2
Proste typy danych

Zgodnie z równaniem Niklausa Wirtha [2]:

$$\text{Algorytmy} + \text{Struktury danych} = \text{Programy}$$

programy operują na danych.

Dane można podzielić na proste – będące jedną wartością, oraz złożone – składające się z więcej niż jednej wartości.

Przykładem prostej danej jest liczba całkowita reprezentująca numer albumu studenta, natomiast tablica zawierająca dane o wszystkich studentach danego roku to przykład danej złożonej.

W tym rozdziale rozpatrywać będziemy *proste dane* opisane *typami prostymi*.

2.1 Pojęcie typu

W językach programowania *typ* opisany jest przez podanie następujących trzech zbiorów:

wartości zbiór wszystkich wartości jakie dostępne są w danym typie,

operacje zbiór wszystkich operacji jakie można wykonywać na wartościach danego typu,

relacje zbiór wszystkich relacji jakie można sprawdzać między wartościami danego typu.

2.2 Typ bool

C99

Począwszy od roku 1999, w pliku nagłówkowym `stdbool.h` zdefiniowany jest *typ bool*,[1] o dwuelementowym zbiorze wartości, na który składają się stałe `true`

[1] Precyzyjniej, `bool` jest tylko aliasem nazwy `_Bool` oznaczającej dostępny w standardzie C99 typ boolowski (patrz Dodatek B.1).

Tabela 2.1: Operacje logiczne

Operator	**Operacja logiczna**
`&&`	koniunkcja
`||`	alternatywa
`!`	negacja

i `false`.

Stała `true` reprezentuje logiczną wartość prawdy, natomiast `false` logiczną wartość fałszu.

Dzięki zdefiniowaniu w standardzie C99 typu `bool` każde dwie wartości oznaczające prawdę są sobie równe. Wcześniej tak nie było, bo każda wartość całkowita różna od zera mogła być interpretowana jako stała oznaczająca logiczną wartość prawda.

W Tabeli 2.1 zebrano operacje na wartościach logicznych jakie dostępne są w języku C.

Koniunkcja jest prawdziwa tylko gdy jej oba argumenty są prawdziwe:

`p`	`q`	`p && q`
false	false	false
false	true	false
true	false	false
true	true	true

Alternatywa jest fałszywa tylko gdy jej oba argumenty są fałszywe:

`p`	`q`	`p || q`
false	false	false
false	true	true
true	false	true
true	true	true

Koniunkcja jest prawdziwa tylko gdy jej argument jest fałszywy:

`p`	`! p`
false	true
true	false

Wartości logiczne można między innymi porównywać czy są równe (relacja `==`) albo różne (relacja `!=`).

2.3 Typ char

Język programowania C, tak na prawdę, nie ma typu do przechowywania znaków jakie znamy np. z języków Pascal czy Ada. W języku C dostępny jest typ `char`, którego wartości są liczbami całkowitymi i zajmują w pamięci komputera

jeden bajt. Najczęściej typ ten jest typem ze znakiem tj. zakresem jego wartości są liczby całkowite od −128 do 127.

W języku C dostępne są literały znakowe pisane między znakami cudzysłowia. Wartościami ich są odpowiednie kody tych znaków.

Standard języka C precyzuje jedynie, że kody cyfr muszą stanowić zwarty odcinek zakresu wartości, tj.

```
'0' + 1 = '1'
'1' + 1 = '2'
...
'8' + 1 = '9'
```

O kodach liter powiedziane jest tylko tyle, że wartości ich kodów muszą zachowywać porządek alfabetyczny, tj.

```
'A' < 'B' < ... < 'Z'
'a' < 'b' < ... < 'z'
```

2.4 Typ int

Do przechowywania wartości całkowitych służy typ `int`. Najczęściej zajmują one w pamięci 4 bajty (32 bity).

Zbiorem wartości typu `int` jest zakres od $-2,147,483,648$ do $+2,147,483,647$.

Jeśli liczymy na wartościach nieujemnych, to powinniśmy użyć typu `unsigned int`. Jego wartości również zajmują 4 bajty ale ich zakres to liczby od 0 do $4,294,967,295$.

Jeśli potrzebujemy do obliczeń szerszego zakresu wartości, to powinniśmy użyć typu `long int`. Wówczas jego wartości przechowywane są na 8 bajtach (64 bity) i mają zakres od $-9,223,372,036,854,775,808$ do $+9,223,372,036,854,775,807$.

Podobnie jest w przypadku typu `int`, jeśli liczymy na liczbach nieujemnych, to mamy dostępny typ `unsigned long int` z zakresem wartości od 0 do $18,446,744,073,709,551,615$.

Pamiętajmy, że jeśli podczas obliczeń na liczbach typu całkowitego, wynik wykracza poza zakres typu, to „przekręci się" tj. powyżej największej wartości dodatniej przejdzie na wartości ujemne i, analogicznie, poniżej wartości ujemnych przejdzie na wartości dodatnie.

Dla przykładu, chociaż liczba 2000000000 należy do zbioru wartości typu `int`, to suma 2000000000 + 2000000000 wykracza poza największą wartość w tym typie. Wynikiem obliczeń będzie wartość ujemna −294967296.

W Tabeli 2.2 zebrano dwuargumentowe operacje na liczbach całkowitych jakie dostępne są w języku C.

Poza operacjami dwuargumentowymi są jeszcze operacje jednoargumentowe:

`+a` wartość `a`

`-a` wartość przeciwna do `a`

`~a` bitowa negacja wartości `a`

Tabela 2.2: Operacje dwuargumentowe na liczbach całkowitych

Operator	Operacja	Przykład
`+`	suma	
`-`	różnica	
`*`	iloczyn	
`/`	iloraz całkowity	`7 / 2 = 3`
`%`	reszta z dzielenia	`7 % 3 = 1`
`|`	bitowa alternatywa	`2 | 1 = 3 (10 | 01 = 11)`
`&`	bitowa koniunkcja	`5 & 3 = 1 (101 & 011 = 001)`
`^`	bitowa alternatywa wykluczająca	`12 ^ 9 = 5 (1100 ^ 1001 = 0101)`

`a++` post-inkrementacja zmiennej `a`

`a--` post-dekrementacja zmiennej `a`

`++a` pre-inkrementacja zmiennej `a`

`--a` pre-dekrementacja zmiennej `a`

2.5 Typ float

Do przechowywania liczb rzeczywistych służy typ `float`. Jest on typem liczb zmiennopozycyjnych pojedynczej precyzji. Jego wartości przechowywane są na 4 bajtach (32 bitach).

Reprezentowane są one zgodnie ze standardem IEEE 754, w którym jeden bit przechowuje znak liczby, 8 bitów przechowują wykładnik (cechę) a 23 bity przechowują ułamek (mantysę).

Jeśli $s \in \{0, 1\}$ jest znakiem liczby, $w \in [-126, 127]$ jest wykładnikiem a $m \in [0, 1)$ mantysą, to zapisana za ich pomocą liczba ma wartość:

$$(-1)^s \cdot (1 + m) \cdot 2^w.$$

Wartość ta należy do zakresu

$$[-3.4 \cdot 10^{38}, -1.18 \cdot 10^{-38}] \cup \{0\} \cup [1.18 \cdot 10^{-38}, 3.4 \cdot 10^{38}].$$

Wynikają z tego następujące własności zakresu wartości typu `float`:

- Jest najmniejsza wartość $-3.4 \cdot 10^{38}$.
- Jest największa wartość ujemna $-1.18 \cdot 10^{-38}$.
- Jest najmniejsza wartość dodatnia $1.18 \cdot 10^{-38}$.
- Jest największa wartość $3.4 \cdot 10^{38}$.

Zakres wartości typu `float` ma jeszcze jedną bardzo ważną własność. Otóż w typie tym jest tylko skończenie wiele wartości, które nie są równomiernie rozmieszczone w całym zakresie. Im dalej jesteśmy od zera, tym odstępy między kolejnymi wartościami typu `float` są coraz większe (podwojenie wartości podwaja odległość między liczbami).

Powoduje to, że nie każda liczba da się reprezentować w tym typie i im co do modułu jest ona większa, tym większy może być związany z jej pamiętaniem błąd reprezentacji.

Liczby z typu `float` zapisuje się w notacji inżynierskiej z literą E reprezentującą potęgowanie dziesiątki i sufiksem F oznaczającym pojedynczą precyzję:

```
 -1.2E3F = -1200.0
0.031415E2F =     3.1415
   1E-4F =     0.0001
```

Typ `float` gwarantuje dokładne reprezentowanie siedmiu najbardziej znaczących cyfr. Za siódmą cyfrą mogą pojawić się już błędne cyfry wynikające z błędów reprezentacji.

Prosty program:


```c
// float.c

#include <stdio.h>

int main(void)
{
  float x;
  scanf("%f", &x);    //       wpisz 1234.5678
  printf("%f\n", x); // pojawi się 1234.567749
  return 0;
}
```

i wynik jego działania:

```
$ ./float
1234.5678
1234.567749
```

2.6 Typ double

Do dokładniejszego przechowywania liczb rzeczywistych służy typ `double`. Jest on typem liczb zmiennopozycyjnych podwójnej precyzji. Jego wartości przechowywane są na 8 bajtach (64 bitach).

Reprezentowane są one zgodnie ze standardem IEEE 754, w którym jeden bit przechowuje znak liczby, 11 bitów przechowują wykładnik (cechę) a 52 bity przechowują ułamek (mantysę).

Jeśli $s \in \{0, 1\}$ jest znakiem liczby, $w \in [-1022, 1023]$ jest wykładnikiem a $m \in [0, 1)$ mantysą, to zapisana za ich pomocą liczba ma wartość:

$$(-1)^s \cdot (1 + m) \cdot 2^w.$$

Wartość ta należy do zakresu

$$[-1.8 \cdot 10^{308}, -2.2 \cdot 10^{-308}] \cup \{0\} \cup [2.2 \cdot 10^{-308}, 1.8 \cdot 10^{308}].$$

Wynikają z tego następujące własności zakresu wartości typu `double`:

- Jest najmniejsza wartość $-1.8 \cdot 10^{308}$.
- Jest największa wartość ujemna $-2.2 \cdot 10^{-308}$.
- Jest najmniejsza wartość dodatnia $2.2 \cdot 10^{-308}$.
- Jest największa wartość $1.8 \cdot 10^{308}$.

Zakres wartości typu `double` ma jeszcze jedną bardzo ważną własność. Otóż w typie tym jest tylko skończenie wiele wartości, które nie są równomiernie rozmieszczone w całym zakresie. Im dalej jesteśmy od zera, tym odstępy między kolejnymi wartościami typu `double` są coraz większe (podwojenie wartości podwaja odległość między liczbami).

Powoduje to, że nie każda liczba da się reprezentować w tym typie i im co do modułu jest ona większa, tym większy może być związany z jej pamiętaniem błąd reprezentacji.

Liczby z typu `double` zapisuje się w notacji inżynierskiej z literą E reprezentującą potęgowanie dziesiątki:

```
    -1.2E3 = -1200.0
0.031415E2 =     3.1415
      1E-4 =     0.0001
```


Typ `double` gwarantuje dokładne reprezentowanie szesnastu najbardziej znaczących cyfr. Za szesnastą cyfrą mogą pojawić się już błędne cyfry wynikające z błędów reprezentacji.

2.7 Deklaracje zmiennych

Deklaracja zmiennej ma postać:

```
Typ nazwa_zmiennej;
```

Deklaracja może zawierać również inicjowanie wartości zmiennej:

```
Typ nazwa_zmiennej = wartość_początkowa;
```

Zatem zamiast pisać:

```
unsigned int wiek;
wiek = 19;
```

można napisać:

```
unsigned int wiek = 19;
```

Jeśli jest więcej zmiennych tego samego typu, mogą one być zadeklarowane w jednej deklaracji:

```
Typ pierwsza_zmienna, druga_zmienna = wartość_początkowa, trzecia_zmienna;
```

Każda zmienna użyta w programie musi być wcześniej zadeklarowana. W przeciwnym przypadku kompilacja kończy się błędem `use of undeclared identifier`.

2.7.1 Deklaracje zmiennych po pierwszej instrukcji

C90

Do chwili opracowania w roku 1990 standardu C90, deklaracje zmiennych mogły pojawiać się tylko na początku bloku funkcji (zmienne lokalne funkcji) lub na zewnątrz funkcji (zmienne globalne dostępne w dalszej części programu).

Standard C90 dopuszcza deklarowanie zmiennej w dowolnym miejscu (za pierwszą instrukcją funkcji).

Dzięki temu można zawęzić zakres dostępności takiej zmiennej, tj. ograniczyć fragment programu, w którym można korzystać z tej zmiennej (to bardzo ważne bo ułatwia analizę i zrozumienie działania programu).

Prosty przykład:


```
int i, n;
scanf("%d", &n);
int suma = 0;
for(i = 1; i <= n; i++)
{
  int kwadrat = i*i;
  suma = suma + kwadrat;
  if(kwadrat > 100)
    printf("liczba %d ma kwadrat większy od 100\n", i);
}
printf("suma kwadratów liczb od 1 do %d = %d", n, suma);
```

Aby nie liczyć dwukrotnie kwadratu liczby `i` został on policzony raz i zapamiętany w zmiennej `kwadrat`. Zmienna ta jest zadeklarowana w bloku instrukcji `for` i tylko w nim jest dostępna (to jej zakres deklaracji).

2.7.2 Deklaracje zmiennych w pętli

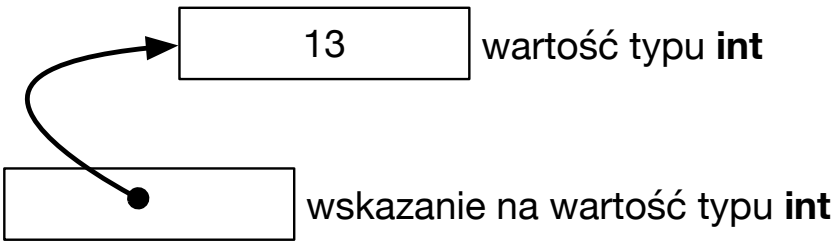
C99

Opracowany w roku 1999 standard C99 pozwala na deklarowanie zmiennych sterujących pętlą wewnątrz tych pętli.

To ważne ze względu na analizę działania programu, bo często wartości zmiennych sterujących pętlą nie mają sensu poza tą pętlą.

Jeszcze raz przykład z liczeniem sumy kwadratów:

```
int n;
scanf("%d", &n);
int suma = 0;
scanf("%d", &n);
for(int i = 1; i <= n; i++)
{
  int kwadrat = i*i;
  suma = suma + kwadrat;
  if(kwadrat > 100)
```



Rysunek 2.1: Wartość liczbowa i wskazanie na nią.

```
    printf("liczba %d ma kwadrat większy od 100\n", i);
}
printf("suma kwadratów liczb od 1 do %d = %d", n, suma);
```

2.8 Typ wskaźnikowy

Szczególnym typem jest typ wskaźnikowy. Nie służy on do przechowywania wartości ale do przechowywania wskazania prowadzącego do wartości.

Na rysunku 2.1 przedstawiono różnicę między wartością a wskazanie do wartości.

Wskazanie prowadzące do wskazywanej wartości realizuje się za pomocą adresu. Jeśli wartość przechowywana jest pod adresem `addr`, to wskaźnik prowadzący do niej ma wartość `addr`.

Wartość wskaźnika oczywiście jest określonego typu. Typ ten zależy od typu wskazywanej wartości. Jeśli `T` jest typem wskazywanej wartości, to wskaźnik do niej jest typu `T*`.

Poniższa deklaracja definiuje zmienną `ptr` służącą do przechowywania wskazania na wartość całkowitą:

```
int* ptr;
```

Aby dostać się do wskazywanej przez zmienną `ptr` wartość, należy użyć jednoargumentowy operator `*` (wyłuskania, dereferencji):

```
*ptr
```

Wartość `*ptr` jest typu `int` dlatego, że `ptr` jest typu `int*`.

Jeśli `*ptr` występuje w kontekście wyrażenia, to dostarcza wskazywaną wartość:

```
x = *ptr; // pod zmienną x podstawiana jest wskazywana przez ptr wartość
```

Jeśli `*ptr` jest po lewej stronie znaku przypisania, to zachowuje się jak zmienna wskazywanego typu, pod którą można podstawić wartość:

```
*ptr = x; // we wskazywane przez ptr miejsce wstaw wartość zmiennej
```

Aby zarezerwować w pamięci komputera miejsce na przechowywanie wartości należy wywołać funkcję `malloc()` (ang. *Memory ALLOCation*).

Argumentem tej funkcji jest liczba rezerwowanych bajtów. Rozmiar pamięci, wyrażony w bajtach, potrzebny do przechowania wartości typu `T` można wyliczyć operatorem `sizeof`.

Przykład deklaracji zmiennej `ptr` wskazującej na wartości całkowite i zainicjowanie jej adresem zarezerwowanego miejsca pamięci, w którym zmieszczą się takie wartości:

```
int* ptr = malloc(sizeof(int));
```

Jeśli nie ma w pamięci komputera dostatecznie dużo miejsca na zarezerwowanie zadanej liczby bajtów, to funkcja `malloc()` zwraca wartość `NULL`.

Wartość `NULL` jest stałą oznaczającą puste wskazanie, które nie prowadzi do żadnego miejsca (do żadnej wartości).

Jeśli wskazywana wartość nie jest już potrzebna i można zwolnić zajmowaną przez nią pamięć, należy wywołać funkcję `free()`.

Argumentem funkcji `free()` jest wskazanie na zwalniane miejsce pamięci (niepotrzebną już wartość).

Jeśli zmienna `ptr` wskazuje na niepotrzebną już wartość, to należy wywołać `free(ptr)`.

Wartość zmiennej `ptr`, po wywołaniu `free(ptr)`, nie ulega zmianie, jednak wskazuje na miejsce, którego nie należy już używać bo zostało zwolnione.

Przykład błędnego programu, w którym wskazanie na liczbę całkowitą zostało skopiowane do innej zmiennej:


```c
// ptr.c
//
// wynik analizy statycznej:
//
// $ clang --analyze ptr.c
// ptr.c:23:18: warning: Use of memory after it is freed
//   printf("%d\n", *y); // ta instrukcja nie ma sensu
//                  ^~
// 1 warning generated.

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int* x;
  int* y;
  x = malloc(sizeof(int));
  *x = 13;
  y = x;
  printf("%d\n", *y); // wydrukuje 13
  free(x);
  printf("%d\n", *y); // ta instrukcja nie ma sensu
  return 0;
}
```

Kompilacja nie wskazuje żadnego błędu:

```
$ clang -Wextra --pedantic -std=c11 -o ptr ptr.c
$
```

Chociaż uruchomienie programu przypadkiem[2] nie kończy się błędem:

```
$ ./ptr
13
13
$
```

to jednak analiza statyczna ostrzega przed odwołaniem się do zwolnionej już pamięci:

```
$ clang --analyze ptr.c
ptr.c:23:18: warning: Use of memory after it is freed
  printf("%d\n", *y); // ta instrukcja nie ma sensu
                 ^~
1 warning generated.
```

2.9 Stan obliczeń

Przez stan obliczeń będziemy rozumieć wartości wszystkich zmiennych w programie w danej chwili. Stan obliczeń zmienia się w trakcie działania programu.

2.9.1 Asercje

Często jest tak, że w danym miejscu programu stan obliczeń musi spełniać jakiś warunek. W przeciwnym przypadku działanie programu należy przerwać gdyż wykryto sytuację, w której nie działa on poprawnie.

Do wyrażenia warunku koniecznego w tym miejscu służy polecenie:

```
assert(Warunek);
```

Aby z niego skorzystać należy dołączyć dyrektywą **include** plik `assert.h`.

2.9.2 Obliczanie wartości wyrażeń

Wartość wyrażenia zależy od wartości zmiennych w danym stanie obliczeń. Oznacza to, że w innym stanie wartość tego samego wyrażenia może być inna.

2.9.3 Zmiana stanu

Wszelkie instrukcje, które zmieniają wartości zmiennych, zmieniają stan. Aby zrozumieć działanie programu dobrze jest prześledzić zmiany stanu obliczeń. W tym celu można narysować tabelkę, której kolumny odpowiadają zmiennym a wiersze przedstawiają kolejne stany obliczeń.

Oto prosty program i tabela zawierająca kolejne stany jego obliczeń:

[2] Pamiętajmy, że aplikacje mogą być wielowątkowe i gdy w jednym wątku zwolniono pamięć, to za chwilę drugi wątek może ją zarezerwować dla zapamiętania innej wartości, zanim pierwszy wątek odwoła się nieaktualnym już wskaźnikiem.

```c
int main(void)
{
  int x = 10;
  int y = 4;
  int z = 12;
  x = x + y;
  y = x - z;
  z = x + y;
  return 0;
}
```

x	y	z	komentarz
10	4	12	stan początkowy
14	4	12	po wykonaniu `x = x + y`
14	2	12	po wykonaniu `y = x - z`
14	2	16	po wykonaniu `z = x + y`

2.9.4 Efekty uboczne

Jeśli podczas obliczania wartości wyrażenia zmienia się stan obliczeń, to mówimy, że wyrażenie to ma efekt uboczny.

W języku C często mamy do czynienia z wyrażeniami, które mają efekty uboczne.

Dla przykładu podczas obliczania wartości wyrażenia `a++` efektem ubocznym jest zwiększenie wartości zmiennej `a` o jeden.

Efekty uboczne są bardzo niebezpieczne bo utrudniają analizę programu.

Języki programowania dla zastosowań krytycznych (zastosowania, w których od poprawności działania programu zależy zdrowie albo życie ludzi) nie dopuszczają efektów ubocznych lub przynajmniej bardzo ograniczają ich użycie.

Rozpatrzmy następujący program:

```c
// main.c

#include <stdio.h>

int main(void)
{
  int a = 10;
  printf("%d\n", ++a + --a);
  return 0;
}
```

Okazuje się, że nie da się przewidzieć jaka wartość zostanie wydrukowana.

Oto dwa uruchomienia programu skompilowanego dwoma różnymi kompilatorami:

```
$ gcc -o main main.c
$ ./main
20
$ clang -o main main.c
main.c:6:18: warning: multiple unsequenced modifications to 'a' [-Wunsequenced]
  printf("%d\n", ++a + --a);
```

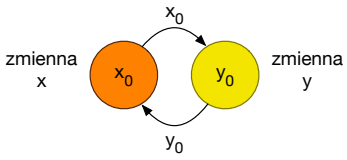
```
                  ^      ~~
1 warning generated.
$ ./main
21
```

Przynajmniej kompilator Clang ostrzegł nas o niejednoznaczności wyrażenia.

2.10 Ciekawostka

Jedną z podstawowych operacji jaką wykonuje się na danych jest ich uporządkowanie. Do jego wykonania potrzebne jest operacja zamieniająca miejscami dwie dane.

Załóżmy, że dwie zmienne całkowite, `x` i `y`, muszą wymienić się wartościami:



Najczęściej pierwszy pomysł jaki przychodzi do głowy nie jest poprawny:

```
x = y; // x równe y0
y = x; // y równe y0
// obie zmiennej mają tę samą wartość!
```

Zadanie jest proste gdy dostępna jest pomocnicza zmienna:

```
int pomoc = x; // pomoc równe x0
x = y;         // x równe y0
y = pomoc;     // y równe x0
```

Co jednak zrobić gdy nie jest dostępna pomocnicza zmienna?

Należy na chwilę w jednej zmiennej przechować wartość zawierającą w sobie jednocześnie obie wartości ale tak, aby było możliwe ich odtworzenie:

```
x = x + y; // x równe x0 + y0
y = x - y; // y równe (x0 + y0) - y0 czyli x0
x = x - y; // x równe (x0 + y0) - x0 czyli y0
```

Powyższe rozwiązanie ma jednak tę wadę, że wynik dodawania może wykraczać poza zakres wartości dopuszczalnych dla zmiennej `x`.

Bardzo praktyczną operacją jest alternatywa wykluczająca (XOR). Niech a i b będą wartościami zero-jedynkowymi. Alternatywa wykluczająca $a \oplus b$ ma następujące wartości:

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Operacja alternatywy wykluczającej ma następujące własności:

$$\begin{aligned} p \oplus q &= q \oplus p, \\ (p \oplus q) \oplus r &= p \oplus (q \oplus r), \\ p \oplus p &= 0, \\ p \oplus 0 &= p, \end{aligned}$$

dla dowolnych wartości $p, q, r \in \{0, 1\}$.

Z powyższych własności wynika, że:

$$(p \oplus q) \oplus q = p,$$

ponieważ:

$$(p \oplus q) \oplus q = p \oplus (q \oplus q) = p \oplus 0 = p.$$

Analogicznie:

$$(p \oplus q) \oplus p = q,$$

ponieważ:

$$(p \oplus q) \oplus p = (q \oplus p) \oplus p = q \oplus (p \oplus p) = q \oplus 0 = q.$$

Zanim przedstawimy rozwiązanie wykorzystujące operację alternatywy wykluczającej powiemy parę słów o bramkach odwracalnych.

Wyobraźmy sobie następujące dwie operacje dwuargumentowe.

$$\begin{aligned} U(p, q) &= (p \oplus q, q) \\ D(p, q) &= (p, p \oplus q) \end{aligned}$$

Każdą z nich możemy wyobrazić sobie jako bramkę o dwóch wejściach i dwóch wyjściach. Gdy na wejściach bramki U pojawiają się wartości p i 1, to na jej wyjściach pojawiają się wartości $p \oplus q$ i q.

Powyższe operacje są idempotentne, tj. złożenie dowolnej z nich z nią samą jest identycznością:

$$\begin{aligned} U \circ U(p, q) &= U(p \oplus q, q) = ((p \oplus q) \oplus q, q) = (p, q) \\ D \circ D(p, q) &= D(p, p \oplus q) = (p, p \oplus (p \oplus q)) = (p, q) \end{aligned}$$

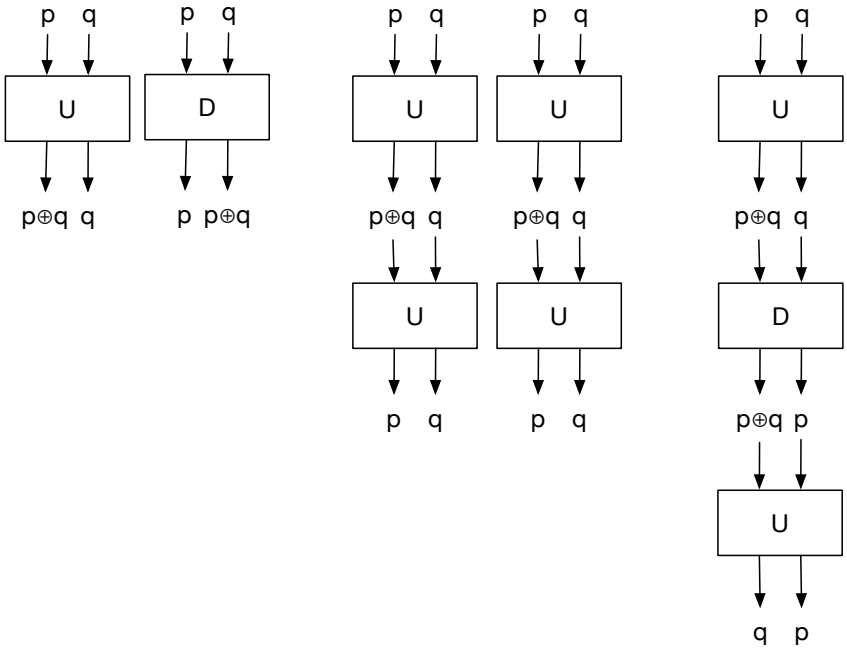
Ciekawą własność ma trzykrotne złożenie tych bramek:

$$\begin{aligned} U \circ D \circ U(p, q) &= U \circ D(p \oplus q, q) \\ &= U(p \oplus q, (p \oplus q) \oplus q) = U(p \oplus q, p) \\ &= ((p \oplus q) \oplus p, p) = (q, p) \end{aligned}$$

Jak widać po trzykrotnym złożeniu bramek składowe par zamieniły się miejscami.

Na rysunku 2.2 zebrano własności bramek U i D.

Poniższy program wymienia dwie zmienne wartościami przy użyciu operacji sumy wykluczającej:



Rysunek 2.2: Własności bramek U i D.


```c
// zamiana.c

#include <stdio.h>

int main(void)
{
  int x = 13;
  int y = 7;

  printf("x=%d, y=%d\n", x, y);
  x = x ^ y;
  y = x ^ y;
  x = x ^ y;
  printf("x=%d, y=%d\n", x, y);

  return 0;
}
```

Ćwiczenie 1 *Uzasadnij poprawność powyższego rozwiązania.*