Rozdział 2

Klasy

2.1 Wprowadzenie

Programowanie obiektowe polega na operowaniu w programie obiektami, którymi są dane wraz z tak zwanymi metodami, czyli funkcjami wykorzystywanymi wyłącznie dla ściśle określonych danych.

Zmienna obiektowa lub po prostu obiekt jest elementem pewnej klasy. Klasa jest to nowy typ danych zdefiniowany przez programistę. Klasę definiujemy w podobny sposób jak strukturę. Dla przykładu rozważmy następujący fragment programu:

```
#define MAX  256

class ZbiorZnaki
{
protected:
   int znaki[MAX];
}
```

Została zdefiniowana klasa o nazwie ZbiorZnaki, zawierająca pola będące elementami tablicy `znaki`. Przed deklaracją tablicy jest umieszczone słowo **protected**. Oznacza to, że elementy tej tablicy są dostępne (przy użyciu operatora wyboru składowej - kropki) wyłącznie w danej klasie i wszystkich klasach pochodnych, natomiast nie są dostępne na zewnątrz klasy. Dostęp do zadeklarowanych pól, w naszym przypadku elementów tablicy, z programu oraz wyprowadzanie ich zawartości może odbywać

się wyłącznie przy wykorzystaniu metod czyli funkcji zdefiniowanych w danej klasie.

Przypuśćmy teraz, że klasę ZbiorZnaki chcemy zastosować do realizacji operacji na zbiorze znaków (zakładamy, że znaków tych mamy 256 i stąd odpowiednia definicja stałej). Elementy tablicy `znaki` będą nam służyły do przechowywania informacji o tym, czy dany znak należy do zbioru znaków, czy też nie. Jeżeli dany znak należy do zbioru znaków, to w odpowiednim miejscu tablicy `znaki` umieścimy wartość 1, a w przeciwnym przypadku wartość 0. To odpowiednie miejsce bardzo łatwo można wyznaczyć, ponieważ jeżeli w zmiennej `x` typu char jest umieszczony jakiś znak, to zmienną tę można bezpośrednio wykorzystać do zaindeksowania tablicy `znaki`. Rozważmy na przykład następujący fragment programu:

```
char x;
x = 'w';
znaki[x] = 1;
```

Ponieważ kod znaku 'w' to 119 więc wartość 1 będzie wpisana do 119 elementu tablicy `znaki`. Jak widać, jeżeli wykorzystujemy zmienną typu char do określenia indeksu tablicy, to dany znak zostanie zastąpiony jego kodem.

Załóżmy dalej, że chcemy dysponować następującymi operacjami na zbiorze znaków: dodanie elementu do zbioru, obliczenie liczby elementów, wyprowadzenie wszystkich elementów zbioru oraz sprawdzenie czy dany znak należy do zbioru. Każdą z tych operacji zrealizujemy przy pomocy oddzielnej metody, tzn. funkcji, która może operować na ściśle określonych danych, a w naszym przypadku na tablicy `znaki` zadeklarowanej w części **protected**.

Prototypy projektowanych metod musimy umieścić w deklaracji klasy najlepiej w części **public**, co oznacza, że mogą być one stosowane w całym programie. A oto pełna definicja klasy ZbiorZnaki:

```
#define MAX  256

class ZbiorZnaki
{
protected:
   int znaki[MAX];
public:
```

```
   ZbiorZnaki();
   void Dodaj(unsigned char);
   int Nalezy(unsigned char);
   int Liczba();
   void Wszystkie();
};

ZbiorZnaki::ZbiorZnaki()
// konstruktor
{
   int i;
   for (i=0; i<MAX; i++) znaki[i] = 0;
}

void ZbiorZnaki::Dodaj(unsigned char znak)
// dodawanie elementu do zbioru znaków
{
   znaki[znak] = 1;
}

int ZbiorZnaki::Nalezy(unsigned char znak)
// sprawdzenie, czy znak należy do zbioru znaków
{
   return znaki[znak];
}

int ZbiorZnaki::Liczba()
// określenie liczby elementów zbioru znaków
{
   int i, licz = 0;
   // określenie liczby niezerowych elementów tablicy znaki
   for (i=0; i < MAX; i++)
      if (znaki[i]) licz ++;
   return licz;
}

void ZbiorZnaki::Wszystkie()
// wyprowadzenie wszystkich znaków ze zbioru
{
```

```
  int i;
cout << "Zbiór znaków" << endl;
for (i=0; i<MAX; i++)
if (znaki[i]) {
cout << (char) i;   // rzutowanie typu int na char
      }
}
```

Zwróćmy uwagę na sposób zapisywania nagłówka definicji metody. Otóż po podaniu typu metody należy podać nazwę klasy, a następnie po dwóch dwukropkach nazwę metody. Nie można też zapomnieć o podaniu prototypu metody w definicji klasy.

Jedna z metod ma taką samą nazwę jak nazwa klasy. Jest to specyficzna metoda zwana konstruktorem. Będzie ona automatycznie wywoływana w momencie tworzenia obiektu danej klasy.

Metoda zdefiniowana w klasie ma dostęp do pól obiektu tej klasy. Daną metodę wywołujemy w ten sposób, że najpierw podajemy nazwę obiektu, a następnie po kropce nazwę metody.

Obiekty deklarujemy analogicznie jak inne zmienne w języku C++. Podajemy najpierw nazwę klasy a następnie listę obiektów. Jak już wiemy, w momencie tworzenia obiektu jest wywoływany konstruktor.

Rozważmy teraz sposób wykorzystania klasy ZbiorZnaki. Po pierwsze można utworzyć obiekt klasy ZbiorZnaki w sposób następujący:

```
ZbiorZnaki pierwszy;
```

Został utworzony obiekt o nazwie `pierwszy` i jednocześnie wywołany konstruktor, którego zadaniem jest przypisanie wszystkim elementom tablicy `znaki` wartości 0. Jeżeli napiszemy:

```
pierwszy.Dodaj('a');
```

to do naszego zbioru reprezentowanego przez tablicę `znaki` dodajemy element 'a'. Instrukcja:

```
pierwszy.Dodaj('b');
```

dodaje do zbioru element 'b'. Natomiast powtórne wykonanie instrukcji:

```
pierwszy.Dodaj('a');
```

nie zmieni już zawartości zbioru.

Warto podkreślić, że gdyby w programie utworzyć jeszcze jeden obiekt klasy ZbiorZnaki, na przykład tak:

```
ZbiorZnaki drugi;
```

to wtedy możemy wykorzystywać niezależnie dwa zbiory: jeden reprezentowany przez obiekt `pierwszy` i drugi reprezentowany przez obiekt `drugi`. Każdy z tych obiektów posiada własną tablicę `znaki`, która przyjmuje odpowiednie wartości w zależności od przechowywanego zbioru. Ilustruje to rys. 2.1.

Rysunek 2.1: Ilustracja przykładowej zawartości zbiorów reprezentowanych przez obiekt `pierwszy` i `drugi`

Do tablicy `znaki` każdego obiektu dostęp mają wyłącznie metody klasy ZbiorZnaki z tym, że metody zdefiniowane w klasie ZbiorZnaki nie powielają się wraz z tworzeniem nowych obiektów. A zatem każdy obiekt ma swój własny zestaw pól zdefiniowanych w klasie, natomiast metody określone w klasie są wspólne dla wszystkich obiektów. Musi być więc dokładnie zaznaczone dla jakiego obiektu jest wywoływana dana metoda.

Na przykład instrukcja:

```
cout << pierwszy.Liczba();
```

podaje liczbę elementów w zbiorze reprezentowanym przez obiekt `pierwszy`, a instrukcja:

```
cout << drugi.Liczba();
```

podaje liczbę elementów w zbiorze reprezentowanym przez obiekt `drugi`. Dzieje się tak dlatego, że do metody Liczba() zdefiniowanej w klasie ZbiorZnaki automatycznie jest przesyłany wskaźnik obiektu, dla którego dana metoda ma działać. Wskaźnik ten nazywa się **this** i zawiera adres obiektu klasy ZbiorZnaki. A zatem metoda Liczba() jest automatycznie modyfikowana w następujący sposób:

```
int ZbiorZnaki::Liczba()
// określenie liczby elementów zbioru znaków
{
   int i, licz = 0;
   // określenie liczby niezerowych elementów tablicy znaki
   for (i=0; i < MAX; i++)
      if (this -> znaki[i]) licz ++;
   return licz;
}
```

Jak widzimy mamy teraz:

```
this -> znaki[i]
```

a więc jest wykorzystywana tablica `znaki` zawarta w obiekcie wskazywanym przez wskaźnik **this**.

W większości przypadków dodawanie przez programistę wskaźnika **this** jest zbędne i praktyka programowania jest taka, że wskaźnik ten dodaje się tylko wtedy, gdy jest to konieczne. Przykłady wykorzystania wskaźnika **this** zawiera dalsza część książki.

W przypadku, gdy chcemy zdefiniowaną przez nas klasę wykorzystać w innych programach, warto się zastanowić, jak to zrobić najwygodniej. Otóż powszechną praktyką programowania jest, że deklaracje klasy umieszcza się w pliku nagłówkowym (z rozszerzeniem .h), a metody zdefiniowane w klasie w pliku z rozszerzeniem .cpp. W naszym przypadku umówmy się, że deklaracja klasy ZbiorZnaki jest zawarta w pliku *kl_zbior.h*, a metody tej klasy są umieszczone w pliku *kl_zbior.cpp*. W pliku *kl_zbior.cpp* musi być umieszczona dyrektywa preprocesora:

```
#include "kl_zbior.h"
```

co pozwala na dołączenie deklaracji klasy. Jest to niezbędne, ponieważ nowo zdefiniowany typ danych, jakim jest klasa ZbiorZnaki, powinien być dostępny przed kompilacją metod tej klasy.

Wykorzystanie klasy ZbiorZnaki, do obliczenia ile jest różnych znaków we wczytywanym słowie ilustruje poniższy program. Warto zwrócić uwagę, że w programie tym musi być zawarta dyrektywa:

```
#include "kl_zbior.h"
```

a ponadto program ten powinien być kompilowany równocześnie z plikiem *kl_zbior.cpp*.

Przykład 2.1

```
#include <iostream.h>
#include <string.h>
#include "kl_zbior.h"    // dołączenie deklaracji klasy
                         // ZbiorZnaki

void main(void)
{
   ZbiorZnaki pierwszy;   // deklaracja obiektu
   char slowo[30];
   int i;
   cout << "Wprowadź słowo" << endl;
   cin >> slowo;
   for (i=0; i<strlen(slowo); i++) pierwszy.Dodaj(slowo[i]);
   cout << "Słowo zawiera " << pierwszy.Liczba()
   << " znaków (różnych)" << endl;
   pierwszy.Wszystkie(); // wyprowadzenie wszystkich
                         // różnych znaków
}
```

Rozważmy jeszcze sytuację, gdy nazwa zmiennej lokalnej w jakiejś metodzie jest taka sama jak nazwa pola klasy. W tym przypadku zmienna lokalna przesłania pole zdefiniowane w klasie. Aby uzyskać dostęp do tego

pola, należy wykorzystać operator **::** (dwa dwukropki) lub użyć wskaźnika **this**.

Na marginesie warto zaznaczyć, że takich sytuacji należy unikać i stosować nazwy zmiennych lokalnych, które nie kolidują z nazwami pól.

2.2 Składowe prywatne i publiczne

Jak pamiętamy, w klasie ZbiorZnaki były zdefiniowane dwie części wyróżnione słowami **protected** oraz **public**.

Jak wspominaliśmy w poprzednim podrozdziale, słowo **public** oznacza, że umieszczone za nim pola i metody są dostępne w całym programie. Natomiast pola i ewentualnie metody wyszczególnione po słowie **protected** są dostępne w sposób bezpośredni (przy użyciu operatora wyboru składowej - kropki) tylko w klasach pochodnych od danej klasy (klasy pochodne są omówione w podrozdziale 3). Istnieje jeszcze możliwość całkowitego zablokowania bezpośredniego dostępu do pól przy pomocy słowa **private**.

Ponieważ w przykładzie zamieszczonym w poprzednim podrozdziale nie są definiowane klasy pochodne, to nie ma znaczenia, czy użyjemy słowa **private**, czy też **protected**. W obu przypadkach dostęp do pól obiektu `pierwszy` klasy ZbiorZnaki (polami są elementy tablicy) uzyskujemy poprzez wykonanie odpowiednich metod. Oczywiście każda z metod wykorzystywała pola obiektu do innych celów, na przykład metoda Wszystkie() podawała wszystkie elementy zbioru, a metoda Liczba() określała liczbę elementów zbioru.

W tym miejscu warto samodzielnie sprawdzić, że na przykład konstrukcja:

```
unsigned char x;
for (i=0; i<MAX; i++)
   if (pierwszy.znaki[i] != 0) {
      x = i;
      cout << x;
   }
```

umieszczona w dowolnej funkcji, która nie jest metodą klasy ZbiorZnaki jest niepoprawna, ponieważ nie mamy dostępu do pola `znaki[i]`, a zatem poniższy fragment instrukcji:

```
   pierwszy.znaki[i]
```

będzie przyczyną błędu kompilacji.

W momencie zmiany deklaracji klasy na:

```
class ZbiorZnaki
{
public:
   int znaki[MAX];
public:
   ZbiorZnaki();
   void Dodaj(unsigned char);
   int Nalezy(unsigned char);
   int Liczba();
   void Wszystkie();
};
```

analizowany wyżej fragment programu będzie poprawny, ponieważ tym razem tablica `znaki` jest umieszczona w części **public**. Oczywiście, taka deklaracja kłóci się z podstawową zasadą programowania obiektowego, która mówi, że należy maksymalnie ograniczyć niekontrolowany dostęp do pól obiektu.

■

2.3 Konstruktory i destruktory

W niniejszym punkcie zwrócimy uwagę na specjalne metody, a mianowicie konstruktory i destruktory. Przypomnijmy sobie, że z konstruktorem mieliśmy już kontakt w punkcie 2.1, a więc na wstępie podsumujemy poznane wiadomości. Konstruktor jest funkcją składową o takiej samej nazwie jak nazwa klasy. Kompilator automatycznie wywołuje konstruktor w momencie tworzenia obiektu danej klasy. Deklaracja konstruktora nie może zawierać typu (nie wolno zamieścić nawet słowa **void**). Ponadto warto wiedzieć, że w przypadku gdy w klasie nie zamieścimy definicji konstruktora, to kompilator sam ją wygeneruje. Konstruktory można przeciążać tzn. definiować kilka funkcji o takiej samej nazwie, ale różniących się parametrami formalnymi.

Jak pamiętamy, w klasie ZbiorZnaki był zdefiniowany jeden konstruktor, którego zadaniem było przypisanie elementom tablicy `znaki` wartości 0.

Obecnie zmodyfikujemy definicję klasy ZbiorZnaki, dodając jeszcze jeden konstruktor. A oto deklaracja klasy:

```
class ZbiorZnaki
{
protected:
   int znaki[MAX];

public:
   ZbiorZnaki();                 // konstruktor
   ZbiorZnaki(char *tekst);      // konstruktor
   void Dodaj(unsigned char);
   int Nalezy(unsigned char);
   int Liczba();
   void Wszystkie();
};
```

Został dodany drugi konstruktor zawierający jeden parametr.

Przypomnijmy najpierw treść uprzednio zdefiniowanego konstruktora:

```
ZbiorZnaki::ZbiorZnaki()
{
   int i;
   for (i=0; i<MAX; i++) znaki[i] = 0;
}
```

A oto treść konstruktora dodanego do definicji klasy ZbiorZnaki.

```
ZbiorZnaki::ZbiorZnaki(char *tekst)
// konstruktor umożliwiający utworzenie zbioru znaków
// na podstawie tekstu wskazywanego przez parametr formalny
{
int i;
for (i=0; i<MAX; i++) znaki[i] = 0;

   // inicjalizacja tablicy znaki
```

```
   while (*tekst)
      znaki[*tekst++] = 1;
}
```

Konstruktor ten różni się od poprzedniego tym, że tekst wskazywany przez parametr formalny zostanie wykorzystany do utworzenia zbioru znaków.

Mając do dyspozycji dwa konstruktory, obiekty klasy ZbiorZnaki możemy tworzyć na dwa sposoby. Po pierwsze możemy wykorzystać konstruktor domyślny (ten, który nie zawiera parametrów):

```
ZbiorZnaki pierwszy;
```

Ponadto można zastosować drugi konstruktor:

```
ZbiorZnaki drugi("abrakadabra");
```

W tym przypadku zostanie utworzony obiekt o nazwie `drugi`, a różne litery zawarte w słowie *abrakadabra* będą umieszczone w zbiorze znaków.

Destruktor to specjalna funkcja bez parametrów formalnych wywoływana zawsze w momencie usuwania obiektu. Nazwą destruktora jest nazwa klasy poprzedzona znakiem falki „~”. Na przykład, dla klasy ZbiorZnaki nazwą destruktora jest ~ZbiorZnaki. W przypadku nie umieszczenia definicji destruktora zostanie ona automatycznie wygenerowana przez kompilator.

Dla prostych klas nie jest konieczne definiowanie własnych destruktorów. Wystarczy to zadanie pozostawić kompilatorowi. Dlatego przykładowa klasa ZbiorZnaki nie zawiera definicji destruktora. Podanie definicji destruktora staje się konieczne w przypadku, gdy musimy wykonać jakąś specyficzną operację w momencie usuwania obiektu. Przykład wykorzystania destruktora jest zawarty w dalszej części książki.

2.4 Struktury i unie jako klasy

W niniejszym punkcie krótko zwrócimy uwagę na fakt, że omawiane w pracy [9] struktury oraz unie są w języku C++ również klasami. Można wobec tego definiować w nich metody. Najważniejsza różnica polega na

tym, że jeżeli składowe struktur i unii nie są umieszczone w części **private**, to są traktowane jak publiczne. W związku z tym przyjęło się stosować struktury bez metod, nie wykorzystując w ogóle hermetyzacji, a wszędzie tam gdzie jest konieczna ścisła kontrola dostępu do danych używa się klas.