Wstęp do algorytmów rekurencyjnych

O rekurencji mówimy wówczas, gdy definicja jakiegoś pojęcia odwołuje się do niego samego. Z logicznego punktu widzenia poprawna definicja pojęcia nie może odwoływać się do niego samego (właśnie to wyśmiewa potoczne powiedzenie: „masło maślane“), a tylko do pojęć, uznanych za nie wymagające definicji (tzw. pojęć pierwotnych), lub pojęć zdefiniowanych wcześniej. W pewnych sytuacjach jednak, po nałożeniu odpowiednich ograniczeń, możliwa jest definicja pojęcia na bazie niego samego. Jak może wyglądać taka definicja, dowiemy się analizując następujące zagadnienie. Liczba, zwana silnią (z liczby naturalnej) określona jest następująco:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \quad , gdzie\ n \in \mathbb{N}$$

Ponieważ silnia z liczby *n* jest równa ilości permutacji zbioru *n-elementowego* (ilości sposobów, w jaki można ustawić wszystkie elementy zbioru w szereg, w różnej kolejności), wygodnie jest rozszerzyć definicję silni przyjmując, że silnia z zera wynosi 1 (zbiór pusty daje tylko jedną możliwość ustawienia jego „elementów“). Tak rozszerzoną definicję silni można wyrazić rekurencyjnie w następujący sposób:

$$n! = \begin{cases} 1 & , dla\ n = 0 \\ n * (n-1)! & , dla\ n > 0 \end{cases}$$

W związku z powyższą definicją silni można mieć dwa pytania:

- gdzie jest odwoływanie się pojęcia silni do niego samego?
- dlaczego taka definicja jest poprawna?

Druga linia powyższej definicji to stwierdzenie, że silnia z pewnej liczby *n* może być wyznaczona na podstawie silni z liczby *n-1*. Definicja pojęcia silni odwołuje się zatem do pojęcia silni, a to jest cecha definicji rekurencyjnej. Pozostaje wyjaśnienie, dlaczego taka definicja jest poprawna. Jest taka dlatego, że definicja silni z *n* co prawda opiera się o definicję silni, ale z innej liczby (konkretnie z *n-1*), a w tzw. *przypadku brzegowym* – jest nim pierwsza linia definicji, określająca, że silnia z zera, będącego najmniejszą liczbą naturalną, jest równa 1 – wartość silni dana jest z definicji. Można ponadto zauważyć, że skoro dana jest z definicji wartość silni z zera, a silnie z wszystkich większych od zera liczb można wyznaczyć na podstawie znajomości silni z liczby o jeden mniejszej, to taka definicja, zgodnie z zasadą indukcji matematycznej, generuje nieskończony ciąg liczb – wartości silni z *n*. Konsekwencją odwoływania się definicji silni do pojęcia silni z liczby o jeden mniejszej jest to, że aby obliczyć silnię z *n* większego od zera, musimy wyznaczyć kolejno silnie z *n-1*, *n-2* (potrzebną do wyznaczenia silni z *n-1*), n-3 itd. aż dojdziemy do wyznaczenia silni z zera, która jest dana z definicji. Zaletą algorytmów rekurencyjnych jest to, że bardzo prosto dają się zapisać w językach programowania, które dopuszczają wywołanie funkcji przez samą siebie. C++ jest tego typu językiem. Spójrzmy na listing (plik *silnia.cpp*):

```
#include<iostream>
using namespace std;

unsigned long silnia(unsigned long);

int main()
{
unsigned long argument;

   cout<<"Program wylicza silnie z liczby algorytmem rekurencyjnym"<<endl;
   cout<<"Podaj liczbe: ";
   cin>>argument;
   cout<<"Silnia z liczby "<<argument<<" wynosi "<< silnia(argument) <<endl;
}

unsigned long silnia(unsigned long liczba)
{
```

```
   if(liczba==0) return 1;
   else
   return liczba * silnia(liczba-1);
}
```

W powyższym listingu pojawia się nowy typ zmiennej, a mianowicie *unsigned long*. Do tej pory używaliśmy jako zmiennych, przechowujących liczby całkowite wyłącznie zmiennych typu *int*. Istnieje sześć typów całkowitoliczbowych w C++, tworzone są one następująco:

- najpierw podaje się, czy liczba ma być „ze znakiem" – *signed* , czy „bez znaku" – *unsigned*. Jeśli nie podamy żadnego z tych dwóch modyfikatorów, domyślnie przyjmuje się modyfikator *signed* (liczba ze znakiem charakteryzuje się możliwością zapisania liczb ujemnych, zera i liczb dodatnich, liczba bez znaku umożliwia zapisanie wyłącznie zera i liczb dodatnich).
- następnie podaje się opcjonalnie, czy chcemy „krótszą" postać niż typ *int* (mniej bajtów pamięci, co oznacza mniejszy zakres liczb, możliwych do zapisania) – *short* lub „dłuższą" – *long* (więcej bajtów itd.).
- Na koniec pozostaje słowo *int*, oznaczające liczbę całkowitą.

Z powyższych trzech elementów (*signed/unsigned*, *short/long*, *int*) musi pojawić się co najmniej jeden. Jeśli tym jedynym elementem *nie jest* słowo *int*, to możemy (choć nie musimy) je pominąć. Przykładowo użyty w programie typ *unsigned long* to liczba długa bez znaku (*int* pominięto).

Następny przykład definicji rekurencyjnej, który zostanie omówiony, to sposób określenia formatu standardowych papierów serii „*A*". Rozmiary tych papierów, o oznaczeniach od *A0* do *A7*, określone są rekurencyjnie w następujący sposób: format *A0* ma rozmiar 840 na 1188 mm, zaś aby przejść na numer o jeden większy, należy podzielić większy z wymiarów (czyli szerokość lub wysokość arkusza) przez dwa. Poniższy program wypisuje na ekran wielkość wszystkich arkuszy papieru formatów *A*.

```
#include<iostream>
using namespace std;

bool papier(unsigned numer, double &szerokosc, double &wysokosc);

int main()
{
   cout<<"Program wyznacza wielkosci znormalizowanego papieru"<<endl;
   cout<<"Format\tSzerokosc\tWysokosc"<<endl;
   for(unsigned i=0; i<=7; i++)
   {
   double a,b;

         if( papier(i, a, b) )
         cout<<"A"<<i<<"\t"<<a<<" mm\t\t"<<b<<" mm"<<endl;
   }
}

bool papier(unsigned numer, double &szerokosc, double &wysokosc)
{
   if(numer==0)
   {
         szerokosc=840;
         wysokosc=1188;
         return true;
   }
   else if(numer<=7)
   {
   double szer,wys;

         papier(numer-1, szer, wys);
         if(szer>wys) szer/=2; else wys/=2;
         szerokosc=szer;
```

```
        wysokosc=wys;
        return true;
   }
   else return false;
}
```

Program w trakcie pracy wyświetla na ekranie:

```
Program wyznacza wielkosci znormalizowanego papieru
Format Szerokosc          Wysokosc
A0     840 mm             1188 mm
A1     840 mm             594 mm
A2     420 mm             594 mm
A3     420 mm             297 mm
A4     210 mm             297 mm
A5     210 mm             148.5 mm
A6     105 mm             148.5 mm
A7     105 mm             74.25 mm
```

Nowym elementem języka C++, który pojawił się w programie jest tzw. przekazywanie (parametrów do funkcji) przez referencję. Spójrzmy na prototyp:

```
void papier(unsigned numer, double &szerokosc, double &wysokosc);
```

Argumenty *szerokość* oraz *wysokość* poprzedzone są znakiem & (ang. *ampersand*). Co spowoduje takie zdefiniowanie argumentów? Przekazanie zmiennej przez referencję pozwala zmienić jej wartość z wnętrza funkcji, do której ją wysyłamy jako argument. Jak powyższy mechanizm wykorzystywany jest w omawianym programie? Oto fragment funkcji:

```
{
double szer,wys;

        papier(numer-1, szer, wys);
        i.t.d.
}
```

Po wywołaniu funkcji *papier*, w zmiennych *szer* oraz *wys* będą wpisane rozmiary papieru A (*numer - 1*), zapisane tam przez wywołaną funkcję. Jeśli zamiast przekazywania argumentów przez referencję użyte zostałoby przekazywanie argumentów przez wartość, funkcja przez nas wywołana nie byłaby w stanie zmienić wartości zmiennych *szer* ani *wys*. Argumenty są zmiennymi dostępnymi tylko dla funkcji, do których na etapie jej wywołania wpisywane są konkretne wartości, zaprogramowane w miejscu wywołania. Do tej pory w funkcjach, mimo iż jest to dozwolone, nie próbowaliśmy przypisywać nic do argumentów funkcji. Argumenty są pełnoprawnymi zmiennymi, więc można dokonywać na nich przypisania, choć zazwyczaj się tego unika, by nie stracić oryginalnych wartości argumentów. Jednakże w poniższym wywołaniu:

```
int silnia(int k);

int  main()
{
int m, n;

        n=silnia(m);
}
```

nawet jeżeli spróbujemy przypisać wartość do argumentu funkcji *k* wewnątrz funkcji *silnia*, nie uda się nam zmienić wartości zmiennej *m, znajdującej się w* funkcji *main()* - zmieni się jedynie wartość *k*. Jeżeli jednak funkcja *silnia* byłaby zadeklarowana następująco (znak & musi pojawić się również w definicji funkcji):

```
int silnia(int &k);
```

to przypisanie wartości referencji *k* wewnątrz funkcji *silnia* zmieni wartość zmiennej *m* w funkcji

main()! Tę technikę należy stosować ostrożnie, gdyż może prowadzić do powstania trudnych do wykrycia błędów. Często stosuje się ją do zwracania przez funkcję kilku wartości, tak jak w naszym przypadku – chcemy zwrócić szerokość i wysokość arkusza papieru, zaś z funkcji można zwrócić tylko jedną wartość (jakkolwiek można zwrócić z funkcji tzw. strukturę, która zazwyczaj zawiera w sobie wiele informacji, jest to traktowane nadal jako zwrot jednej wartości złożonego typu języka), więc wykorzystujemy argumenty funkcji, których może być wiele, jako „drogę“ zwrócenia informacji na zewnątrz. Dla porządku dodam, że istnieje również trzeci sposób wysyłania argumentów do funkcji, tzw. wysyłanie przez wskaźnik, który również pozwala na zmianę wartości zmiennych, znajdujących się poza funkcją.

Za wyjątkiem przekazywania argumentów przez referencje, funkcja realizuje w możliwie prosty sposób swoje działanie definicyjne, polegające na podzieleniu dłuższego z rozmiarów papieru o mniejszym o jeden indeksie przez dwa i zwróceniu wyników przez referencje lub zwrócenie wymiarów formatu *A0*.

Więcej informacji na temat referencji i ich używaniu znajda Państwo w rozdziale 9. książki „C++ dla każdego”, autorstwa Jesse Liberty.