return ideTail(tail(s));
}
```

Представената редица от задачи за знакови низове може лесно да се преформулира в редица от задачи за масиви от знаци (C низове), масиви от числен тип, линийни списъци и двоични дървета. Много от базовите операции са идентични или сходни по смисъл операции. Естествено, тяхната реализация, която се извършва от преподавателя, ще е различна, но тя остава скрита (за обучаемите) и не е необходимо да се разглежда по време на темата за рекурсия.

**4. Заключителни бележки.** В настоящата статия се описва опит от преподаването на рекурсия в класове, в които обучаемите са без предварителна подготовка и при силно ограничен обем от часове. Представената идея може да се използва както в класове от средното училище, така и в университетски курсове. С разгледания пример се подсказва как се изграждат редици от задачи, подходящи за уводен курс по програмиране, а също и за курс по структури от данни, в които наред с абстракцията с данни се прилага и функционална абстракция. Много от изучаваните структури от данни имат рекурсивна природа и реализацията им с рекурсивни операции е съвсем естествена. Примерите в тази статия са записани на езика C++, но подходът е независим от конкретния език за програмиране и може директно да бъде приложен при изучаване на други езици като например Pascal, Java или C#.

## ЛИТЕРАТУРА


[1] Учебни програми III част за задължителна и профилирана подготовка IX, X, XI XII клас (математика, информатика и информационни технологии). Главна редакция на педагогическите издания към МОН, София, 2003.

[2] П. Азълов. Информатика за 9.–10. клас. Профилирана подготовка. Езикът C++ в примери и задачи. София, Просвета, 2005.

[3] П. Азълов, Ф. Златарова, М. Тодорова. Информатика за 10. клас. Профилирана подготовка. София, Просвета, 2003.

[4] П. Азълов. Обектно-ориентирано програмиране. Структури от данни и STL. София, Сиела, 2008.

[5] М. Тодорова. Програмиране на C++, част I. София, Сиела, 2009.

[6] P. Azalov, F. Zlatarova. Teaching Programming through Successive Problem Transformations. *JCSC*, **18** (2003), No 4, 175–182.

[7] K. Bruce, A. Danyluk, T. Murtagh. Why Structural Recursion Should Be Taught Before Arrays in CS1. SIGCSE, 2005, 23–27.

[8] S. Bergman. Teaching Recursion before Control Structures and Functions in CS1. *JCSC*, **18** (2003), No 3, 210–216.



Павел Азълов
Пенсилвански Държавен Университет
Пенсилвания, САЩ
e-mail: pka10@psu.edu


## INTRODUCING RECURSION THROUGH ABSTRACTION AND SEQUENCES OF PROBLEMS


**Pavel Azalov**



Recursion is a powerful technique for producing simple algorithms. It is a main topics in almost every introductory programming course. However, educators often refer to difficulties in learning recursion, and suggest methods for teaching recursion. This paper offers a possible solutions to the problem by (1) expressing the recursive definitions through base operations, which have been predefined as a set of base functions and (2) practising recursion by solving sequences of problems. The base operations are specific for each sequence of problems, resulting in a smooth transitions from recursive definitions to recursive functions. Base functions hide the particularities of the concrete programming language and allows the students to focus solely on the formulation of recursive definitions.