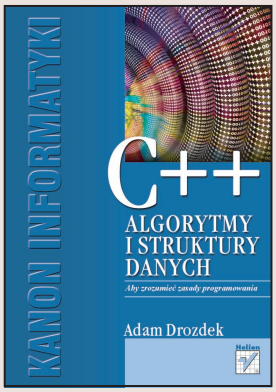
C++. Algorytmy i struktury danych



Autor: Adam Drozdek
Tłumaczenie: Piotr Rajca, Tomasz Żmijewski
ISBN: 83-7361-385-4
Tytuł oryginału: Data Structures and Algorithms in C++, Second Edition
Format: B5, stron: 616

Badanie struktur danych, elementarnych składników wykorzystywanych w informatyce, jest podstawą, w oparciu o którą możesz zdobywać cenne umiejętności. Znajomość struktur danych jest niezbędna studentom, którzy chcą programować czy też testować oprogramowanie.

W niniejszej książce zwrócono uwagę na trzy ważne aspekty struktur danych: po pierwsze, na związek struktur danych z algorytmami, między innymi na złożoność obliczeniową algorytmów. Po drugie, struktury te prezentowane są w sposób zgodny z zasadami projektowania obiektowego i obiektowym paradygmatem programowania. Po trzecie, ważną częścią książki są implementacje struktur danych w języku C++.

Książka prezentuje:

- Podstawy projektowania obiektowego w C++
- Analizę złożoności
- Listy powiązane
- Stosy i kolejki
- Rekurencję
- Drzewa binarne
- Sterty
- Drzewa wielokrotne
- Grafy
- Sortowanie i mieszanie
- Kompresja danych
- Zarządzanie pamięcią

Książka ta dostarcza studentom informatyki nie tylko niezbędnej wiedzy na temat algorytmów i struktur danych, ale prezentuje jednocześnie sposoby ich implementacji w języku C++, obecnie jednym z wiodących języków programowania. Dostarcza ona więc nie tylko wiedzy teoretycznej, ale również pozwala rozwinąć praktyczne umiejętności przydatnych w przyszłej pracy zawodowej.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

1

Programowanie obiektowe w C++

1.1. Abstrakcyjne typy danych

Przed napisaniem jakiegokolwiek programu trzeba wiedzieć dość dokładnie, jak powinno być realizowane zadanie wykonywane przez ten program. Wobec tego przed początkiem kodowania powinien powstać szkic programu opisujący wszystkie wymagania. Im większy i bardziej złożony jest projekt, tym bardziej szczegółowy powinien być ten szkic. Szczegóły implementacyjne powinny zostać odłożone do późniejszego etapu prac. W szczególności na później powinny zostać odłożone wszelkie szczegółowe ustalenia dotyczące struktur danych nieokreślone na samym początku.

Od samego początku trzeba określić każde zadanie za pomocą jego danych wejściowych i wyjściowych. Na wstępie bardziej powinno nas interesować to, co program ma robić, a nie jak ma to robić. Zachowanie się programu jest ważniejsze od tego, dlaczego program tak właśnie działa. Jeśli, na przykład, realizacja pewnego zadania wymaga jakiegoś narzędzia, narzędzie to opisuje się jako zestaw możliwych operacji, a nie podaje się jego wewnętrznej struktury. Operacje takie mogą modyfikować narzędzie, wyszukiwać dotyczące go szczegóły lub zapisywać w nim dodatkowe dane. Kiedy wszystkie te operacje zostaną dokładnie opisane, można rozpocząć kodowanie. Na tym etapie decyduje się o tym, które struktury danych będą najlepsze z punktu widzenia czasu wykonania i zajętości pamięci. Takie narzędzie opisane przez jego operacje nazywamy *abstrakcyjnym typem danych*. Abstrakcyjny typ danych nie jest częścią programu, gdyż program napisany w pewnym języku programowania wymaga zdefiniowania struktury danych, a nie tylko operacji na tej strukturze. Jednak języki obiektowe, takie jak C++, bezpośrednio wiążą abstrakcyjne typy danych z kodem w formie klas.

1.2. Enkapsulacja

Programowanie obiektowe obraca się wokół pojęcia obiektu. Obiekty tworzone są zgodnie z definicją klasy. *Klasa* to szablon, według którego tworzy się obiekty. Klasa to fragment kodu, który zawiera opis danych oraz funkcje przetwarzające te dane i, ewentualnie, także dane z innych klas. Funkcje definiowane w ramach klasy nazywamy *metodami* lub *funkcjami składowymi*, zaś zmienne używane w klasach to *dane składowe* lub po prostu *pola*. Takie łączenie danych i związanych z nimi operacji nazywamy *enkapsulacją* danych. *Obiekt* to instancja klasy — coś, co jest tworzone zgodnie z definicją tej klasy.

W przeciwieństwie do funkcji z języków nieobiektowych, w obiektach połączenie między danymi a metodami jest ścisłe i ma istotne znaczenie. W językach nieobiektowych deklaracje danych i definicje funkcji mogą przeplatać się w całym programie, a jedynie z dokumentacji wynika, jak dane z funkcjami są powiązane. W językach obiektowych związek ten jest ustalany już na wstępie; cała struktura programu opiera się na tym powiązaniu. Obiekt jest opisany przez powiązane dane i operacje, a jako że w tym samym programie używanych może być wiele obiektów, obiekty komunikują się, wymieniając komunikaty, które o ich strukturze wewnętrznej mówią tylko tyle, ile jest niezbędne. Podział programu na obiekty pozwala osiągnąć kilka celów.

Po pierwsze, silnie powiązane dane i operacje mogą znacznie lepiej opisywać modelowany fragment świata, co jest szczególnie istotne w przypadku inżynierii oprogramowania. Nie powinno zaskakiwać, że programowanie obiektowe ma swoje korzenie w symulacjach, czyli modelowaniu rzeczywistych obiektów. Pierwszym językiem obiektowym była Simula, która powstała w latach 60-tych w Norwegii.

Po drugie, użycie obiektów ułatwia szukanie błędów, gdyż operacje są wiązane od razu ze „swoimi" obiektami. Nawet jeśli pojawiają się efekty uboczne, łatwiej jest je wyśledzić.

Po trzecie, użycie obiektów pozwala ukryć pewne szczegóły operacji przed innymi obiektami, dzięki czemu operacje te nie ulegają tak łatwo zmianom w przypadku zmieniania innych obiektów. Jest to zasada *ukrywania informacji*. W językach nieobiektowych zasada ta w pewnym zakresie jest realizowana przez stosowanie zmiennych lokalnych, a (na przykład w Pascalu) także przez stosowanie lokalnych funkcji i procedur, które są dostępne jedynie w funkcjach je zawierających. Jednak jest to zawsze albo bardzo dokładne ukrywanie, albo ukrywania nie ma w ogóle. Czasami (na przykład w Pascalu) funkcja *f2* zdefiniowana w *f1* może być potrzebna poza *f1*, ale poza *f1* jest ona całkowicie niewidoczna. Czasami potrzebne jest sięgnięcie do pewnych danych lokalnych *f1* bez dokładnej znajomości struktury tych danych, ale to też jest niemożliwe. Dlatego też w programowaniu obiektowym nieco zmieniają się zasady dostępności danych.

Użytkowników interesuje działanie zegarka, ale jego budowa wewnętrzna nie ma już znaczenia. Wiadomo, że wewnątrz są tryby i sprężyny, ale zwykle niewiele wiadomo o sposobie ich działania, wobec czego dostęp do nich jest zbędny, gdyż mógłby prowadzić do uszkodzenia mechanizmu. Mechanizm jest przed użytkownikiem ukryty, nie ma do niego bezpośredniego dostępu, dzięki czemu zegarek nie jest narażony na zniszczenie bardziej, niż gdyby mechanizm był cały czas otwarty.

Obiekt to czarna skrzynka o dokładnie opisanym działaniu; obiektu używa się dlatego, że wiadomo, co on robi, ale nie ma się dostępu do jego wnętrza. Owa nieprzezroczystość obiektów jest wyjątkowo przydatna do serwisowania jednych obiektów niezależnie od innych. Jeśli dobrze zostaną określone kanały komunikacyjne między obiektami, zmiany w jednym obiekcie będą wpływały na inne obiekty tylko wtedy, gdy będą dotyczyły tych kanałów. Wiedząc, jakie informacje są wysyłane i odbierane przez obiekt, można łatwiej zastąpić jeden obiekt innym — nowy obiekt może realizować te same zadania inaczej, na przykład szybciej w danym środowisku. Obiekt ujawnia tylko tyle informacji o sobie, ile jest potrzebnych użytkownikowi, aby tego obiektu użyć. Obiekt ma część publiczną, która jest dostępna dla wszystkich, którzy prześlą do obiektu komunikat w formie jednej z funkcji przez obiekt udostępnianych. Do części publicznej należą przyciski pokazywane użytkownikowi; wciśnięcie każdego z tych przycisków powoduje wykonanie operacji obiektu. Użytkownik zna jedynie nazwy operacji i wie, co te operacje robią.

Ukrywanie informacji prowadzi do zacierania się rozgraniczenia między danymi a operacjami. W językach programowania takich jak Pascal podział na dane i funkcje jest prosty i stały. Inaczej definiuje się jedne i drugie, inne jest ich zastosowanie. W programowaniu obiektowym dane i metody są zestawione razem, zaś dla użytkownika obiektu podział jest znacznie mniej wyraźny.

W pewnym zakresie jest to też cecha języków funkcyjnych. LISP, jeden z najstarszych języków programowania, pozwala traktować dane i funkcje tak samo, gdyż struktura jednych i drugich jest identyczna.

Dokonaliśmy już podziału między konkretne obiekty a typy obiektów czyli klasy. Funkcje piszemy tak, aby używały różnych zmiennych, nie chcemy jednak być zmuszani do zapisywania tylu deklaracji obiektów, ilu obiektów potrzebujemy w całym programie. Pewne obiekty są tego samego typu i chcielibyśmy używać pojedynczego typu jako ogólnej definicji takiego typu. W przypadku pojedynczych zmiennych odróżniamy deklarację typu od deklaracji zmiennej. W przypadku obiektów mamy deklarację klasy i deklarację obiektu. Poniżej pokazano przykład klasy `C` i obiektów `object1`, `object2` i `object3`:

```
class C {
public:
    C(char *s = "", int i = 0, double d = 1) {
        strcpy(dataMember1,s);
        dataMember2 = i;
        dataMember3 = d;
    }
    void memberFunction1() {
        cout << dataMember1 << ' ' << dataMember2 << ' '
             << dataMember3 << endl;
    }
    void memberFunction2(int i, char *s = "nieznana") {
        dataMember2 = i;
        cout << i << " pobrana z " << s << endl;
    }
protected:
    char dataMember1[20];
    int dataMember2;
    double dataMember3;
};

C object1("obiekt1",100,2000), object2("obiekt2"), object3;
```

Przekazywanie komunikatów jest odpowiednikiem wywoływania funkcji w tradycyjnych językach programowania, jednak ważne jest, że w przypadku programowania obiektowego metody są powiązane z obiektami — stąd właśnie używany jest nowy termin. Przykładowo, wywołanie publicznej metody `memberFunction1()` obiektu `object1`:

```
object1.memberFunction1();
```

jest widziane jako wysłanie do obiektu `object1` komunikatu `memberFunction1()`. Po otrzymaniu takiego komunikatu obiekt wywołuje swoją metodę i wyświetla wszystkie odpowiednie w danej sytuacji informacje. Komunikaty mogą zawierać parametry:

```
object1.memberFunction2(123);
```

Powyższy zapis to wywołanie komunikatu `memberFunction2()` obiektu `object1` z parametrem `123`.

Wiersze zawierające takie komunikaty mogą być umieszczone albo w głównym programie, albo w funkcji, albo w metodzie innego obiektu. Wobec tego odbiorca komunikatu może zostać zidentyfikowany, ale nie dotyczy to wywołującego. Jeśli `object1` otrzyma komunikat `memberFunction1()`, nie wie, skąd ten komunikat pochodzi. Odpowiada na komunikat, pokazując dane zakodowane w metodzie `memberFunction1()`, to samo dotyczy metody `memberFunction2()`. Wobec tego nadawca komunikatu może do komunikatu dołączyć informacje umożliwiające jego zidentyfikowanie:

```
object1.memberFunction2(123, "obiekt1");
```

Istotną cechą C++ jest możliwość deklarowania ogólnych klas dzięki użyciu parametrów typu w deklaracji klasy. Jeśli, na przykład, trzeba zadeklarować klasę, która będzie coś przechowywała w tablicy, można zadeklarować ją następująco:

```
class intClass {
    int storage[50];
    ................
};
```

jednak w ten sposób ogranicza się przydatność takiej klasy jedynie do liczb całkowitych; jeśli potrzebna będzie analogiczna klasa z liczbami zmiennoprzecinkowymi, konieczna będzie nowa deklaracja:

```
class floatClass {
    float storage[50];
    ................
};
```

Jeśli `storage` ma zawierać struktury lub wskaźniki do znaków, konieczne byłoby zadeklarowanie kolejnych dwóch klas. Znacznie lepiej byłoby zadeklarować ogólną klasę i dopiero podczas definiowania obiektów decydować, jakiego typu dane będą w obiekcie przechowywane. Język C++ umożliwia takie deklarowanie klasy, na przykład:

```
template<class genType>
class genClass {
    genType storage[50];
    ..................
};
```

Dopiero potem decydujemy, jak inicjalizowany będzie typ `genType`:

```
genClass<int> intObject;
genClass<float> floatObject;
```

Taka klasa ogólna przybiera różne postacie w zależności od konkretnej deklaracji. Jedna ogólna deklaracja może udostępnić wszystkie te postacie.

Możemy pójść jeszcze dalej i decyzję o wielkości bufora, obecnie 50 komórek, odłożyć na później, do czasu tworzenia obiektu. Jednak na wszelki wypadek możemy też zostawić wartość domyślną:

```
template<class genType, int size = 50>
class genClass {
    genType storage[size];
    ..................
};
```

Definicje obiektów będą teraz wyglądały następująco:

```
genClass<int> intObject1; // użyty zostanie rozmiar domyślny
genClass<int,100> intObject2;
genClass<float,123> floatObject;
```

Pokazana metoda używania typów ogólnych nie ogranicza się jedynie do klas; można używać jej też w deklaracjach funkcji. Przykładowo, standardowa operacja zamiany miejscami dwóch wartości może być zdefiniowana następująco:

```
template<class genType>
void swap(getType& el1, genType& el2) {
    genType tmp = el1; el1 = el2; el2 = tmp;
}
```

Przykład ten pokazuje konieczność dostosowania operatorów wbudowanych do konkretnych sytuacji. Jeśli `genType` jest liczbą, znakiem lub strukturą, operator przypisania zadziała prawidłowo. Jeśli jednak `genType` jest tablicą, należy się spodziewać kłopotów ze strony funkcji `swap()`. Problem ten można rozwiązać, przeciążając operator przypisania tak, aby rozszerzyć jego możliwości w stosunku do określonych typów danych.

Po zadeklarowaniu funkcji ogólnej w czasie kompilacji programu można wygenerować konkretne funkcje. Jeśli, na przykład, kompilator „zobaczy” następujące dwa wywołania:

```
swap(n,m); // zamiana dwóch liczb całkowitych;
swap(x,y); // zamiana dwóch liczb zmiennoprzecinkowych;
```

wygeneruje dwie funkcje `swap()`, które będą używane w programie.

1.3. Dziedziczenie

Programowanie obiektowe pozwala tworzyć własne hierarchie klas tak, że obiekty nie muszą należeć tylko do pojedynczych klas. Zanim przejdziemy do omówienia dziedziczenia, przyjrzyjmy się następującym definicjom klas:

```
class BaseClass {
public:
   BaseClass() { }
    void f(char *s = "nieznany") {
        cout << "Funkcja f() w BaseClass wywołana z " << s << endl;
        h();
    }
```

```
protected:
    void g(char *s = "nieznany") {
        cout << "Funkcja g() w BaseClass wywołana z " << s << endl;
    }
private:
    void h() {
        cout << "Funkcja h() w BaseClass\n";
    }
};
class DerivedLevel1 : public virtual BaseClass {
public:
    void f(char *s = "nieznany") {
        cout << "Funkcja f() w DerivedLevel1 wywołana z " << s << endl;
        g("DerivedLevel1");
        h("DerivedLevel1");
    }
    void h(char *s = "nieznana") {
        cout << "Funkcja h() w DerivedLevel1 wywołana z " << s << endl;
    }
};
class Derived2Level1 : public virtual BaseClass {
public:
    void f(char *s = "nieznany") {
        cout << "Funkcja f() w Derived2Level1 wywołana z " << s << endl;
        g("Derived2Level1");
//      h(); // błąd: BaseClass::h() nie jest dostępna
    }
};
class DerivedLevel2 : public DerivedLevel1, public Derived2Level1 {
public:
    void f(char *s = "nieznany") {
        cout << "Funkcja f() w DerivedLevel2 wywołana z " << s << endl;
        g("DerivedLevel2");
        DerivedLevel1::h("DerivedLevel2");
        BaseClass::f("DerivedLevel2");
    }
};
```

A oto przykładowy program:

```
int main() {
    BaseClass bc;
    DerivedLevel1 d1l1;
    Derived2Level1 d2l1;
    DerivedLevel2 dl2;
    bc.f("main(1)");
//  bc.g(); // błąd: BaseClass::g() jest niedostępna
//  bc.h(); // błąd: BaseClass::h() jest niedostępna
    d1l1.f("main(2)");
//  d1l1.g() // błąd: BaseClass::g() jest niedostępna
    d1l1.h("main(3)");
    d2l1.f("main(4)");
```

```
//  d2l1.g(); // błąd: BaseClass::g() jest niedostępna
//  d2l1.h(); // błąd: BaseClass::h() jest niedostępna
    d1l2.f("main(5)");
//  d1l2.g(); // błąd: BaseClass::g() jest niedostępna
    d1l2.h();
      return 0;
}
```

Program powyższy da następujące wyniki:

```
Funkcja f() w BaseClass wywołana z main(1)
Funkcja h() w BaseClass
Funkcja f() w DerivedLevel1 wywołana z main(2)
Funkcja g() w BaseClass wywołana z DerivedLevel1
Funkcja h() w DerivedLevel1 wywołana z DerivedLevel1
Funkcja h() w DerivedLevel1 wywołana z main(3)
Funkcja f() w Derived2Level1 wywołana z main(4)
Funkcja g() w BaseClass wywołana z Derived2Level1
Funkcja f() w DerivedLevel2 wywołana z main(5)
Funkcja g() w BaseClass wywołana z DerivedLevel2
Funkcja h() w DerivedLevel1 wywołana z DerivedLevel2
Funkcja f() w BaseClass wywołana z DerivedLevel2
Funkcja h() w BaseClass
Funkcja h() w DerivedLevel1 wywołana z nieznany
```

Klasa `BaseClass` jest nazywana *klasą bazową*, zaś pozostałe klasy to *klasy pochodne*, gdyż pochodzą od klasy bazowej w tym sensie, że mogą korzystać z pól i metod `BaseClass` oznaczonych jako chronione (słowo kluczowe `protected`) lub publiczne (`public`). Klasy te dziedziczą wszystkie takie pola po klasie bazowej, dzięki czemu zbędne jest powtarzanie w nich tych samych definicji. Jednak klasa pochodna może przykryć definicję z klasy bazowej, podając własną definicję tej samej składowej. W ten sposób zarówno klasa bazowa, jak i klasy pochodne mogą zapanować nad swoimi składowymi.

Klasa bazowa może decydować, które metody i które pola mogą zostać ujawnione klasom pochodnym, wobec czego zasada ukrywania informacji jest zachowana nie tylko względem użytkowników klasy bazowej, ale także klas pochodnych. Co więcej, klasa pochodna może zdecydować, które części metod i pól publicznych i chronionych zostaną zachowane, a które będą modyfikowane. Przykładowo, w obu klasach pochodnych pierwszego poziomu, `DerivedLevel1` i `Derived2Level1`, zdefiniowane są lokalne wersje metody `f()`. Jednak nadal można skorzystać z tak samo nazwanej metody z wyższych poziomów hierarchii; w tym celu nazwę metody należy poprzedzić nazwą klasy i operatorem zakresu, jak pokazano to w przypadku wywołania `BaseClass::f()` z metody `f()` w klasie `DerivedLevel2`.

Klasa pochodna sama też może zawierać dodatkowe pola danych. Klasa taka może stać się z kolei klasą bazową dla innej klasy i tak dalej; hierarchię dziedziczenia można dowolnie rozszerzać. Przykładowo, klasa `DerivedLevel1` pochodzi od `BaseClass`, ale jednocześnie jest klasą bazową dla `DerivedLevel2`.

W pokazanych przykładach dziedziczenie jest publiczne — po dwukropku w nagłówku definicji klasy pochodnej występuje słowo `public`. Dziedziczenie publiczne oznacza, że pola publiczne klasy bazowej będą w klasie pochodnej także publiczne, zaś chronione też będą chronione. W przypadku dziedziczenia chronionego (w nagłówku klasy po dwukropku pojawia się słowo kluczowe `protected`) pola i metody publiczne i chronione klasy bazowej w klasie pochodnej będą chronione. W końcu, jeśli dziedziczenie jest prywatne (słowo kluczowe `private`), pola publiczne i chronione klasy bazowej stają się prywatnymi polami klasy pochodnej. We wszystkich rodzajach

dziedziczenia metody i pola prywatne klasy bazowej we wszystkich klasach pochodnych są niedostępne. Przykładowo, próba wywołania `h()` z metody `f()` w klasie `Derived2Level1` powoduje błąd kompilacji „`BaseClass::h()` is not accessible". Jednak wywołanie z `f()` metody `h()` w klasie `DerivedLevel1` nie sprawia żadnych kłopotów, gdyż jest ono interpretowane jako wywołanie metody `h()` zdefiniowanej w klasie `DerivedLevel1`.

Pola i metody chronione klasy bazowej dostępne są jedynie dla klasy pochodnej tej klasy. Z tego powodu klasy `DerivedLevel1` i `Derived2Level1` mogą wywołać metodę chronioną `g()` klasy `BaseClass()`, ale wywołanie tej metody w funkcji `main()` jest już niedopuszczalne.

Klasa pochodna nie musi pochodzić od jednej tylko klasy bazowej. Przykładowo, klasa `DerivedLevel2` pochodzi od klas `DerivedLevel1` i `Derived2Level1`, dziedzicząc w ten sposób wszystkie metody obu tych klas. `DerivedLevel2` dziedziczy jednak te same metody z `BaseClass` dwukrotnie, gdyż obie klasy pochodne poziomu pierwszego pochodzą od klasy `BaseClass`. Jest to w najlepszym razie nadmiarowe i jako takie zbędne, a w najgorszym razie może spowodować błąd kompilacji „member is ambiguous `BaseClass::g()` and `BaseClass::g()`". Aby to się nie zdarzyło, definicje obu klas bazowych zawierają modyfikator `virtual`, który oznacza, że klasa `DerivedLevel2` zawiera jedną tylko kopię każdej metody klasy `BaseClass`. Analogiczny problem pojawi się, jeśli `f()` z `DerivedLevel2` wywoła `h()` bez podania operatora zakresu i nazwy klasy, tutaj `DerivedLevel1::h()`. Nie ma znaczenia fakt, że metoda `h()` w `BaseClass` jest prywatna i przez to niedostępna w `DerivedLevel2` — opisana sytuacja spowodowałaby wystąpienie błędu „member is ambiguous `DerivedLevel1::h()` and `BaseClass::h()`".

1.4. Wskaźniki

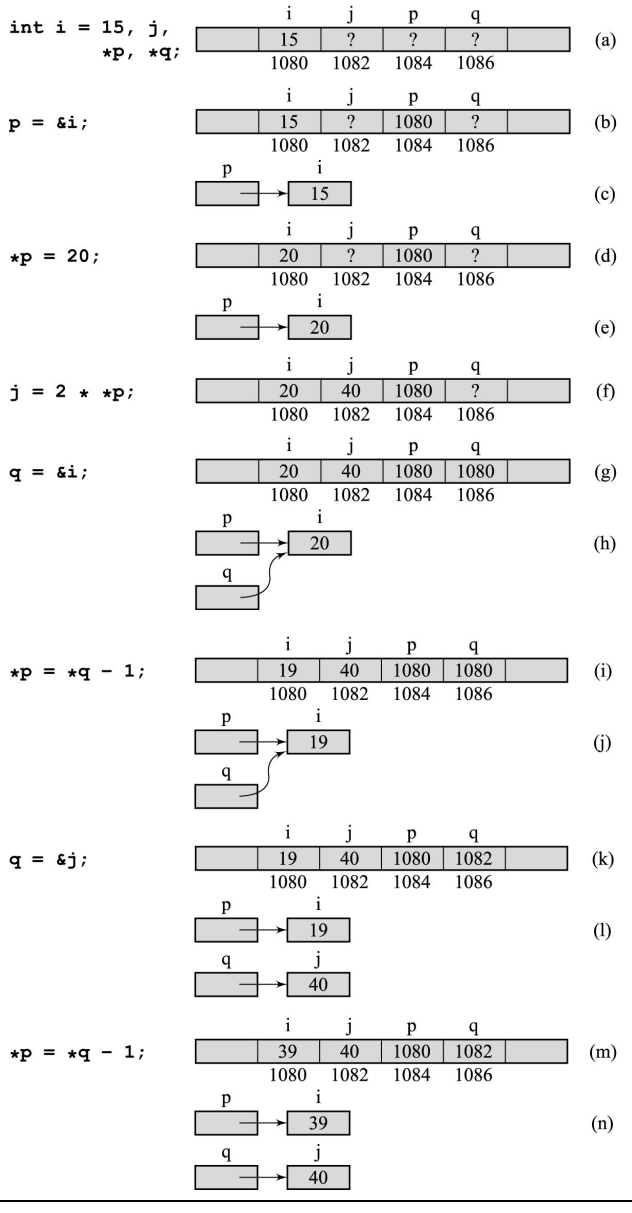
Zmienne używane w programie można traktować jako pojemniki, które nigdy nie są puste — zawsze coś jest w środku, albo wstawione przez programistę, albo, w przypadku braku inicjalizacji, przez system operacyjny. Każda taka zmienna ma co najmniej dwa atrybuty: zawartość czyli wartość oraz położenie pojemnika w pamięci komputera. Zawartość może być liczbą, znakiem lub wielkością złożoną, jak struktura czy unia. Jednak wartość ta może być też położeniem innej zmiennej; zmienna z taką wartością nazywane są *wskaźnikami*. Wskaźniki to zwykle zmienne pomocnicze, które umożliwiają nam pośrednie sięganie do innych wartości. Wskaźnik jest analogią do znaku drogowego, który wskazuje pewne miejsce, lub fiszki, na której zanotowano adres. Są to zmienne prowadzące do zmiennych; pomocnicze zmienne wskazujące wartości, które są naprawdę istotne.

Jeśli mamy następującą deklarację:

```
int i = 15, j, *p, *q;
```

`i` i `j` są zmiennymi liczbowymi, zaś `p` i `q` są wskaźnikami na te zmienne liczbowe; o ich roli decyduje poprzedzająca je gwiazdka. Zakładając, że adresy zmiennych `i`, `j`, `p` i `q` to, odpowiednio, 1080, 1082, 1084 i 1086, po przypisaniu zmiennej `i` wartości 15, uzyskamy w pamięci komputera obraz taki, jak na rysunku 1.1a.

Moglibyśmy teraz wykonać przypisanie `p = i` (lub `p = (int*) i`, gdyby kompilator nie chciał uznać pierwszej wersji), ale zmienna `p` została stworzona po to, aby przechowywać adres zmiennej liczbowej, a nie jej wartość. Wobec tego prawidłowe przypisanie to `p = &i`, gdzie symbol `&` występujący przed `i` oznacza, że chodzi o adres zmiennej `i`, a nie o jej wartość. Pokazano to na rysunku 1.1b. Na rysunku 1.1c strzałka z `p` do `i` wskazuje, że `p` jest wskaźnikiem zawierającym adres `i`.



RYSUNEK 1.1. Zmiany wartości po przypisaniach realizowane są przy użyciu wskaźników. Przypadki (b) i (c) opisują tę samą sytuację, tak samo (d) i (e), (g) i (h), (i) i (j), (k) i (l) oraz (m) i (n)

Musimy być w stanie odróżniać wartość p — adres — od wartości umieszczonej pod tym adresem. Aby na przykład przypisać wartość 20 zmiennej wskazywanej przez p trzeba użyć wyrażenia:

```
*p = 20;
```

Gwiazdka to operator dereferencji, który wymusza najpierw pobranie wartości p, potem sięgnięcie do wskazywanego przez tę wartość miejsca i wpisanie tam liczby 20 (rysunek 1.1d). Rysunki 1.1e do 1.1n to dalsze przykłady instrukcji przypisania, na których pokazano sposób umieszczania wartości w pamięci komputera.

Tak naprawdę wskaźniki, tak samo jak inne zmienne, mają dwa atrybuty: zawartość i położenie. Położenie można zapisać w innej zmiennej, która staje się wskaźnikiem na wskaźnik.

Na rysunku 1.1 adresy zmiennych są przypisywane wskaźnikom, jednak wskaźniki mogą odnosić się do anonimowych lokalizacji w pamięci dostępnych jedynie przez adres, a niemających nazw. Lokalizacje takie muszą być dynamicznie określane przez proces zarządzający pamięcią podczas wykonywania programu, podczas gdy położenie zmiennych określane jest już na etapie kompilacji.

Aby dynamicznie alokować i zwalniać pamięć, używa się dwóch funkcji. Jedna to `new`; pobiera ona z pamięci taki fragment, jaki jest potrzebny na zapisanie danej typu, który zostanie podany za słowem `new`. Przykładowo, instrukcja:

```
p = new int;
```

spowoduje zażądanie przez program pamięci potrzebnej na zamieszczenie jednej liczby całkowitej, zaś adres tej pamięci zostanie umieszczony w zmiennej `p`. Teraz można blokowi wskazywanemu przez `p` przypisywać wartości; trzeba to robić pośrednio, przez wskaźnik `p` lub dowolny inny wskaźnik `q`, któremu przypiszemy adres z `p` instrukcją `q = p`.

Kiedy pamięć zajmowana przez liczbę wskazywaną przez `p` przestaje być potrzebna, można ją zwolnić do puli obsługiwanej przez system operacyjny instrukcją:

```
delete p;
```

Jednak nawet po wykonaniu tej instrukcji zmienna `p` nadal zawiera adres zwolnionego bloku, choć blok ten przestaje być w programie dostępny. To tak, jakby traktować adres zburzonego domu jako adres możliwego zamieszkania. Każda próba znalezienia kogoś pod takim adresem jest z góry skazana na niepowodzenie. Analogicznie, jeśli po wywołaniu instrukcji `delete` nie usuniemy ze zmiennej wskaźnikowej adresu, narażamy się na ryzyko błędów; program może przestać działać podczas prób dostępu do nienależącej już do niego pamięci. Jest to *szczególnie niebezpieczne* w przypadku sięgania do obiektów bardziej złożonych niż liczby. Aby uniknąć opisanego problemu, należy wskaźnikowi przypisać nowy adres; jeśli nie ma żadnego prawidłowego adresu na podorędziu, należy użyć adresu pustego, o wartości `0`. Po wykonaniu przypisania:

```
p = 0;
```

nie mówimy, że `p` wskazuje zero czy `NULL`, lecz że `p` ma wartość `NULL` (pustą).

1.4.1. Wskaźniki a tablice

W pokazanym powyżej przykładzie wskaźnik `p` wskazywał blok pamięci zawierający jedną liczbę całkowitą. Znacznie ciekawsza jest sytuacja, kiedy wskaźnik wskazuje strukturę danych tworzoną i modyfikowaną dynamicznie. Pozwala to obejść ograniczenia dotyczące tablic. W C++, podobnie jak większości innych języków programowania, tablice trzeba zawczasu zadeklarować. Wobec tego ich rozmiar musi być znany przed uruchomieniem programu, a zatem programista musi dość dużo wiedzieć o oprogramowanym problemie, gdyż jest to warunkiem poprawnego określenia wielkości tablicy. Jeśli tablica będzie zbyt duża, niepotrzebnie zajmowana przez nią pamięć będzie zmarnowana. Jeśli tablica będzie zbyt mała, dane będą umieszczane za jej końcem, co powoduje poważną awarię programu. Czasami jednak rozmiaru tablicy po prostu nie da się przewidzieć — w takiej sytuacji decyzję o wielkości tablicy i przez to alokowanej pamięci odkłada się do czasu wykonywania programu.

Do rozwiązania opisanego problemu używa się wskaźników. Spójrzmy na rysunek 1.1b. Wskaźnik `p` wskazuje adres 1080, ale można za jego pośrednictwem sięgnąć też pod adresy 1082, 1084 i tak dalej — jest to możliwe dzięki temu, że kolejne dane przesuwane są o taką samą wielkość. Aby, na przykład, sięgnąć do wartości zmiennej `j` sąsiadującej z `i`, wystarczy dodać do adresu `i` trzymanego w `p` rozmiar wartości zmiennej `j`; wtedy wskaźnik `p` będzie wskazywał adres zmiennej `j`. Tak właśnie w C++ obsługiwane są tablice.

Przyjrzyjmy się następującym deklaracjom:

```
int a[5], *p;
```

Deklaracje te mówią, że `a` jest wskaźnikiem bloku pamięci, który może zawierać pięć liczb całkowitych. Wskaźnik jest stały, to znaczy należy go traktować jako stały; wobec tego każda próba przypisania zmiennej `a` wartości, na przykład:

```
a = p;
```

lub:

```
a++;
```

traktowana jest jako błąd kompilacji. Zmienna `a` jest wskaźnikiem, więc do sięgnięcia do poszczególnych komórek tablicy `a` można używać zapisu wskaźnikowego. Tak oto pętlę dodającą wszystkie wartości z tej tablicy:

```
for (sum = a[0], i = 1; i < 5; i++)
    sum += a[i];
```

można zamienić na zapis wskaźnikowy tak:

```
for (sum = *a, i = 1; i < 5; i++)
    sum += *(a + i);
```

lub tak:

```
for (sum = *a, p = a+1; p < a+5; p++)
    sum += *p;
```

Zauważmy, że skoro `a+1` oznacza położenie następnej komórki tablicy `a`, to `a+1` jest odpowiednikiem `&a[1]`. Wobec tego, jeśli `a` wskazuje adres 1020, to `a+1` nie zawiera adresu 1021, lecz 1022, gdyż stosowana jest arytmetyka wskaźników zależna od typu wskazywanych danych. Jeśli dane są deklaracje:

```
char b[5];
long c[5];
```

i wiadomo, że `b` równe jest 1050, `c` równe jest 1055, to `b+1` równe jest 1051, gdyż jeden znak zajmuje jeden bajt, zaś `c+1` równe jest 1059, gdyż liczba typu `long` zajmuje cztery bajty. Jest to wynikiem faktu, że w arytmetyce wskaźników wyrażenie `c+i` oznacza adres `c+i*sizeof(long)`.

Dotąd tablica `a` była deklarowana statycznie i zawierała pięć komórek. Rozmiar takiej tablicy jest ustalony na cały czas działania programu. Tablice można deklarować także dynamicznie; wtedy używane są zmienne wskaźnikowe. Przykładowo, przypisanie:

```
p = new int[n];
```

powoduje zaalokowanie pamięci na n liczb całkowitych. Wskaźnik p można traktować jako zmienną tablicową i można stosować doń notację tablicową. Sumę liczb z tablicy p można wyznaczyć tak:

```
for (sum = p[0], i = 1; i < n; i++)
    sum += p[i];
```

Można zastosować też notację wskaźnikową normalnie:

```
for (sum = *p, i = 1; i < n; i++)
    sum += *(p+i);
```

lub korzystając z dwóch wskaźników:

```
for (sum = *p, q = p+1; q < p+n; q++)
    sum += *q;
```

p jest zmienną, więc można jej przypisać nową tablicę. Kiedy tablica wskazywana przez p przestaje być potrzebna, przypisaną jej pamięć można zwolnić instrukcją:

```
delete [] p;
```

Użycie pustej pary nawiasów kwadratowych jest ważne, gdyż informuje, że p wskazuje tablicę.

Bardzo ważnym rodzajem tablic są łańcuchy czyli tablice znaków. Istnieje wiele funkcji predefiniowanych przetwarzających łańcuchy. Nazwy tych funkcji zaczynają się od str, na przykład strlen(s) określająca długość łańcucha s, strcpy(s1,s2) kopiująca łańcuch s2 do s1. Trzeba pamiętać, że wszystkie te funkcje zakładają, że łańcuchy kończą się znakiem NULL, '\0'. I tak strcpy(s1,s2) kopiuje s2 tak długo, aż znajdzie w nim znak NULL. Jeśli programista tego znaku w łańcuchu nie umieści, kopiowanie zostanie przerwane dopiero po napotkaniu pierwszego takiego znaku gdziekolwiek w pamięci komputera, za adresem zmiennej s2. Oznacza to, że kopiowane będą znaki za s1, co w końcu może doprowadzić do awarii programu.

1.4.2. Wskaźniki a konstruktory kopiujące

Kiedy wskaźniki zapisane w polach obiektu nie będą podczas kopiowania obiektów prawidłowo obsłużone, trzeba liczyć się z problemami. Niech dana będzie taka oto definicja:

```
struct Node {
    char *name;
    int age;
    Node( char *n = "", int a = 0) {
        name = new char[strlen(n)+1];
        strcpy(name,n);
        age = a;
    }
};
```

Celem deklaracji:

```
Node node1("Roger",20), node2(node1); // lub node2 = node1;
```

jest stworzenie obiektu `node1`, przypisanie wartości dwu jego polom, a następnie stworzenie obiektu `node2` i zainicjalizowanie jego pól takimi samymi wartościami jak `node1`. Obiekty muszą być od siebie niezależne, więc przypisanie wartości pól jednego obiektu nie powinno wpływać na drugi obiekt. Jednak po przypisaniu:

```
strcpy(node2.name,"Wendy");
node2.age = 30;
```

przy próbie zajrzenia do zawartości obiektów:

```
cout<<node1.name<<' '<<node1.age<<' '<<node2.name<<' '<<node2.age;
```

otrzymamy:

```
Wendy 30 Wendy 20
```

Wiek jest różny, ale imię jest takie samo. Co się właściwie stało? Chodzi o to, że w definicji klasy `Node` nie podano konstruktora kopiującego:

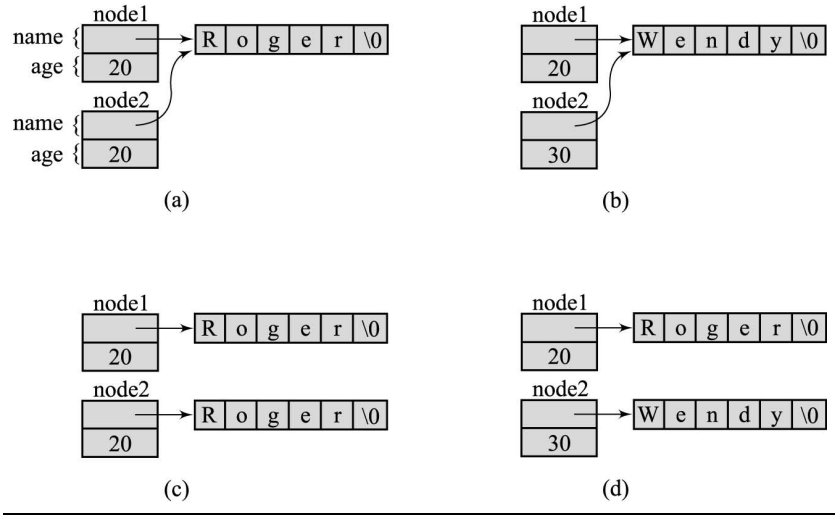
```
Node(const Node&);
```

który jest niezbędny do wykonania inicjalizacji `node2(node1)`. Jeśli użytkownik nie poda konstruktora kopiującego, konstruktor taki zostanie automatycznie wygenerowany przez kompilator. Jednak konstruktor stworzony przez kompilator kopiuje pole po polu. Pole `name` jest wskaźnikiem, więc skopiowany zostanie adres łańcucha `node1.name` do `node2.name`, a zawartość łańcucha już kopiowana nie będzie. Wobec tego po wykonaniu pokazanej wcześniej deklaracji sytuacja będzie się przedstawiała tak jak na rysunku 1.2a. Jeśli teraz wykonane zostaną przypisania:

```
strcpy(node2.name,"Wendy");
node2.age = 30;
```

pole `node2.age` zostanie przypisane prawidłowo, ale łańcuch `"Roger"` wskazywany przez pole `name` obu obiektów zostanie nadpisany napisem `"Wendy"` wskazywanym przez oba obiekty (rysunek 1.2b). Aby do takiej sytuacji nie dopuścić, konieczne jest zdefiniowanie prawidłowego konstruktora kopiującego:

```
struct Node {
    char * name;
    int age;
    Node(char *n = 0, int a = 0) {
        name = new char[strlen(n)+1];
        strcpy(name,n);
        age = a;
    }
```



RYSUNEK 1.2. Dlaczego w przypadku obiektów zawierających pola wskaźnikowe konieczne jest użycie konstruktora kopiującego

```
    Node(const Node& n) { // konstruktor kopiujący;
        name = new char[strlen(n.name)+1];
        strcpy(name,n.name);
        age = n.age;
    }
};
```

Kiedy dany jest nowy konstruktor, `node2(node1)` wygeneruje kopię napisu `"Roger"` wskazywanego przez `node2.name` (rysunek 1.2c), a przypisanie pól jednego obiektu nie wpływa na pola drugiego obiektu:

```
strcpy(node2.name,"Wendy");
node2.age = 30;
```

Obiekt `node1` pozostanie niezmieniony, co widać na rysunku 1.2d.

Podobne problemy pojawiają się w przypadku użycia operatora przypisania. Jeśli użytkownik nie poda definicji operatora przypisania, instrukcja:

```
node1 = node2;
```

spowoduje przypisanie kolejno każdego pola, co spowoduje problemy analogiczne jak na rysunku 1.2a – b. Aby tych problemów uniknąć, konieczne jest przeciążenie operatora przypisania. W przypadku klasy `Node` przeciążenie może wyglądać następująco:

```
Node& operator=(const Node& n) {
    if (this != &n) { // nie przypisujemy obiektu do samego siebie
        if (name != 0)
            delete [] name;
        name = new char[strlen(n.name)+1];
        strcpy(name,n.name);
        age = n.age;
```

```
    }
    return *this;
}
```

W powyższym fragmencie kodu użyto specjalnego wskaźnika `this`. Każdy obiekt może za pomocą tego wskaźnika uzyskać swój własny adres, wobec czego `*this` to sam obiekt.

1.4.3. Wskaźniki i destruktory

Co się dzieje z lokalnie definiowanymi obiektami typu `Node`? Tak jak wszystkie inne zmienne lokalne, są niszczone w tym sensie, że poza blokiem, w którym je zdefiniowano, stają się niedostępne, zaś zajmowana przez nie pamięć jest zwalniana. Jednak, mimo że pamięć zajmowana przez obiekt `Node` jest zwalniana, to nie cała zaalokowana pamięć zostanie zwolniona. Jedno z pól tego obiektu to wskaźnik łańcucha, wobec czego zwolniona zostanie pamięć przeznaczona na wskaźnik, ale nie zostanie zwolniona pamięć na sam łańcuch. Po zniszczeniu obiektu dostępny dotąd łańcuch z pola `name` staje się niedostępny (o ile nie został przypisany polu `name` innego obiektu lub całkiem innej zmiennej łańcuchowej), więc nie można już nawet zwolnić pamięci przyporządkowanej temu łańcuchowi. Jest to problem związany z polami wskazującymi dynamicznie alokowaną pamięć. Aby tego problemu uniknąć, definicja klasy powinna zawierać definicję destruktora. *Destruktor* to funkcja automatycznie wywoływana w chwili niszczenia obiektu, które to niszczenie odbywa się w chwili wyjścia z bloku, w którym obiekt zdefiniowano, lub w chwili wywołania `delete`. Destruktory nie mają żadnych parametrów ani wartości zwracanych, więc w każdej klasie istnieć może jeden tylko destruktor. W przypadku klasy `Node` będzie on wyglądał następująco:

```
~Node() {
    if (name != 0)
        delete [] name;
}
```

1.4.4. Wskaźniki a referencje

Przyjrzyjmy się następującym deklaracjom:

```
int n = 5, *p = &n, &r = n;
```

Zmienna `p` została zadeklarowana jako zmienna typu `int*`, wskaźnik na liczbę całkowitą, zaś `r` jako zmienna typu `int&` — referencja do liczby całkowitej. Zmienna referencji musi być zainicjalizowana w chwili deklaracji referencją do konkretnej zmiennej, wobec czego nigdy nie może być pusta. Zmienną referencji `r` można traktować jako inną nazwę zmiennej `n`, jeśli zatem zmieni się `n`, to zmieni się też wartość `r`. Wynika to stąd, że zmienne referencji implementuje się jako stałe wskaźniki do zmiennych.

Wywołanie instrukcji wyświetlającej dane po powyższych deklaracjach:

```
cout << n << ' ' << *p << ' ' << r << endl;
```

da w wyniku 5 5 5. Po przypisaniu:

```
n = 7;
```

ta sama instrukcja da już 7 7 7. Po przypisaniu:

```
*p = 9;
```

da 9 9 9, zaś po przypisaniu:

```
r = 10;
```

da 10 10 10. Z pokazanych instrukcji wynika, że to, co można uzyskać, stosując dereferencję wskaźników, można też uzyskać, stosując dereferencję zmiennych będących referencjami. Nie jest to przypadek, gdyż jak wspomniano wcześniej, referencje są implementowane jako stałe wskaźniki. Zamiast deklaracji:

```
int& r = n;
```

można zapisać:

```
int *const r = &n;
```

gdzie `r` jest stałym wskaźnikiem na liczbę całkowitą, czyli przypisanie:

```
r = q;
```

gdzie `q` jest kolejnym wskaźnikiem, jest błędne, gdyż wartość `r` nie może się zmieniać. Jednak przypisanie:

```
*r = 1;
```

już jest dopuszczalne, jeśli tylko `n` nie jest stałą liczbą całkowitą.

Trzeba tu odnotować różnicę między daną typu `int *const` a typu `const int *`. Ten drugi typ to wskaźnik na stałą liczbę całkowitą:

```
const int *s = &m;
```

i po takiej deklaracji przypisanie:

```
s = &m;
```

gdzie `m` jest liczbą całkowitą (stałą lub nie), jest dopuszczalne, zaś przypisanie:

```
*s = 2;
```

jest błędne, nawet jeśli `m` nie jest stałą.

Zmienne będące referencjami są wykorzystywane przy przykazywaniu do funkcji parametrów przez referencję. Przekazywanie przez referencję jest potrzebne, jeśli podczas wykonywania

funkcji przekazany parametr powinien zostać trwale zmieniony. Ten sam efekt można uzyskać, stosując wskaźniki (w języku C jest to jedyny mechanizm przekazywania przez referencję). Na przykład zadeklarowanie funkcji:

```
void f1(int i, int* j, int& k) {
    i = 1;
    *j = 2;
    k = 3;
}
```

powoduje, że zmienne:

```
int n1 = 4, n2 = 5, n3 = 6;
```

po wywołaniu:

```
f1(n1,&n2,n3);
```

mają wartości: `n1` = 4, `n2` = 2, `n3` = 3.

Typy referencyjne są też używane do wskazywania rodzaju wartości zwracanych przez funkcje. Jeśli dana jest na przykład funkcja:

```
int& f2(int a[], int i) {
    return a[i];
}
```

i zadeklarowana zostanie tablica:

```
int a[] = {1,2,3,4,5};
```

funkcji `f2()` można użyć z dowolnej strony operatora przypisania, po stronie prawej:

```
n = f2(a,3);
```

lub po stronie lewej:

```
f2(a,3) = 6;
```

co spowoduje przypisanie liczby 6 do `a[3]`, czyli `a` = [1 2 3 6 5]. Zauważmy, że taki sama efekt można uzyskać za pomocą wskaźników, ale w takim przypadku konieczne byłoby użycie jawnej dereferencji:

```
int* f3(int a[], int i) {
    return &a[i];
}
```

a potem:

```
*f3(a,3) = 6;
```

1.4.5. Wskaźniki na funkcje

Jak wspomniano w punkcie 1.4.1, jednym z atrybutów zmiennej jest adres określający położenie tej zmiennej w pamięci. To samo dotyczy funkcji — jeden z atrybutów funkcji to adres określający położenie treści tej funkcji w pamięci. Po wywołaniu funkcji system przekazuje sterowanie do tego miejsca po to, aby wykonać tę funkcję. Z tego powodu można używać wskaźników na funkcje. Wskaźniki te są bardzo przydatne przy implementowaniu funkcjonałów (czyli funkcji otrzymujących jako parametry inne funkcje), takie jak całki.

Niech dana będzie prosta funkcja:

```
double f(double x) {
    return 2*x;
}
```

W przypadku takiej definicji `f` jest wskaźnikiem na funkcję `f()`, `*f` to sama funkcja, zaś `(*f)(7)` to wywołanie tej funkcji.

Teraz zastanówmy się nad funkcją C++ wyliczającą następującą sumę:

$$\sum_{i=n}^{i=m} f(i)$$

Aby wyliczyć sumę, trzeba podać nie tylko wartości graniczne *n* i *m*, ale też funkcję *f*. Wobec tego zastosowana implementacja pozwolić przekazywać nie tylko parametry liczbowe, ale także funkcyjne. W C++ robi się to następująco:

```
double sum(double (*f)(double), int n, int m) {
    double result = 0;
    for(int i = n; i <= m; i++)
        result += f(i);
    return result;
}
```

W podanej definicji funkcji `sum()` deklaracja pierwszego parametru formalnego:

```
double (*f)(double)
```

oznacza, że `f` jest wskaźnikiem na funkcję mającej jeden parametr typu `double` i zwracającej wartość `double`. Dookoła `*f` konieczne jest zastosowanie nawiasów; mają one wyższy priorytet niż operator dereferencji, *, więc wyrażenie:

```
double *f(double)
```

to deklaracja funkcji zwracającej wskaźnik na wartość typu `double`.

Teraz funkcja `sum()` może być wywołana z dowolną wbudowaną lub zdefiniowaną przez użytkownika funkcję mającą parametr typu `double` i taki parametr zwracającą:

```
cout << sum(f,1,5) << endl;
cout << sum(sin,3,7) << endl;
```

Innym przykładem może być funkcja znajdująca pierwiastek funkcji ciągłej na przedziale. Pierwiastek znajduje się wielokrotnie, dzieląc przedział na pół i znajdując środek aktualnego przedziału. Jeśli jest on zerem lub jeśli przedział jest mniejszy od pewnej, założonej małej wartości, zwracany jest ten środek. Jeśli wartości funkcji na lewym brzegu przedziału i w jego środku mają przeciwne znaki, przeszukiwanie kontynuowane jest w lewej połowie przedziału. W przeciwnym przypadku przedziałem bieżącym staje się prawa połowa przedziału dotychczasowego. Oto implementacja algorytmu:

```
double root(double (*f)(double), double a, double b, double epsilon) { // pierwiastek
   double middle = (a + b) / 2;
    while (f(middle) != 0 && fabs(b - a) > epsilon) {
        if (f(a) * f(middle) < 0)   // jeśli f(a) i f(middle) mają
            b = middle;             // przeciwne znaki;
        else a = middle;
        middle = (a + b) / 2;
    }
    return middle;
}
```

1.5. Polimorfizm

Polimorfizm oznacza możliwość przybierania wielu postaci. W kontekście programowania obiektowego oznacza on, że ta sama nazwa funkcji odpowiada wielu funkcjom należącym do różnych obiektów. Oto przykład:

```
class Class1 {
public:
    virtual void f() {
        cout << "Funkcja f() w klasie Class1\n";
    }
    void g() {
        cout << "Funkcja g() w klasie Class1\n";
    }
};
class Class2 {
public:
    virtual void f() {
        cout << "Funkcja f() w klasie Class2\n";
    }
    void g() {
        cout << "Funkcja g() w klasie Class2\n";
    }
};
class Class3 {
public:
    virtual void h() {
        cout << "Funkcja h() w klasie Class3\n";
    }
```

```
};
int main() {
    Class1 object1, *p;
    Class2 object2;
    Class3 object3;
    p = &object1;
    p->f();
    p->g();
    p = (Class1*) &object2;
    p->f();
    p->g();
    p = (Class1*) &object3;
//  p->f(); // Awaryjne przerwanie programu;
    p->g();
//  p->h(); // h() nie jest metodą klasy Class1;
    return 0;
}
```

Wynik działania powyższego programu jest następujący:

```
Funkcja f() w klasie Class1
Funkcja g() w klasie Class1
Funkcja f() w klasie Class2
Funkcja g() w klasie Class1
Funkcja g() w klasie Class1
```

Nie powinien zaskakiwać fakt, że kiedy `p` jest zadeklarowane jako wskaźnik na obiekt `object1` klasy `Class1`, uruchamiane są dwie funkcje zadeklarowane w klasie `Class1`. Jednak kiedy `p` staje się wskaźnikiem na obiekt `object2` klasy `Class2`, `p->f()` aktywuje funkcję zdefiniowaną w klasie `Class2`, zaś `p->g()` aktywuje funkcję z klasy `Class1`. Dlaczego? Chodzi o to, kiedy podejmowana jest decyzja, która funkcja zostanie wywołana.

W przypadku tak zwanego *wiązania statycznego* decyzja o wyborze wykonywanej funkcji podejmowana jest na etapie kompilacji. W przypadku *wiązania dynamicznego* decyzja ta jest odkładana do czasu wykonania programu. W C++ wiązanie dynamiczne jest wymuszane przez zadeklarowanie metody jako wirtualnej, słowem kluczowym `virtual`. W ten sposób, kiedy wywoływana jest metoda wirtualna, wybór funkcji na etapie wykonania programu zależny jest nie od typu wskaźnika określonego w deklaracji, ale od typu wskaźnika faktycznie użytego. W pokazanym przykładzie wskaźnik `p` zadeklarowano jako wskaźnik typu `Class1*`. Wobec tego, jeśli `p` wskazuje funkcję `g()` i nie jest to funkcja wirtualna, to niezależnie od tego, w którym miejscu wystąpi `p->g()`, zawsze jest ono traktowane jako wywołanie funkcji `g()` z klasy `Class1`. Wynika to stąd, że kompilator podejmuje decyzję na podstawie deklaracji typu wskaźnika `p` i na podstawie faktu, że funkcja `g()` nie jest zadeklarowana ze słowem kluczowym `virtual`. W przypadku metod wirtualnych sytuacja zasadniczo się zmienia. Tym razem decyzja podejmowana jest na etapie wykonywania programu — jeśli metoda jest wirtualna, system określa typ wskaźnika na bieżąco i wywołuje odpowiednią metodę. Początkowo, `p` zostało zadeklarowane jako wskaźnik typu `Class1*`, więc wywoływana jest funkcja wirtualna `f()` z klasy `Class1`. Po przypisaniu `p` adresu obiektu `object2` klasy `Class2` wywoływana jest metoda `f()` klasy `Class2`.

Trzeba też odnotować, że po przypisaniu `p` adresu obiektu `Object3`, nadal wywoływana jest metoda `g()` z klasy `Class1`. Wynika to stąd, że `g()` nie zdefiniowano ponownie w klasie `Class3`, wobec czego używana jest definicja z klasy `Class1`. Jednak próba wywołania `p->f()` powoduje

awarię programu, gdyż funkcję `f()` zadeklarowano jako wirtualną w klasie `Class1`; system szuka jej zatem (bezskutecznie) w klasie `Class3`. Poza tym, mimo że `p` wskazuje obiekt `object3`, instrukcja `p->h()` powoduje błąd kompilacji, gdyż kompilator nie znajduje w klasie `Class1` metody `h()`, a `p` nadal jest wskaźnikiem typu `Class1*`. Z punktu widzenia kompilatora nie ma znaczenia, że `h()` zdefiniowano w klasie `Class3` — czy to jako metodę wirtualną czy nie.

Polimorfizm jest potężnym narzędziem programowania obiektowego. Wystarczy wysłać standardowy komunikat do wielu różnych obiektów, nie podając, jak komunikat ma być interpretowany. Nie trzeba znać typów obiektów. Odbiorca jest odpowiedzialny za interpretację komunikatu i jego wykonanie. Nadawca nie musi modyfikować komunikatu w zależności od typu odbiorcy. Zbędne jest stosowanie jakichkolwiek instrukcji warunkowych. Poza tym do złożonych programów można bezproblemowo dodawać nowe moduły bez konieczności ponownej kompilacji całego programu.

1.6. C++ a programowanie obiektowe

Dotąd zakładaliśmy, że C++ jest językiem programowania obiektowego, wszystkie możliwości programowania obiektowego były ilustrowane kodem C++. Jednak C++ nie jest językiem czysto obiektowym. C++ jest bardziej obiektowy niż C czy Pascal, które nie mają żadnych cech obiektowości. C++ jest też bardziej obiektowy niż Ada obsługująca klasy (pakiety) i instancje. Jednak język C++ jest w mniejszym stopniu obiektowy niż języki *stricte* obiektowe, jak Smalltalk czy Eiffel.

C++ nie zmusza do programowania obiektowego. Możemy programować w C++ bez znajomości obiektowych cech tego języka. Powodem tego jest ogromna popularność języka C. C++ stanowi nadzbiór C, więc programiści C nie mają problemów z „przesiadką” na C++; mogą stosować tylko łatwiejsze w użyciu operacje wejścia i wyjścia, mechanizm wywołania przez referencję, domyślne parametry funkcji, przeciążanie operatorów, funkcje *inline* i im podobne. Użycie obiektowego języka programowania typu C++ nie gwarantuje programowania obiektowego. Z drugiej strony wywoływanie całej maszynerii klas i metod nie zawsze jest konieczne, szczególnie w przypadku niewielkich programów. Wobec tego możliwość niekorzystania z obiektowości nie musi być wadą. Poza tym C++ jest łatwiejsze w integrowaniu z istniejącym kodem C niż inne języki programowania obiektowego.

C++ zapewnia doskonały mechanizm enkapsulacji pozwalający dobrze kontrolować ukrywanie informacji. Jednak od tej reguły istnieją wyjątki — chodzi o „funkcje zaprzyjaźnione”. Informacje prywatne należące do pewnej klasy nie mogą być dla nikogo dostępne, zaś informacje publiczne są dostępne dla wszystkich. Jednak czasami dobrze byłoby pozwolić niektórym użytkownikom na dostęp do danych prywatnych. Można to zrealizować, jeśli klasa pokazuje funkcje użytkownika jako funkcje zaprzyjaźnione. Jeśli na przykład dana jest następująca definicja klasy:

```
class C {
    int n;
    friend int f();
} ob;
```

to funkcja `f()` ma bezpośredni dostęp do zmiennej `n` należącej do klasy `C`:

```
int f ()
{   return 10 * ob.n; }
```

Można byłoby tę sytuację uznać za naruszenie zasady ukrywania informacji. Jednak to sama klasa `C` daje dostęp do swoich części prywatnych wybranym funkcjom, pozostałym te dane pozostają niedostępne. Wobec tego, skoro klasa ma kontrolę nad tym, które funkcje będą uważane za zaprzyjaźnione, mechanizm funkcji zaprzyjaźnionych można uznać za rozszerzenie zasady ukrywania informacji. Mechanizm ten jest używany do ułatwienia programowania i do przyspieszenia wykonywania programu, gdyż przepisywanie kodu bez użycia funkcji zaprzyjaźnionych mogłoby być nader kłopotliwe. Takie rozluźnienie niektórych zasad nie jest w informatyce czymś niezwykłym; wystarczy wspomnieć istnienie pętli w językach funkcyjnych (jak LISP) czy przechowywanie dodatkowych informacji w nagłówku pliku danych będące naruszeniem modelu relacyjnego (jak w dBase III+).

1.7. Standardowa biblioteka szablonów

C++ jest językiem obiektowym, ale niedawne rozszerzenia języka wynoszą C++ na wyższy poziom. Najistotniejszym dodatkiem do języka jest standardowa biblioteka szablonów (STL, *Standard Template Library*) początkowo stworzona przez Alexandra Stepanova i Menga Lee. Biblioteka ta zawiera trzy rodzaje bytów ogólnych: kontenery, iteratory i algorytmy. Algorytmy są często używanymi funkcjami, które można stosować do różnych struktur danych. W ich stosowaniu pośredniczą iteratory decydujące, które algorytmy mają być stosowane do jakiego typu obiektów. STL zwalnia programistów z pisania własnych implementacji rozmaitych klas i funkcji; od razu gotowe są ogólne implementacje rozwiązania poszczególnych problemów.

1.7.1. Kontenery

Kontener to struktura danych zawierająca obiekty, zwykle wszystkie tego samego typu. Różne rodzaje kontenerów różnie układają obiekty. Wprawdzie liczba możliwych ułożeń jest teoretycznie nieograniczona, ale jedynie niewielka część tych ułożeń ma znaczenie praktyczne; właśnie te najczęściej używane zostały zamieszczone w bibliotece STL. Biblioteka ta zawiera następujące kontenery: `deque`, `list`, `map`, `multimap`, `set`, `multiset`, `stack`, `queue`, `priority_queue` i `vector`.

Kontenery STL zaimplementowano jako szablony klas zawierające metody umożliwiające opisanie operacji, które będą wykonywane na obiektach umieszczonych w strukturze lub na samej strukturze. Niektóre operacje są wspólne dla wszystkich kontenerów, choć mogą być one różnie zaimplementowane w różnych strukturach. Do metod wspólnych dla wszystkich kontenerów należą: konstruktor, konstruktor kopiujący, destruktor, `empty()`, `max_size()`, `size()`, `swap()`, `operator=` i (za wyjątkiem kolejki `priority_queue`) sześć operatorów relacyjnych: `operator<` i tak dalej. Poza tym wszystkie kontenery poza `stack`, `queue` i `priority_queue` mają metody `begin()`, `end()`, `rbegin()`, `rend()`, `erase()` i `clear()`.

Elementy umieszczane w kontenerach mogą być dowolnego typu — muszą zapewniać przynajmniej konstruktor domyślny, destruktor i operator przypisania. Jest to szczególnie istotne w przypadku typów definiowanych przez użytkownika. Niektóre kompilatory mogą też wymagać przeciążenia niektórych operatorów relacyjnych (zwykle == i <, czasem też != i >), nawet jeśli operatory te nie będą w programie używane. Poza tym, jeśli pola są wskaźnikami, powinny być zdefiniowane konstruktor kopiujący i `operator=`, gdyż wstawianie korzysta z kopii elementu, a nie samego elementu.

1.7.2. Iteratory

Iterator to obiekt używany do wskazania elementu z kontenera. Iteratory to swojego rodzaju uogólnienie wskaźników; iterator pozwala sięgnąć do danych z kontenera tak, aby umożliwić wykonanie na tych danych potrzebnych operacji.

Będąc uogólnieniem wskaźników, iteratory mają taki sam zapis dereferencji; na przykład `*i` to element wskazywany przez iterator `i`. Poza tym arytmetyka iteratorów jest podobna do arytmetyki wskaźników, choć nie wszystkie operacje na iteratorach dozwolone są na wszystkich kontenerach.

Nie istnieją iteratory kontenerów `stack`, `queue` ani `priority_queue`. Operacje na iteratorach dla klas `list`, `map`, `multimap`, `set` i `multiset` są następujące (przy czym `i1` i `i2` to iteratory, `n` jest liczbą):

```
i1++, ++i1, i1--, --i1
i1 = i2
i1 == i2, i1 != i2
*i1
```

Poza tym dla klas `deque` i `vector` istnieją jeszcze operacje:

```
i1 < i2, i1 <= i2, i1 > i2, i1 >= i2
i1 + n, i1 - n
i1 += n, i1 -= n
i1[n]
```

1.7.3. Algorytmy

Biblioteka STL zawiera ponad 70 ogólnych funkcji nazywanych algorytmami; mogą być one stosowane do kontenerów STL i tablic. Listę wszystkich algorytmów z biblioteki STL zamieszczono w dodatku B. Algorytmy te implementują operacje często używane w typowych programach, takie jak znalezienie elementu w kontenerze, wstawienie elementu do ciągu elementów, usunięcie elementu z ciągu, modyfikacja i porównywanie elementów, znajdowanie wartości według ciągu elementów, sortowania ciągu i tak dalej. Niemalże wszystkie algorytmy biblioteki STL do wskazywania zakresu elementów, na których działają, używają iteratorów. Pierwszy iterator wskazuje pierwszy element zakresu, drugi — element *po* ostatnim elemencie zakresu. Wobec tego zakłada się, że zawsze można dojść do miejsca wskazywanego przez drugi iterator przez inkrementację iteratora pierwszego. Oto kilka przykładów.

Wywołanie:

```
random_shuffle(c.begin(), c.end());
```

zmieni losowo kolejność elementów w kontenerze `c`. Wywołanie:

```
i3 = find(i1, i2, e1);
```

zwraca iterator wskazujący położenie elementu `e1` w zakresie od `i1` do `i2` (bez samego `i2`). Wywołanie:

```
n = count_if(i1, i2, oddNum);
```

zlicza algorytmem `count_if()` elementy z zakresu wskazanego iteratorami `i1` i `i2`, dla których jednoparametrowa funkcja użytkownika `oddNum()` zwraca wartość `true`.

Algorytmy to funkcje, które stanowią dodatki do metod kontenerów. Jednak niektóre algorytmy zdefiniowano także jako metody, w celu uzyskania szybszego ich działania.

1.7.4. Funktory

W C++ operator wywołania funkcji `()` może być traktowany jak każdy inny operator, w szczególności może być przeciążany. Może zwracać daną dowolnego typu i mieć dowolną liczbę parametrów, ale tak samo jak operator przypisania, może być przeciążany tylko jako metoda klasy. Każdy obiekt zawierający definicję operatora wywołania funkcji nazywany jest *funktorem*. Funktor to taki obiekt, który zachowuje się, jakby był funkcją. Kiedy obiekt taki jest wywoływany, jego parametry stają się parametrami operatora wywołania funkcji.

Zajmijmy się, dla przykładu, znalezieniem sumy liczb uzyskiwanych z zastosowania funkcji *f* do liczb całkowitych z przedziału [*n*, *m*]. Implementacja `sum()`, pokazana w punkcie 1.4.5, bazowała na użyciu jako parametru funkcji `sum()` wskaźnika do funkcji. To samo można uzyskać, definiując najpierw klasę z przeciążonym operatorem wywołania funkcji:

```
class classf {
    public:
    classf() {
    }
    double operator() (double x) {
        return 2*x;
    }
};
```

i definiując funkcję:

```
double sum2(classf f, int n, int m) {
    double result = 0;
    for (int i = n; i <= m; i++)
        result += f(i);
    return result;
}
```

różniącą się od funkcji `sum()` tylko pierwszym parametrem, który teraz jest funktorem, a nie funkcją; poza tym wszystko inne jest bez zmian. Nowa funkcja może być teraz wywołana następująco:

```
classf cf;
cout << sum2(cf,2,5) << endl;
```

lub po prostu:

```
cout << sum2(classf(),2,5) << endl;
```

Druga metoda wywołania wymaga zdefiniowania konstruktora `classf()` (mimo że nie ma on ciała) tworzącego obiekt typu `classf()` w chwili wywołania `sum2()`.

To samo można uzyskać, nie przeciążając operatora wywołania funkcji, jak poniżej:

```
class classf2 {
public:
    classf2 () {
    }
    double run (double x) {
        return 2*x;
    }
};
double sum3 (classf2 f, int n, int m) {
    double result = 0;
    for (int i = n; i <= m; i++)
        result += f.run(i);
    return result;
}
```

gdzie wywołanie przybierze postać:

```
cout << sum3(classf2(),2,5) << endl;
```

Biblioteka STL w dużej mierze oparta jest na funktorach. Mechanizm wskaźników na funkcje nie wystarcza w przypadku operatorów wbudowanych. Jak można byłoby przekazać unarny minus do funkcji `sum()`? Składnia `sum(-,2,5)` jest niepoprawna. Aby uniknąć opisanego problemu, w bibliotece STL zdefiniowano funktory `<functional>` wspólne dla wszystkich operatorów C++. Przykładowo, minus unarny zdefiniowano następująco:

```
template<class T>
struct negate : public unary_function<T,T> {
    T operator()(const T& x) const {
        return -x;
    }
};
```

Teraz, po ponownym zdefiniowaniu funkcji `sum()`, staje się ona już funkcją ogólną:

```
template<class F>
double sum(F f, int n, int m) {
    double result = 0;
    for (int i = n; i <= m; i++)
        result +=f(i);
    return result;
}
```

Można też wywołać funkcję z funktorem `negate`: `sum(negate<double>(),2,5)`.

1.8. Wektory w standardowej bibliotece szablonów

Najprostszy kontener STL to wektor (`vector`) — struktura umieszczana w kolejnych blokach pamięci, podobnie jak tablica. Wobec tego, że bloki pamięci są umieszczane kolejno, można do nich sięgać w dowolnej kolejności, zaś czas dostępu do każdego z elementów jest taki sam. Zarządzanie pamięcią jest wykonywane automatycznie, więc w przypadku próby wstawienia nowego element do pełnego wektora na wektor ten alokowany jest większy obszar pamięci, elementy wektora są wstawiane w nowe miejsce, a obszar dotychczasowy jest zwalniany. Wobec tego wektor to elastyczna tablica, której wielkość może być dynamicznie zmieniana.

W tabeli 1.1 zestawiono w kolejności alfabetycznej wszystkie metody kontenera `vector`. Zastosowanie wektora pokazano na listingu 1.1. Zawartość wektorów, na których wykonywane są operacje, oznaczono na listingu komentarzami. Zawartość wektorów pokazywana jest ogólną funkcją `printVector()`, ale w programie z listingu 1.1 pokazano tylko jedno jej wywołanie.

TABELA 1.1.
Alfabetyczna lista metod klasy vector

Metoda	Operacja
`void assign(first, last)`	Usuwa wszystkie węzły wektora i wstawia do tego wektora elementy z zakresu określonego iteratorami `first` i `last`.
`void assign(n, el = T())`	Usuwa wszystkie węzły wektora, wstawia do tego wektora `n` kopii `el`.
`T& at(n)`	Zwraca `n`-ty element wektora.
`const T& at(n) const`	Zwraca `n`-ty element wektora.
`T& back()`	Zwraca ostatni element wektora.
`const T& back() const`	Zwraca ostatni element wektora.
`iterator begin()`	Zwraca iterator wskazujący pierwszy element wektora.
`const_iterator begin() const`	Zwraca iterator wskazujący pierwszy element wektora.
`size_type capacity() const`	Zwraca liczbę elementów wektora.
`void clear()`	Usuwa z wektora wszystkie jego elementy.
`bool empty() const`	Jeśli wektor nie zawiera żadnych elementów, zwraca `true`. W przeciwnym razie zwraca `false`.
`iterator end()`	Zwraca iterator wskazujący pozycję za ostatnim elementem wektora.
`const_iterator end() const`	Zwraca iterator wskazujący pozycję za ostatnim elementem wektora.
`iterator/erase(i)`	Usuwa element wskazywany przez iterator `i` i zwraca iterator wskazujący pozycję elementu występującego po elemencie usuniętym.
`iterator/erase(first, last)`	Usuwa elementy z zakresu określonego przez iteratory `first` i `last` i zwraca iterator wskazujący pozycję elementu występującego po elemencie wskazywanym przez iterator `last`.
`T& front()`	Zwraca pierwszy element wektora.
`const T& front() const`	Zwraca pierwszy element wektora.
`iterator insert(i, el = T())`	Wstawia `el` przed elementem wskazywanym przez iterator `i`, zwraca iterator wskazujący nowowstawiony element.
`void insert(i, n, el)`	Wstawia `n` kopii elementu `el` przed element wskazywany przez iterator `i`.
`void insert(i, first, last)`	Wstawia przed węzłem wskazywanym iteratorem `i` elementy z zakresu wskazywanego iteratorami `first` i `last`.
`size_type max_size() const`	Zwraca maksymalną liczbę elementów wektora.
`T& operator[]`	Operator indeksowania.
`const T& operator[] const`	Operator indeksowania.
`void pop_back()`	Usuwa z wektora ostatni element.

TABELA 1.1.
Alfabetyczna lista metod klasy vector (ciąg dalszy)

Metoda	Operacja
`void push_back(el)`	Wstawia element `el` na koniec wektora.
`reverse_iterator rbegin()`	Zwraca iterator wskazujący ostatni element wektora.
`const_reverse_iterator rbegin() const`	Zwraca iterator wskazujący ostatni element wektora.
`reverse_iterator rend()`	Zwraca iterator wskazujący przed pierwszy element wektora.
`const_reverse_iterator rend() const`	Zwraca iterator wskazujący przed pierwszy element wektora.
`void reserve(n)`	Zwraca miejsce wystarczające, aby wektor mógł przechować `n` elementów, jeśli pojemność jest mniejsza od `n`.
`void resize(n, el = T())`	Powoduje, że wektor może przechowywać `n` elementów; robi się to, dodając `n - size()` pozycji zawierających element `el` lub odrzucając zbędne `size() - n` pozycji z końca wektora.
`size_type size() const`	Zwraca liczbę elementów wektora.
`void swap(v)`	Zamienia zawartość jednego wektora na zawartość innego wektora.
`vector()`	Konstruktor pustego wektora.
`vector(n, el = T())`	Konstruktor wektora zawierającego `n` kopii elementu `el` typu `T` (jeśli nie podano `el`, używany jest konstruktor domyślny `T()`).
`vector(first, last)`	Konstruktor wektora z elementami z zakresu wskazywanego iteratorami `first` i `last`.
`vector(v)`	Konstruktor kopiujący.

LISTING 1.1.
Program pokazujący użycie metod klasy vector

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional> // greater<T>

using namespace std;

template<class T>
void printVector(char *s, const vector<T>& v) {
    cout << s << " = (";
    if (v.size() == 0) {
        cout << ")\n";
        return;
    }
    typename vector<T>::const_iterator i = v.begin();
    for ( ; i != v.end()-1; i++)
        cout << *i << ' ';
    cout << *i << ")\n";
}

bool f1(int n) {
    return n < 4;
}
```

```
int main() {
    int a[] = {1,2,3,4,5};
    vector<int> v1;      // v1 jest pusty, ilość elementów = 0, pojemność = 0
    printVector("v1",v1);
    for (int j = 1; j <= 5; j++)
        v1.push_back(j); // v1 = (1 2 3 4 5), ilość elementów = 5, pojemność = 8
    vector<int> v2(3,7); // v2 = (7 7 7)
    vector<int>::iterator i1 = v1.begin()+1;
    vector<int> v3(i1,i1+2); // v3 = (2 3), ilość elementów = 2, pojemność = 2
    vector<int> v4(v1); // v4 = (1 2 3 4 5), ilość elementów = 5, pojemność = 5
    vector<int> v5(5); // v5 = (0 0 0 0 0)
    v5[1] = v5[3] = 9; // v5 = (0 9 0 9 0)
    v3.reserve(6); // v3 = (2 3), ilość elementów = 2, pojemność = 6
    v4.resize(7); // v4 = (1 2 3 4 5 0 0), ilość elementów = 7, pojemność = 10
    v4.resize(3); // v4 = (1 2 3), ilość elementów = 3, pojemność = 10
    v4.clear(); // v4 jest pusty, ilość elementów = 0, pojemność = 10 (!)
    v4.insert(v4.end(),v3[1]);                  // v4 = (3)
    v4.insert(v4.end(),v3[1]);                  // v4 = (3 3)
    v4.insert(v4.end(),2,4);                    // v4 = (3 3 4 4)
    v4.insert(v4.end(),v1.begin()+1,v1.end()-1); // v4 = (3 3 4 4 2 3 4)
    v4.erase(v4.end()-2);                       // v4 = (3 3 4 4 2 4)
    v4.erase(v4.begin(), v4.begin()+4);         // v4 = (2 4)
    v4.assign(3,8);                             // v4 = (8 8 8)
    v4.assign(a,a+3);                           // v4 = (1 2 3)
    vector<int>::reverse_iterator i3 = v4.rbegin();
    for ( ; i3 != v4.rend(); i3++)
        cout << *i3 << ' ';                     // wyświetl: 3 2 1
    cout << endl;

//  algorytmy

    v5[0] = 3;                                  // v5 = (3 9 0 9 0)
    replace_if(v5.begin(),v5.end(),f1,7);       // v5 = (7 9 7 9 7)
    v5[0] = 3; v5[2] = v5[4] = 0;               // v5 = (3 9 0 9 0)
    replace(v5.begin(),v5.end(),0,7);           // v5 = (3 9 7 9 7)
    sort(v5.begin(),v5.end());                  // v5 = (3 7 7 9 9)
    sort(v5.begin(),v5.end(),greater<int> ());  // v5 = (9 9 7 7 3)
    v5.front() = 2;                             // v5 = (2 9 7 7 3)
    return 0;
}
```

Jeśli w programie ma być użyta klasa `vector`, program musi zawierać dyrektywę:

```
#include <vector>
```

Klasa `vector` ma cztery konstruktory. Deklaracja:

```
vector<int> v5(5);
```

korzysta z tego samego konstruktora, co deklaracja:

```
vector<int> v2(3,7);
```

ale w przypadku wektora v5 element, którym jest wypełniany ten wektor, jest określony przez domyślny konstruktor całkowitoliczbowy, czyli jest to zero.

Wektor v1 zadeklarowany jest jako pusty, a potem wstawiane są doń elementy funkcją push_back(). Dodanie do wektora nowego elementu zwykle działa szybko, chyba że wektor jest już pełny i musi być skopiowany do nowego bloku pamięci. Z taką sytuacją mamy do czynienia wtedy, wielkość wektora równa jest jego pojemności. Jeśli jednak wektor zawiera jakieś nieużyte komórki, nowy element może zostać dopisany w stałym czasie. Aktualne parametry wektora można sprawdzać funkcjami size() (zwraca liczbę elementów umieszczonych w wektorze) oraz capacity() (zwraca liczbę dostępnych komórek wektora). W razie potrzeby liczbę dostępnych komórek można zmieniać funkcją reserve(). Przykładowo, wywołanie:

```
v3.reserve(6);
```

powoduje, że wektor v3 = (2 3) zachowa te same elementy i aktualną wielkość równą 2, choć jego maksymalny rozmiar zmieni się z 2 na 6. Funkcja reserve() wpływa jedynie na maksymalną pojemność wektora, a nie na jego zawartość. Funkcja resize() wpływa na zawartość i ewentualnie także pojemność. Przykładowo, jeśli dla wektora v4 = (1 2 3 4 5) rozmiar jest równy pojemności równej 5, po wykonaniu wywołania:

```
v4.resize(7);
```

zawartość v4 zmieni się na (1 2 3 4 5 0 0), rozmiar na 7, zaś pojemność na 10; po kolejnym wywołaniu resize():

```
v4.resize(3);
```

wektor v4 będzie zawierał (1 2 3), jego rozmiarem będzie 3, zaś pojemnością 10. Oznacza to, że na wektor została zaalokowana nowa pamięć, ale nie została już ona zwolniona.

Warto zwrócić uwagę na brak metody push_front(). Jest to echo tego, że dodawanie nowego elementu na początek wektora jest skomplikowaną operacją, gdyż wymaga przesunięcia wszystkich elementów o jeden, aby zrobić miejsce na nowy element. Jest to zatem czasochłonna operacja, ale można ją wykonać metodą insert(). Jest to jednocześnie funkcja alokująca w razie potrzeby większą ilość pamięci na wektor. Inne funkcje wykonujące podobne zadania to konstruktory, funkcje reserve() oraz operator=.

Elementy wektora dostępne są przez indeksowanie charakterystyczne dla tablic:

```
v4[0] = n;
```

lub przez iteratory; w tym wypadku korzysta się ze wskaźnikowej notacji dereferencji:

```
vector<int>::iterator i4 = v4.begin();
*i4 = n;
```

Niektóre metody jako typ zwracany mają T& (czyli referencję). Przykładowo, jeśli wektor zawiera liczby całkowite, sygnatura metody front() ma postać:

```
int& front();
```

Oznacza to, że metody `front()` można użyć zarówno po lewej, jak i po prawej stronie operatora przypisania:

```
v5.front() = 2;
v4[1] = v5.front();
```

Metoda `front()` to przykład metody przeciążonej, gdyż zwracana może być referencja do wartości lub stała referencja. Aby dostrzec różnicę między nimi, przyjrzyjmy się następującym dwóm przypisaniom:

```
int& n1 = v5.front();        // użyto T& front()
const int& n2 = v5.front();  // użyto const T& front() const
```

Co istotne, `front()` zwraca wartość, a nie referencję do wartości, mimo że w deklaracji występuje `T&`. Aby dostrzec różnicę, spójrzmy na jeszcze jedno przypisanie:

```
int n3 = v5.front();
```

W tej chwili zmienne mają następujące wartości:

```
n1 = 2, n2 = 2, n3 = 2
```

Jednak po przypisaniu:

```
v5.front() = 3;
```

te same zmienne mają już wartości:

```
n1 = 3, n2 = 3, n3 = 2
```

Do wektorów można stosować wszystkie algorytmy biblioteki STL. Przykładowo, wywołanie:

```
replace(v5.begin(),v5.end(),0,7);
```

zastępuje w wektorze `v5` wszystkie zera siódemkami, tak że `v5` = (3 9 0 9 0) zamienia się w `v5` = (3 9 7 9 7), zaś wywołanie:

```
sort(v5.begin(),v5.end());
```

sortuje wektor `v5` rosnąco. Niektóre algorytmy pozwalają korzystać z parametrów będących funkcjonałami. Przykładowo, jeśli program zawiera definicję następującej funkcji:

```
bool f1(int n) {
    return n < 4;
}
```

to wywołanie:

```
replace_if(v5.begin(),v5.end(),f1,7);
```

powoduje zastosowanie funkcji `f1()` do wszystkich elementów `v5` i zastąpienie elementów mniejszych od 4 wartością 7. W tym wypadku wektor `v5` = (3 9 0 9 0) zamienia się w wektor `v5` = (7 9 7 9 7). Nieco mniej czytelny program, ale robiący to samo bez konieczności jawnego definiowania funkcji `f1`, wygląda następująco:

```
replace_if(v5.begin(),v5.end(),bind2nd(less<int>(),4),7);
```

W powyższym wyrażeniu `bind2nd(op,a)` to funkcja ogólna zachowująca się tak, jakby przekształcała dwuargumentowy funktor w funktor jednoargumentowy przez podanie (powiązanie) drugiego parametru. W tym celu tworzony jest dwuargumentowy funktor, w którym dwuargumentowa operacja `op` jako drugi parametr pobiera `a`.

Równie elastyczny jest algorytm sortujący. W przykładzie z sortowaniem wektora `v5` sortowanie przeprowadzane jest rosnąco. Jak można posortować wektor malejąco? Jednym sposobem jest posortowanie go rosnąco i następnie odwrócenie go algorytmem `reverse()`. Inny sposób to wymuszenie na `sort()` stosowania operatora `>` podczas porównywania. Można to zrobić bezpośrednio, podając jako parametr funktor:

```
sort(v5.begin(),v5.end(),greater<int>());
```

Można też podać funkcję pośrednio:

```
sort(v5.begin(),v5.end(),f2);
```

przy czym definicja `f2` wygląda tak:

```
bool f2(int m, int n) {
    return m > n;
}
```

Poleca się pierwszą metodę, ale można ją stosować tylko dlatego, że funktor `greater` już zdefiniowano w bibliotece STL. Ten funktor jest zdefiniowany jako struktura szablonowa, która w sposób ogólny przeciąża operator `>`. Wobec tego `greater<int>()` oznacza, że operator ma być stosowany do liczb całkowitych.

Pokazana wersja algorytmu `sort()`, pobierająca parametr funkcyjny, jest szczególnie przydatna, kiedy trzeba posortować obiekty bardziej złożone niż liczby i konieczne jest użycie różnych kryteriów porównywania wartości. Przyjrzyjmy się następującej definicji klasy:

```
class Person {
public:
    Person(char *n = "", int a = 0) {
        name = new char[strlen(n)+1];
        strcpy(name,n);
        age = a;
    }
    bool operator==(const Person& p) const {
        return strcmp(name,p.name) == 0 && age == p.age;
    }
    bool operator<(const Person& p) const {
        return strcmp(name,p.name) < 0;
    }
    bool operator>(const Person& p) const {
        return !(*this == p) && !(*this < p);
    }
```

```
private:
    char *name;
    int age;
    friend bool lesserAge(const Person&, const Person&);
};
```

Teraz, mając deklarację:

```
vector<Person> v6(1,Person("Gregg",25));
```

dodajemy do `v6` jeszcze dwa obiekty:

```
v6.push_back(Person("Ann",30));
v6.push_back(Person("Bill",20));
```

i wykonujemy:

```
sort(v6.begin(),v6.end());
```

Zawartość `v6` zmieni się z („Gregg”, 25) („Ann”, 30) („Bill”, 20)) na („Ann”, 30) („Bill”, 20) („Gregg”, 25)), czyli nastąpi zwykłe sortowanie rosnąco, gdyż metoda `sort()` mająca jako jedyne parametry dwa iteratory korzysta z operatora < przeciążonego w klasie `Person`. Wywołanie:

```
sort(v6.begin(),v6.end(),greater<Person>());
```

zmieni `v6` = („Ann”, 30) („Bill”, 20) („Gregg”, 25)) na `v6` = („Gregg”, 25) („Bill”, 20) („Ann”, 30)), czyli sortowanie odbyło się w kolejności malejącej; operator > został przeciążony dla badanej klasy. Jak teraz posortować obiekty według wieku? W tym wypadku potrzebna będzie odpowiednia funkcja porównująca:

```
bool lesserAge(const Person& p1, const Person& p2) {
    return p1.age < p2.age;
}
```

którą można będzie następująco przekazać jako parametr `sort()`:

```
sort(v6.begin(),v6.end(),lesserAge);
```

Teraz zamiast `v6` = („Gregg”, 25) („Bill”, 20) („Ann”, 30)) otrzymamy `v6` = („Bill”, 20) („Gregg”, 25) („Ann”, 30)).

1.9. Struktury danych a programowanie obiektowe

Wprawdzie działanie komputerów opiera się na bitach, ale zwykle nie bity się analizuje; byłoby to nader nieporęczne. Mimo że liczba całkowita jest ciągiem — dajmy na to — 16 bitów, lepiej patrzeć na nią jako na osobny byt, na którym można wykonywać operacje charakterystyczne właśnie dla liczb całkowitych. Jako że liczba całkowita składa się z bitów, inne obiekty mogą używać liczb całkowitych jako swoich elementarnych części składowych. Niektóre typy danych są z założenia wbudowane w poszczególne języki, zaś inne muszą być zdefiniowane przez użytkownika. Nowy

typ danych ma swoją specyficzną budowę, nowe rozłożenie elementów; jego budowa wyznacza zachowanie obiektów nowego typu. Analiza struktur danych pozwala badać nowe struktury, sprawdzać ich szybkość działania i wymogi odnośnie pamięci. W przeciwieństwie do programowania obiektowego, gdzie wychodzi się od zachowań obiektów i następnie próbuje znaleźć najodpowiedniejszą strukturę danych, tym razem zaczyna się od opisu struktury danych i sprawdzenia, jakie są jej możliwości i jak szybko ona działa. Badanie struktur danych służy tworzeniu narzędzi, które są potem włączane do aplikacji i tworzeniu struktur danych, które mogą wykonać żądane operacje, nie obciążając zanadto komputera. Badania struktur koncentrują się wokół analizy mechaniki klas, ich budowy wewnętrznej, która zwykle jest dla późniejszych użytkowników klasy całkowicie niewidoczna. Badania struktur danych polegają na badaniu możliwości łączenia różnych klas ze sobą, na próbach poprawiania sprawności klas przez doskonalenie ich wewnętrznych struktur. W ten sposób można dokładniej powiedzieć użytkownikowi, do czego poszczególne klasy mogą służyć i jak ich można używać. Dzięki dziedziczeniu użytkownik może dodawać nowe operacje do klas i próbować z klas „wycisnąć" więcej niż zaplanował projektant. Struktury danych pozostają przed użytkownikiem ukryte, więc nowe operacje można testować, uruchamiając je, ale nie ma się dostępu do struktur wewnętrznych (chyba że dostępny jest kod źródłowy).

Badania struktur danych są najskuteczniejsze, jeśli stosuje się do nich metodykę obiektową. W ten sposób można tworzyć narzędzia, nie ryzykując, że narzędzia będą potem niewłaściwie wykorzystywane. Zamykając struktury danych w klasie i udostępniając jedynie to, co jest niezbędne do użycia klasy, można stworzyć narzędzia, których funkcje nie będą ograniczane przez sztuczne przeszkody.

1.10. Przykład zastosowania: plik z dostępem swobodnym

Niniejszy przykład służy głównie zilustrowaniu użycia klas ogólnych i dziedziczenia. Biblioteka STL będzie stosowana w przykładach zastosowań także w dalszej części książki.

Z punktu widzenia sytemu operacyjnego pliki to zbiory bajtów, niezależnie od tego, co zawierają. Z punktu widzenia użytkownika pliki to słowa, liczby, ciągi danych, rekordy i tak dalej. Jeśli użytkownik chce sięgnąć do piątego słowa w pliku tekstowym, procedura wyszukująca musi przejść po całym pliku od położenia 0 i sprawdzać wszystkie bajty. Zliczana jest liczba sekwencji znaków odstępu i po pominięciu czterech takich sekwencji (lub pięciu, jeśli sekwencja znaków odstępu rozpoczyna plik), zliczanie się kończy; napotkanie piątej sekwencji normalnych znaków jest równoważne napotkaniu piątego słowa. Słowo to może zaczynać się w dowolnym miejscu pliku. W idealnej sytuacji możliwe byłoby przechodzenie do dowolnego miejsca w pliku i zagwarantowanie, że jest się na początku piątego słowa. Problem ze znalezieniem odpowiedniego miejsca wynika z tego, że różne słowa mają różne długości, tak samo sekwencje znaków odstępu. Gdyby było wiadomo, że każde słowo zajmuje tyle samo pamięci, można byłoby przejść bezpośrednio do słowa piątego, wybierając położenie $4 \cdot$ długość(słowo). Jako że słowa mają jednak różne długości, każdemu trzeba byłoby przypisać tyle samo bajtów. W przypadku słów krótszych konieczne byłoby dopełniania słowa dodatkowymi znakami. Dłuższe słowa byłyby przycinane. W ten sposób plik otrzymałby całkiem inną organizację — przestałby być zwykłym zbiorem bajtów, ale stałby się zbiorem rekordów. W naszym przykładzie każdy rekord składałby się z jednego słowa. Jeśli ktoś zażądałby piątego słowa, można byłoby przejść od razu do niego, nie czytając słów poprzednich. I tak oto stworzyliśmy plik o dostępie swobodnym.

Plik o dostępie swobodnym pozwala sięgnąć bezpośrednio do dowolnego rekordu. Rekordy zwykle zawierają więcej danych niż jedno słowo. Pokazany wcześniej przykład zasugerował już sposób tworzenia pliku o dostępie swobodnym — przy użyciu rekordów o stałej długości. W przypadku

niniejszego przykładu zastosowania celem jest napisanie ogólnego programu generującego plik o dostępie swobodnym dla dowolnego typu rekordu. Działanie programu zilustrowano przykładem z rekordami danych o osobach. Każdy rekord zawiera pięć elementów: numer ubezpieczenia społecznego, nazwisko, miasto, rok urodzenia i pobory. Druga ilustracja to plik ze studentami; dane są takie same jak w pliku z osobami, ale dodatkowo rejestrowany jest opiekun naukowy. W ten sposób możliwe będzie pokazanie mechanizmu dziedziczenia.

W niniejszym przypadku program pozwala wstawiać do pliku o dostępie swobodnym nowy rekord, znajdować rekord w pliku i modyfikować go. Nazwa pliku musi być podana przez użytkownika; jeśli plik nie zostanie znaleziony, zostanie utworzony. Jeśli plik zostanie znaleziony, zostanie otwarty do czytania i pisania. Sam program pokazano na listingu 1.2.

LISTING 1.2.
Program zarządzający plikami o dostępie swobodnym

```
//************************  personal.h  ******************************
#ifndef PERSONAL
#define PERSONAL

#include <fstream.h>
#include <string.h>

class Personal {
public:
    Personal();
    Personal(char*,char*,char*,int,long);
    void writeToFile(fstream&) const;
    void readFromFile(fstream&);
    void readKey();
    int size() const {
        return 9 + nameLen + cityLen + sizeof(year) + sizeof(salary);
    }
    bool operator==(const Personal& pr) const {
        return strcmp(pr.SSN,SSN) == 0;
    }
protected:
    const int nameLen, cityLen;
    char SSN[10], *name, *city;
    int year;
    long salary;
    ostream& writeLegibly(ostream&);
    friend ostream& operator<<(ostream& out, Personal& pr) {
        return pr.writeLegibly(out);
    }
    istream& readFromConsole(istream&);
    friend istream& operator>>(istream& in, Personal& pr) {
        return pr.readFromConsole(in);
    }
};

#endif
```

```
//************************  personal.cpp  ****************************
#include "personal.h"

Personal::Personal() : nameLen(10), cityLen(10) {
    name = new char[nameLen+1];
    city = new char[cityLen+1];
}

Personal::Personal(char *ssn, char *n, char *c, int y, long s) :
        nameLen(10), cityLen(10) {
    name = new char[nameLen+1];
    city = new char[cityLen+1];
    strcpy(SSN,ssn);
    strcpy(name,n);
    strcpy(city,c);
    year = y;
    salary = s;
}

void Personal::writeToFile(fstream& out) const {
    out.write(SSN,9);
    out.write(name,nameLen);
    out.write(city,cityLen);
    out.write(reinterpret_cast<const char*>(&year),sizeof(int));
    out.write(reinterpret_cast<const char*>(&salary),sizeof(int));
}

void Personal::readFromFile(fstream& in) {
    in.read(SSN,9);
    in.read(name,nameLen);
    in.read(city,cityLen);
    in.read(reinterpret_cast<char*>(&year),sizeof(int));
    in.read(reinterpret_cast<char*>(&salary),sizeof(int));
}

void Personal::readKey() {
    char s[80];
    cout << "Podaj SSN: ";
    cin.getline(s,80);
    strncpy(SSN,s,9);
}

ostream& Personal::writeLegibly(ostream& out) {
    SSN[9] = name[nameLen] = city[cityLen] = '\0';
    out << "Numer SSN = " << SSN << ", Nazwisko = " << name
        << ", Miasto = " << city << ", Rok urodzenia = " << year
        << ", Pobory = " << salary;
    return out;
}

istream& Personal::readFromConsole(istream& in) {
    char s[80];
```

```
    cout << "Numer SSN: ";
    in.getline(s,80);
    strncpy(SSN,s,9);
    cout << "Nazwisko: ";
    in.getline(s,80);
    strncpy(name,s,nameLen);
    cout << "Miasto: ";
    in.getline(s,80);
    strncpy(city,s,cityLen);
    cout << "Rok urodzenia: ";
    in >> year;
    cout << "Pobory: ";
    in >> salary;
    in.getline(s,80); // get '\n'
    return in;
}

//************************  student.h  ******************************

#ifndef STUDENT
#define STUDENT

#include "personal.h"

class Student : public Personal {
public:
    Student();
    Student(char*,char*,char*,int,long,char*);
    void writeToFile(fstream&) const;
    void readFromFile(fstream&);
    int size() const {
        return Personal::size() + majorLen;
    }
protected:
    char *major;
    const int majorLen;
    ostream& writeLegibly(ostream&);
    friend ostream& operator<<(ostream& out, Student& sr) {
        return sr.writeLegibly(out);
    }
    istream& readFromConsole(istream&);
    friend istream& operator>>(istream& in, Student& sr) {
        return sr.readFromConsole(in);
    }
};

#endif

//************************  student.cpp  ****************************

#include "student.h"
```

```
Student::Student() : majorLen(10) {
    Personal();
    major = new char[majorLen+1];
}

Student::Student(char *ssn, char *n, char *c, int y, long s, char *m) :
        majorLen(11) {
    Personal(ssn,n,c,y,s);
    major = new char[majorLen+1];
    strcpy(major,m);
}

void Student::writeToFile(fstream& out) const {
    Personal::writeToFile(out);
    out.write(major,majorLen);
}

void Student::readFromFile(fstream& in) {
    Personal::readFromFile(in);
    in.read(major,majorLen);
}

ostream& Student::writeLegibly(ostream& out) {
    Personal::writeLegibly(out);
    major[majorLen] = '\0';
    out << ", Opiekun = " << major;
    return out;
}

istream& Student::readFromConsole(istream& in) {
    Personal::readFromConsole(in);
    char s[80];
    cout << "Opiekun: ";
    in.getline(s,80);
    strncpy(major,s,9);
    return in;
}

//*********************  database.h  ****************************

#ifndef DATABASE
#define DATABASE

template<class T>
class Database {
public:
    Database();
    void run();
private:
    fstream database;
    char fName[20];
    ostream& print(ostream&);
```

```
    void add(T&);
    bool find(const T&);
    void modify(const T&);
    friend ostream& operator<<(ostream& out, Database& db) {
        return db.print(out);
    }
};

#endif

//************************  database.cpp  ****************************

#include <iostream.h>
#include <fstream.h>
#include "personal.h"
#include "student.h"
#include "database.h"

template<class T>
Database<T>::Database() {
    cout << "Nazwa pliku: ";
    cin >> fName;
}

template<class T>
void Database<T>::add(T& d) {
    database.open(fName,ios::in|ios::out|ios::binary);
    database.seekp(0,ios::end);
    d.writeToFile(database);
    database.close();
}

template<class T>
void Database<T>::modify(const T& d) {
    T tmp;
    database.open(fName,ios::in|ios::out|ios::binary);
    while (!database.eof()) {
        tmp.readFromFile(database);
        if (tmp == d) {  // przeciążony ==
             cin >> tmp; // przeciążony >>
             database.seekp(-d.size(),ios::cur);
             tmp.writeToFile(database);
             database.close();
             return;
        }
    }
    database.close();
    cout << "Modyfikowanego rekordu nie ma w bazie danych\n";
}

template<class T>
```

```
bool Database<T>::find(const T& d) {
    T tmp;
    database.open(fName,ios::in|ios::binary);
    while (!database.eof()) {
        tmp.readFromFile(database);
        if (tmp == d) { // przeciążony ==
            database.close();
            return true;
        }
    }
    database.close();
    return false;
}

template<class T>
ostream& Database<T>::print(ostream& out) {
    T tmp;
    database.open(fName,ios::in|ios::binary);
    while (1) {
        tmp.readFromFile(database);
        if (database.eof())
            break;
        out << tmp << endl; // przeciążony <<
    }
    database.close();
    return out;
}

template<class T>
void Database<T>::run() {
    char option[5];
    T rec;
    cout << "1. Dodaj 2. Znajdź 3. Modyfikuj rekord; 4. Koniec\n";
    cout << "Podaj opcję: ";
    cin.getline(option,4); // pobierz opcję wraz z '\n';
    while (cin.getline(option,4)) {
        if (*option == '1') {
             cin >> rec;   // przeciążony >>
             add(rec);
        }
        else if (*option == '2') {
             rec.readKey();
             cout << "Rekord ";
             if (find(rec) == false)
                 cout << "nie ";
             cout << "istnieje w bazie danych\n";
        }
        else if (*option == '3') {
             rec.readKey();
             modify(rec);
        }
        else if (*option != '4')
             cout << "Nieprawidłowa opcja\n";
```

```
        else return;
        cout << *this;   // przeciążony <<
        cout << "Podaj opcję: ";
    }
}

void main() {
    Database<Personal> db;
//  Database<Student> db;
    db.run();
    return 0;
}
```

Funkcja `find()` sprawdza, czy rekord jest w pliku. Szukanie jest przeprowadzane sekwencyjnie przez porównywanie każdego odczytanego rekordu `tmp` z szukanym rekordem `d` za pomocą przeciążonego operatora równości ==. Funkcja korzysta z faktu istnienia rekordów o stałej długości, porównując właśnie rekordy, a nie poszczególne bajty. Aby było jasne — rekordy składają się z bajtów, wszystkie bajty należące do żądanych rekordów muszą być odczytane, ale porównywane są jedynie uwzględniane przez operator równości.

Funkcja `modify()` aktualizuje dane zapisane w konkretnym rekordzie. Rekord jest najpierw pobierany z pliku przez wyszukiwanie, potem nowe informacje są pobierane od użytkownika za pomocą przeciążonego operatora >>. Aby zachować poprawiony rekord `tmp` w pliku, funkcja `modify()` wymusza przejście wskaźnika pliku `database` wstecz, do początku odczytanego rekordu `tmp`. W przeciwnym razie nadpisany zostałby rekord znajdujący się za `tmp`. Pozycja początkowa `tmp` może być od razu określona, gdyż każdy rekord zajmuje dokładnie tyle samo bajtów, wobec czego wystarczy przeskoczyć wstecz o tyle bajtów, ile ma jeden rekord. Robi się to, wywołując metodę `database.seekp(-d.size(),ios::cur)`, gdzie metoda `size()` musi być zdefiniowana dla klasy `T` — klasy, która jest typem obiektu `d`.

Ogólna klasa `Database` ma dwie dodatkowe funkcje. Funkcja `add()` umieszcza rekord na końcu pliku. Funkcja `print()` pokazuje zawartość pliku.

Aby zobaczyć użycie klasy `Database`, trzeba zdefiniować konkretną klasę opisującą format pojedynczych rekordów w pliku z danymi. W ramach przykładu zdefiniowano klasę `Personal` mającą pola `SSN`, `name`, `city`, `year` oraz `salary` (odpowiednio numer ubezpieczenia społecznego, nazwisko, miasto, rok, pobory). Pierwsze trzy pola to łańcuchy, przy czym SSN zawsze ma stałą długość, stąd w deklaracji podano rozmiar: `char SSN[10]`. Nieco większa swoboda panuje przy opisie pozostałych dwóch łańcuchów; użyto dwóch stałych, `nameLen` oraz `cityLen`, których wartości są wykorzystywane w konstruktorach. Przykładowo:

```
Personal::Personal() : nameLen(10), cityLen(10) {
    name = new char[nameLen+1];
    city = new char[cityLen+1];
}
```

Zauważmy, że stałych nie można inicjalizować, stosując przypisania:

```
Personal::Personal() {
    nameLen = cityLen = 10;
    name = new char[nameLen+1];
    city = new char[cityLen+1];
}
```

jednak takiej dziwnej składni używanej w C++ do inicjalizowania stałych w klasach można używać też do inicjalizacji zmiennych.

Zapis danych z obiektu wymaga szczególnej troski; jest to zadanie funkcji `writeToFile()`. Pole `SSN` jest najprostsze w obsłudze. Numer taki zawsze ma dziewięć cyfr, więc można użyć po prostu operatora <<, jednak długości nazwiska i nazwy miasta oraz pola przeznaczone na te dane w pliku też powinny zawsze mieć tę samą długość. Zagwarantowanie tego wymaga użycia funkcji `write()`, na przykład `out.write(name,nameLen)`; w ten sposób można nakazać wpisanie do pliku żądanej liczby znaków wraz z końcowym znakiem `'\0'`, niedopisywanym przez operator <<.

Kolejny problem wiąże się z danymi liczbowymi, `year` i `salary`; szczególnie z drugą z nich. Gdyby pobory wpisywać do pliku operatorem <<, pobory 50000 miałyby pięć bajtów (`'50000'`), zaś pobory 100000 — sześć bajtów (`'100000'`). Stanowi to ewidentne naruszenie zasady, że każdy rekord w pliku o dostępie swobodnym ma mieć tę samą długość. Aby uniknąć tego typu problemów, liczby zapisuje się w formie binarnej. Przykładowo, 50000 jest zapisywane jako łańcuch 32 bitów: 00000000000000001100001101010000 (zakładając, że zmienne typu `long` zajmują cztery bajty). Teraz taki ciąg można traktować nie jako ciąg bitów, ale jako cztery znaki: 00000000, 00000000, 11000011, 01010000 zapisane jako kody ASCII odpowiednio 0, 0, 195 i 80. W ten sposób, niezależnie od wartości poborów, wartość ta zawsze zajmie cztery bajty. Do zapisu służy nam instrukcja:

```
out.write(reinterpret_cast<const char*>(&salary),sizeof(long));
```

wymuszająca na funkcji `write()` potraktowanie poborów jako czterobajtowego łańcucha; dzieje się tak dzięki przekształceniu adresu `&salary` na typ `const char*` i podaniu długości typu `long`.

Podobnie odczytujemy dane z pliku; służy do tego metoda `readFromFile()`. Łańcuchy, które mają być danymi liczbowymi, muszą być odpowiednio przekształcone. Jeśli chodzi o pole `salary`, odczyt wygląda następująco:

```
in.read(reinterpret_cast<char*>(&salary),sizeof(long));
```

Robione jest tu rzutowanie `&salary` na `char*`, nakazywany jest odczyt czterech bajtów (`sizeof(long)`).

Zaprezentowana metoda zapisywania rekordów jest dość nieczytelna, szczególnie w przypadku liczb. Przykładowo, 50000 zapisywane jest na czterech bajtach — dwóch zerowych, jednym znaku specjalnym i wielkiej literze P. Jeśli plik danych będzie czytany przez człowieka, trudno będzie odgadnąć, że chodzi o liczbę 50 000. Wobec tego pokazywanie rekordów w czytelnej postaci wymaga specjalnej procedury. Po to przeciążony jest operator <<, który używa funkcji pomocniczej `writeLegibly()`. Klasa bazy danych też przeciąża operator << przy użyciu własnej funkcji pomocniczej `print()`. Funkcja ta wczytuje kolejne dane z pliku do obiektu `tmp` (metoda `readFromFile()`) i za pomocą operatora << pokazuje `tmp` w postaci czytelnej. Dlatego właśnie w programie używane są dwie funkcje do odczytu i dwie do zapisywania danych: jedna służy do obsługi danych w pliku, druga do czytania i pisania danych w postaci czytelnej.

W celu zbadania, na ile elastyczna jest klasa `Database`, zdefiniowana została kolejna klasa użytkowa, `Student`. Klasa ta służy też jako przykład zastosowania dziedziczenia.

Klasa `Student` korzysta z tych samych pól danych co klasa `Personal` (bo jest zdefiniowana jako dziedzicząca po `Personal`), dodatkowo ma jeszcze =jedno pole, łańcuch `major`. Przetwarzanie obiektów klasy `Student` jako danych wejściowych i wyjściowych jest bardzo podobne do przetwarzania obiektów klasy `Personal`, tyle że trzeba uwzględnić dodatkowe pole. W tym celu zmieniana jest definicja metody klasy bazowej i jednocześnie wykorzystywana jest definicja pierwotna. Oto funkcja `writeToFile()` zapisująca rekordy studentów w pliku danych o stałej długości rekordu:

```
void Student::writeToFile(fstream& out) const {
    Personal::writeToFile(out);
    out.write(major,majorLen);
}
```

Funkcja ta do zainicjalizowania pierwszych pięciu pól, `SSN`, `name`, `city`, `year` i `salary` wykorzystuje funkcję `writeToFile()` z klasy bazowej. Konieczne jest zastosowanie operatora zakresu `::`, aby wskazać, o której klasy funkcję chodzi. Jednak klasa `Student` dziedziczy bez żadnych zmian funkcję `readKey()` oraz przeciążony operator `==`, gdyż obiekty klas `Personal` i `Student` wykorzystują taki sam klucz do identyfikowania rekordu — `SSN`.

1.11. Ćwiczenia

(1) Jeśli `i` jest liczbą całkowitą, zaś `p` i `q` to wskaźniki liczb całkowitych, które z poniższych przypisań spowodują błąd kompilacji?

(a) `p = &i;`
(b) `p = *&i;`
(c) `p = &*i;`
(d) `i = *&*p;`
(e) `i = *&p;`
(f) `i = &*p;`
(g) `p = &*&i;`
(h) `q = *&*p;`
(i) `q = **&p;`
(j) `q = *&p;`
(k) `q = &*p;`

(2) Wskazać błędy zakładając, że w (b) i (c) `s2` zadeklarowano jako łańcuch i łańcuch tej zmiennej przypisano:

(a)
```
char* f(char *s) {
    char ch = 'A';
    return &ch;
}
```
(b)
```
char *s1;
strcpy(s1,s2);
```
(c)
```
char *s1;
s1 = new char[strlen(s2)];
strcpy(s1,s2);
```

(3) Jeśli dana jest deklaracja:

```
int intArray[] = {1, 2, 3}, *p = intArray;
```

jaka będzie zawartość `intArray` i `p` po wykonaniu instrukcji:

(a) `*p++;`
(b) `(*p)++;`
(c) `*p++; (*p)++;`

(4) Korzystając jedynie ze wskaźników (bez indeksowania tablic), zapisać:

(a) Funkcję dodającą wszystkie liczby z tablicy.
(b) Funkcję usuwającą wszystkie liczby nieparzyste z tablicy uporządkowanej; tablica powinna pozostać uporządkowana. Czy łatwiej byłoby napisać taką funkcję w przypadku tablicy nieuporządkowanej?

(5) Korzystając jedynie ze wskaźników, zaimplementować następujące funkcje operujące na łańcuchach:
   (a) `strlen()`
   (b) `strcmp()`
   (c) `strcat()`
   (d) `strchr()`

(6) Na czym polega różnica między `if (p == q) { ... }` a `if (*p == *q) { ... }`?

(7) We wczesnych wersjach C++ nie były obsługiwane szablony, ale klasy ogólne można było realizować, stosując makrodefinicje z parametrami. W czym szablony są lepsze od takich makrodefinicji?

(8) Jakie jest znaczenie sekcji `private`, `protected` i `public` w klasie?

(9) Jakiego typu konstruktory i destruktory powinny być zaimplementowane w klasie?

(10) Zakładając, że dana jest następująca deklaracja klasy:

```
template<class genType>
class genClass {
   ...
   char aFunction(...);
   ... };
```

określić, co jest nieprawidłowego w poniższej definicji funkcji?

```
char genClass::aFunction(...) { ... };
```

(11) Przeciążanie to doskonały mechanizm języka C++, choć nie zawsze. Jakie operacje nie powinny być przeciążane?

(12) Jeśli klasa `classA` zawiera zmienną prywatną `n`, zmienną chronioną `m` oaz zmienną publiczną `k`, zaś klasa `classB` dziedziczy po `classA`, które z powyższych zmiennych będą dostępne w `classB`? Czy `n` może stać się zmienną prywatną klasy `classB`? Zmienną chronioną? Co ze zmiennymi `m` i `k`? Czy ma znaczenie, czy dziedziczenie klasy `classB` było prywatne, chronione lub publiczne?

(13) Przekształcić deklarację:

```
template<class genType, int size = 50>
class genClass {
    genType storage[size];
    ..................
    void memberFun() {
        ............
        if (someVar < size) { ....... }
        ................
    }
};
```

w której wykorzystano zmienną liczbową `size` jako parametr szablonu w deklarację klasy `genClass`, która nie będzie używała parametru `size`, a mimo to zapewni elastyczność związaną ze stosowaniem tego parametru. Czy w przypadku definiowania konstruktora klasy `genClass` jego nowa definicja będzie w czymś lepsza od starej?

(14) Jaka jest różnica między metodą wirtualną a niewirtualną?

(15) Co się stanie, jeśli za deklaracją klasy `genClass`:

```
class genClass {
    ..................
    virtual void process1(char);
    virtual void process2(char);
};
znajdzie się definicja klasy derivedClass?
class derivedClass : public genClass {
    ..................
    void process1(int);
    int process2(char);
};
```

Które metody są wywoływane, jeśli za deklaracjami dwóch wskaźników:

```
genClass *objectPtr1 = &derivedClass;
         *objectPtr2 = &derivedClass;
```

znajdują się następujące instrukcje?

```
objectPtr1->process1(1000);
objectPtr2->process2('A');
```

1.12. Zadania programistyczne

(1) Napisać klasę `Fraction`, w której przez przeciążanie standardowych operatorów zdefiniowane będą dodawanie, odejmowanie, mnożenie i dzielenie. Napisać metodę pozwalającą redukować ułamki oraz przeciążyć operatory wejścia i wyjścia, aby możliwe było wprowadzanie i wypisywanie ułamków.

(2) Napisać klasę `Quaternion`, w której zdefiniowane zostaną cztery podstawowe operacje dla kwaternionów oraz operacje wejścia i wyjścia. Kwaterniony, zgodnie z definicją Williama Hamiltona z 1843 roku, opublikowaną w 1853 roku w dziele „Lectures on Quaternions", stanowią rozszerzenie liczb zespolonych. Kwaterniony to czwórki liczb rzeczywistych $(a,b,c,d) = a + bi + cj + dk$, gdzie $1 = (1,0,0,0)$, $j = (0,0,1,0)$ oraz $k = (0,0,0,1)$, a przy tym zachodzić muszą następujące równości:

$$i^2 = j^2 = k^2 = -1$$
$$\text{ij} = k, jk = i, ki = j, ji = -k, kj = -i, ik = -j$$
$$(a + bi + cj + dk) + (p + qi + rj + sk)$$
$$= (a + p) + (b + q)i + (c + r)j + (d + s)k$$

$$(a + bi + cj + dk) \cdot (p + qi + rj + sk)$$
$$= (ap - bq - cr - ds) + (aq + bp + cs - dr)i$$
$$+ (ar + cp + dq - bs)j + (as + dp + br - cq)k.$$

Korzystając z podanych równań, zaimplementować klasę kwaternionów.

(3) Napisać program odtwarzający tekst na podstawie zgodności słów. Problem taki znalazł praktyczne zastosowanie podczas odtwarzania niepublikowanych Zwojów znad Morza Martwego. Oto przykładowy wiersz Williama Wordsworth'a, *Nature and the Poet*, oraz zapis zgodności słów z tego wiersza.

So pure the sky, so quiet was the air!
So like, so very like, was day to day!
Whene'er I look'd, thy image still was there;
It trembled, but it never pass'd away.
Zgodność dla 33 słów wygląda następująco:
1:1 so quiet was the *air!
1:4 but it never pass'd *away.
1:4 It trembled, *but it never
1:2 was *day to day!
1:2 was day to *day!
1:3 thy *image still was there;
.....
1:2 so very like, *was day
1:3 thy image still *was there;
1:3 *Whene'er I look'd,

W pokazanej zgodności każde słowo występuje w kontekście maksymalnie do pięciu słów, zaś słowo, o które w danym wierszu chodzi, poprzedzone jest gwiazdką. W przypadku większych zgodności konieczne jest podawanie dwóch liczb — jednej odpowiadającej wierszowi, drugiej oznaczającej wiersz, w którym występuje słowo. Zakładając na przykład, że 1 to wiersz *Nature and the Poet*, wers „1:4 but it never pass'd *away." oznacza, że chodzi o słowo „away" znajdujące się w tym wierszu w czwartym wersie. Warto zauważyć, że do kontekstu należą także znaki przestankowe.

Napisać program ładujący zgodności z pliku i tworzący wektor, w którym każda komórka jest skojarzona z jednym wierszem zgodności. Następnie, korzystając z przeszukiwania binarnego, odtworzyć tekst.

(4) Zmodyfikować program z przykładu zastosowania tak, aby zachowana była kolejność rekordów przy wstawianiu nowych rekordów do pliku. Wymaga to przeciążenia w klasach `Personal` i `Student` operatora <, a następnie użycia go w zmodyfikowanej wersji funkcji `add()` klasy `Database`. Funkcja znajduje miejsce na rekord `d`, przepisuje wszystkie dalsze rekordy, aby zrobić miejsce na `d` i zapisuje `d` w pliku. Teraz można zmodyfikować też metody `find()` i `modify()`. Przykładowo, `find()` może przerwać przeszukiwanie po natknięciu się na rekord większy od szukanego lub przy dojściu do końca pliku). Lepszym rozwiązaniem byłoby użycie przeszukiwania binarnego opisanego w podrozdziale 2.7.

(5) Napisać program zajmujący się pośrednio kolejnością danych w pliku. Użyć wektora wskaźników położenia w pliku (uzyskiwanych przez `tellg()` i `tellp()`), zachowywać kolejność danych w wektorze bez zmieniania kolejności rekordów w pliku.

(6) Zmodyfikować program z przykładu zastosowania tak, aby usuwać rekordy z pliku danych. Zdefiniować w klasach `Personal` i `Student` funkcję `isNull()`, która sprawdzi, czy rekord jest pusty. Zdefiniować w obu klasach funkcję `writeNullToFile()`, która nadpisze usuwany rekord rekordem pustym. Rekord pusty można zdefiniować jako mający coś innego niż liczbę (krzyżyk) w pierwszym znaku pola `SSN`). Zdefiniować następnie w klasie `Database` funkcję `remove()` (bardzo podobną do `modify()`), która znajdzie usuwany rekord i nadpisze go rekordem pustym. Po zakończeniu działania powinien być wywołany destruktor klasy `Database`, który kopiuje niepuste rekordy do nowego pliku danych, usuwa stary plik i zmienia starą nazwę pliku na nową.

Bibliografia

Breymann Ulrich, *Designing Components with the C++ STL*, Harlow: Addison-Wesley, 2000.

Budd Timothy, *Data Structures in C++ Using the Standard Template Library*, Reading, MA: Addison-Wesley, 1998.

Cardelli Luca, i Wegner Peter, „On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys* 17 (1985), 471-522.

Deitel Harvey M., Deitel P.J., *C++: How to Program*, Upper Saddle River, NJ: Prentice Hall, 2003.

Ege Raimund K., *Programming in an Object-Oriented Environment*, San Diego: Academic Press, 1992.

Flaming Bryan, *Practical Data Structures in C++*, New York: Wiley, 1993.

Johnsonbaugh Richard, Kalin Martin, *Object-Oriented Programming in C++*, Upper Saddle River, NJ: Prentice Hall, 1999.

Khoshafian Setrag, Razmik Abnous, *Object Orientation: Concepts, Languages, Databases, User Interfaces*, New York: Wiley, 1995.

Lippman Stanley B., Lajoie Josée, *C++ Primer*, Reading, MA: Addison-Wesley, 1998.

Meyer Bertrand, *Object-Oriented Software Construction*, Upper Saddle River, NJ: Prentice Hall, 1997.

Stroustrup Bjarne, *The C++ Programming Language*, Reading, MA: Addison-Wesley, 1997.

Wang Paul S., *C++ with Object-Oriented Programming*, Boston: PWS, 1994.