OPERACJE WEJŚCIA-WYJŚCIA W JĘZYKU C++

Krótko o programowaniu obiektowym…

Klasą nazywamy strukturę danych, wraz z metodami (funkcjami) operującymi na tych danych. Klasę możemy zatem potraktować jako *nowy typ danych*, zawierający *pola* (dane) i *metody* (funkcje). Podobnie jak tworzymy zmienne różnych typów, tak możemy tworzyć *obiekty* należące do danej klasy. Deklaracja obiektu podobna jest do deklaracji zmiennej i ma postać: `Nazwa_klasy obiekt;` Stosując *operator* . (kropka) mamy dostęp do pól `obiekt.pole` i metod `obiekt.metoda()`.

Metody mogą być bezparametrowe (działać wyłącznie na polach obiektu) lub mogą mieć parametry dowolnego typu, w tym typu obiektowego. Specjalne metody, zwane *konstruktorami* służą do inicjowania wartości pól w obiektach. Należy wspomnieć też o *destruktorach*, czyli metodach usuwających niepotrzebne obiekty.

Niektóre pola mogą być zastrzeżone jako *prywatne* (nazywamy to *hermetyzacją* danych lub *enkapsulacją*) i wtedy nadanie im wartości jest możliwe jedynie przy pomocy *konstruktora* lub *metod*. Odwrotnie — możemy pewne pola ustanowić jako *publiczne*, a metody jako *prywatne*.

Kilka klas może działać w sposób ze sobą niepowiązany lub mogą tworzyć pewną *hierarchię klas*. Mówimy wtedy o *dziedziczeniu*. *Klasą bazową* nazywamy klasę z której dziedziczymy, natomiast klasę, która po niej dziedziczy nazywamy *klasą pochodną*. Klasa pochodna korzysta z pól i metod klasy bazowej i może (powinna) ją rozszerzać poprzez dodanie nowych pól lub metod. Klasa pochodna może modyfikować (*przysłaniać*) metody klasy bazowej. Klasa bazowa może zawierać tzw. *metody wirtualne*, które muszą być zdefiniowane w klasach pochodnych. Jedna klasa może dziedziczyć jednocześnie z kilku klas bazowych (tzw. *wielodziedziczenie*).

Kolejnym pojęciem są *funkcje zaprzyjaźnione*, które nie są metodami, ale mają dostęp do prywatnych pól klasy. W klasie, na rzecz obiektów klasy, możemy zdefiniować *operatory*, wykorzystując dostępne w języku C++ standardowe operatory (+ – * itd.) i nadając im nowe znaczenie, niekoniecznie zgodne z ich dotychczasowym przeznaczeniem. Mówimy wtedy o *przeciążeniu operatorów*. W pewnych sytuacjach, np. gdy operator nie może być zdefiniowany jako metoda, definiujemy go jako funkcję zaprzyjaźnioną.

Biblioteka IOstream (strumień wejścia-wyjścia)

Włączając plik nagłówkowy `<iostream>` otrzymujemy deklarację obiektów `cin` i `cout` umożliwiających komunikację ze standardowym wejściem-wyjściem (konsolą, czyli w typowych sytuacjach klawiaturą i monitorem).

Obiekt `cin`, pochodzący z klasy `istream` (*input stream*), odpowiada standardowemu wejściu `stdin`. Przeciążony operator >> (standardowe przznaczenie operatora to przesunięcie bitowe w prawo — dzielenie całkowite przez 2) pobiera ze strumienia `cin` sformatowane dane i umieszcza we wskazanych zmiennych:

```
int a;          // liczba całkowita
double b;       // liczba zmiennoprzecinkowa podwójnej precyzji
char c;         // pojedynczy znak
cin >> a >> b;
cin >> c;
```

Jeśli użytkownik wprowadzi z klawiatury ciąg znaków: *–125 45.78, komputer* i naciśnie klawisz Enter, to przytoczony wyżej fragment programu:

- zmiennej `a` przypisze liczbę całkowitą *–125*

- pominie odstęp (tzw. biały znak) i zmiennej `b` przypisze wartość *45.78*

- zmiennej `c` przypisze watrość *44* — kod znaku , (przecinek),

W strumieniu pozostanie *odstęp*, napis *komputer* i znak końca linii (do ewentualnego odczytania przez kolejne operacje wejścia).

Stosują metodę `getline` możemy odczytać ze strumienia określoną ilość znaków lub czytać dane aż do wystąpienia podanego znaku (znak ten jest pobierany ze strumienia i odrzucany). Przypuśćmy, że użytkownik wpisze na konsoli zdanie: *To jest przykładowy tekst.* i naciśnie *Enter*. Po wykonaniu fragmentu kodu

```
char tekst[100];      // tablica znaków
cin.getline(text, 21);       // wczytujemy ze strumienia wejściowego
                             ↪20 znaków
```

w tablicy `tekst` zostanie zapamiętanych 20 znaków i znak `'\0'` kończący łańcuch (*c-string*) lub mniejsza liczba znaków, jeżeli w strumieniu wcześniej pojawi się znak końca linii `'\n'`.

Natomiast gdyby w opisanej sytuacji wywołano metodę `getline` z trzema parametrami, to trzeci parametr, pojedynczy znak, ograniczałby zakres wczytywanych znaków. Oczywiście o przerwaniu czytania znaków zadecyduje wyczerpanie limitu wczytywanych znaków lub znak ograniczający, w zależności od tego, co wystąpi wcześniej:

```
cin.getline(text, 21, ' ');  // wczytujemy ze strumienia wejściowego
                             ↪20 znaków.
```

Dla uzupełnienia, należy dodać że użycie operatora >> (czyli instrukcji wejścia w postaci: `cin >> tekst;`) pozwoliłoby na przeczytanie fragmentu zdania, do pierwszego białego znaku (odstępu, tabulatora lub końca linii, W omawianym przypadku odczytane byłoby wyłącznie słowo *To*.

Obiekt `cout` pochodzi z klasy `ostream` (*output stream*) i odpowiada standardowemu wyjściu `stdout`. Przeciążony operator << (standardowo ten operator przesuwa bity w lewo, co odpowiada mnożeniu liczb całkowitych przez 2) wstawia do strumienia wyjściowego sformatowane dane. Początkowe ustawienia właściwości obiektu cout są takie, że format wyników (liczb) jest ustawiany w sposób dostosowany do wartości:

Instrukcja	Wynik	Uwagi
`cout << 1.0/3;`	0.333333	6 cyfr po przecinku
`cout << 1.0/2;`	0.5	ustawienia nie zmieniły się, ale zera nie znaczące zostały pominięte
`cout << 1.0/0.2;`	5	jw. — zerowa część ułamkowa została pominięta (rezultat nadal jest liczbą zmiennoprzecinkową)

Uwaga. Klasy `istream` i `ostream`, z których pochodzą obiekty `cin` i `cout`, dziedziczą z klasy `ios`, a ta klasa wywodzi się z klasy `ios_base`. W klasie `ios_base` mamy m.in. publiczne pola odpowiedzialne za format liczb, szerokość pola wydruku i prezyzję o raz metody do ustawiania tych wartości. Poprzez mechanizm dziedziczenia, pola i metody są dostępne w obiektach `cin` i `cout`.

Właśności obiektu możemy zmieniać, wywołując odpowiednie metody. Oto kilka przykładów:

`cout.setf(ios::fixed,ios::floatfield);` — liczby będą wyświetlane ze stałą liczbą miejsc po przecinku, jeśli nie dokonano innych zmian, domyślnie jest to 6 cyfr.

`cout.precision(4);` — ustawienie precyzji wyświetlania liczb na cztery miejsca po przecinku.

`cout.unsetf(ios::fixed);` — wyłączenie flagi[1] odpowiedzialnej za wyświetlanie stałej liczby miejsc po przecinku.

Przedstawione przykłady nie wyczerpują wszystkich możliwości, jakie oferuje nam biblioteka `iostream`. Jest to jedynie mała wskazówka do samodzielnego studiowania dokumentacji i testowania nowych, nieznanych funkcji.

Wszystkie pola i metody z klas istream i ostream dziedziczy klasa iostream. Z tej klasy wywodzą się dwie kolejne klasy: fstream — do działań na plikach, stringstream — klasa umożliwiająca wykorzystanie łańcucha znaków, jako strumienia wejścia-wyjścia.

[1] Flaga lub inaczej znacznik to określony bit w wartości pewnej zmiennej. Stan tego bitu 0 lub 1 brany jest pod uwagę przy wykonywaniu określonych czynności przez funkcję, której działanie jest uzależnione od stanu flagi.

Operacje na plikach

Operacje na plikach przebiegają w sposób podobny jak odczytywanie danych ze standardowego wejścia i zapisywanie do standardowego wyjścia. Włączając plik nagłówkowy `<fstream>` dostajemy dostęp do trzech klas:

`ifstream` — klasa dziedzicząca z klasy `istream`, obiekty tej klasy pozwolą nam na odczytywanie danych z plików, przy wykorzystaniu operatora >>, w sposób podobny jak czytanie ze standardowego wejścia.

`ofstream` — klasa dziedzicząca z klasy `ostream`, obiekty tej klasy pozwolą nam na zapisywanie danych do plików, przy wykorzystaniu operatora <<, w sposób podobny jak zapisywanie danych do standardowego wyjścia.

`fstream` — klasa dziedzicząca z klasy `iostream`, umożliwia utworzenie obiektu potrzebnego do zapisywania i odczytywania danych z tego samego pliku (naprzemian, bez zamykania i ponownego otwierania pliku).

Szczegóły związane z operacjami na plikach opisane są w rozdziale 11.

Łańcuchy znaków jako strumienie danych

Operacje na łańcuchu znaków (obiekt typu `string`), traktowanym jako strumień danych, są podobne do operacji na plikach lub standardowym wejściu-wyjściu. Włączając plik nagłówkowy `<sstream>` dostajemy dostęp do trzech klas:

`istringstream` — klasa dziedzicząca z klasy `istream`, obiekty tej klasy pozwolą nam na odczytywanie danych z łańcucha znaków, przy wykorzystaniu operatora >>.

`ostringstream` — klasa dziedzicząca z klasy `ostream`, obiekty tej klasy pozwolą nam na zapisywanie danych do łańcucha znaków, przy wykorzystaniu operatora <<.

`stringstream` — klasa dziedzicząca z klasy `iostream`, umożliwia utworzenie obiektu potrzebnego do zapisywania i odczytywania danych z tego samego łańcucha znaków.

Teoretycznie wszystko jest poprawne, ale w praktyce (w konkretnych implementacjach) nie wszystkie metody działają dobrze.

Przeanalizujmy następujący przykład:

```
#include<iostream>            /* będziemy korzystać z obiektów cin i cout */
#include<sstream>             /* będziemy korzystać z klas operujących
                              ↪ na łańcuchach znaków */
#include<cstdio>              /* dołączono ze względu na użycie funkcji system */
using namespace std;          /* włączamy standardową przestrzeń nazw */
int main()
{
    system("chcp 1250");      /* włączamy stronę kodową 1250 na konsoli */
```

```
/* tworzymy obiekt oss klasy ostringstream w trybie dopisywania znaków na końcu */
    ostringstream oss(ios_base::app);
    for (int i = 0; i < 10; ++i) {   // dla liczb naturalnych od 0 do 9
        oss << i << " ";             // wstawiamy do łańcucha (obiektu) kolejną
                                     ↪ liczbę i znak odstępu
        cout << oss.str() << endl;   // pokazujemy aktualną wartość obiektu
                                     ↪ (łańcucha znaków)
    }
/* tworzymy obiekt iss klasy istringstram – łańcuch znaków jako strumień wejściowy */
    istringstream iss(ios_base::in);
    iss.str(oss.str());              // inicjujemy obiekt łańcuchem znaków –
                                     ↪ wartość strumienia oss
    short num;                  // uwaga na typ tej zmiennej!
    while(iss >> num)           // dopóki strumień nie jest pusty odczytujemy
                                ↪ dane ze strumienia iss
       cout << num << endl;     // i wyświetlamy na konsoli
                                ↪/* koniec pracy... */
    system("pause");            // zatrzymujemy pracę programu, aż do naciśnięcia
                                ↪ dowolnego klawisza
    return 0;                   // powracamy do systemu operacyjnego
}
```

Program działał bez zarzutu[2], jeżeli typem zmiennej `num` był jeden z typów: `char`, `unsigned char`, `short`, `unsigned short`, `float` lub `double`. Dla typu `int`, po odczytaniu pierwszej wartości ze strumienia program przerywał działanie (jakiś błąd w bibliotece?).

Próba ponownego zainicjowania strumienia (w miejscu komentarza `/*koniec pracy */`) nie dała oczekiwanego efektu — ze strumienia nie można już czytać danych.

```
iss.clear();                // wyzerowanie flag sygnalizujących błędy
iss.str("27 72");           // wstawienie łańcucha "27 72" do strumienia
iss >> num;                 // odczytanie wartości ze strumienia
cout << num << endl;        // wyświetlenie wartości zmiennej num na konsoli
```

Zamiast oczekiwanej wartości *27*, na konsoli pojawiła się liczba *9* (poprzednio przeczytana wartość). Świadczy to o niepowodzeniu operacji odczytywania ze strumienia `iss >> num;`.

Po skompilowaniu w środowisku Borland C++ Bulder 6.0 omówione wyżej błędy nie wystąpiły.

[2] Autor używał środowiska Code::Blocks 8.02 z GNU GCC Compiler lub środowiska DevC++.