Dodatek A

Poznaj C++ w pięć minut!

Dodatek ten w swoim założeniu stanowi pomost dla programistów pascalowych, którzy chcą szybko i bezboleśnie poznać podstawowe elementy języka C++, tak by lektura książki nie napotykała bariery niezrozumienia na poziomie użytej składni. Materiał ten celowo został umieszczony w dodatku, bowiem nie wchodzi on w zasadniczy nurt książki.

Rozdział ten nie zastąpi z pewnością monograficznego podręcznika poświęconego językowi C++, nie to jest jednak jego celem. Poniższy szybki kurs języka C++ obejmuje tylko te elementy, które są konieczne do zrozumienia prezentowanych listingów. Jest to niezbędne minimum, zorientowane wyłącznie na składnię. Jeśli poważnie interesujesz się programowaniem w C++, zainwestuj w dobry podręcznik, np. **[Eck02]** lub **[Str04].**

Elementy języka C++ na przykładach

Kolejne sekcje zawierają serię przykładów, na podstawie których osoba znająca język Pascal może na zasadzie analogii poznać podstawowe zasady zapisu algorytmów w C++.

Pierwszy program

Spójrzmy na poniższy program z doskonale znanego gatunku `hello world`:

```
program pr1; {komentarz}
begin
writeln('Witaj!')

end.
```

```
#include <iostream>
using namespace std;
int main() //komentarz
{
cout << "Witaj!\n";
}
```

- Blok w C++ jest ograniczany przez nawiasy klamrowe { }.
- Działanie programu rozpoczyna się od *funkcji* o nazwie `main`. W C++ funkcja ta zwraca zawsze wartości typu `int`, stąd bierze się konieczność poprzedzenia jej nazwy tym typem. W starszym kodzie można było spotkać czasami typ `void`, czyli informację o funkcji niezwracającą wartości w sensie arytmetycznym.
- Tekst w C++ można wypisać, wysyłając do standardowego wyjścia (`cout`) ciąg znaków ograniczony przez cudzysłów ("tekst"). Sekwencja `\x` oznacza znak specjalny, np. `\n` to skok do nowej linii podczas wypisywania tekstu na ekranie, `\t` — znak tabulacji, etc.
- W C++ komentarz // obowiązuje do końca wiersza. Chcąc coś napisać w komentarzu *pomiędzy* instrukcjami, użyj raczej `/* komentarz */` niż `// komentarz`.

Dyrektywa #include

Symbol # zawarty w kodzie C++ oznacza dyrektywy, czyli polecenia dla kompilatora. Język C++ pozwala na zaawansowane manipulowanie treścią kodu źródłowego, ale dla większości programistów wystarczająca jest na początek wiedza o konieczności użycia dyrektywy `#include <isotream>`, która oznacza dołączenie, jeszcze przed właściwą kompilacją, całej zawartości pliku *iostream* do pliku z programem. Plik ten jest wymagany, gdy zamierzamy używać strumieni `cout`, `cin`, `cerr`, które odpowiadają standardowemu wyjściu (np. ekran), wejściu (np. klawiatura) oraz miejscu, do którego należy wysyłać komunikaty o błędach. To ostatnie zazwyczaj odpowiada ekranowi.

Dyrektywa towarzysząca: `using namespace std` oznacza „odsłonięcie" deklaracji i nazw zawartych w standardowej bibliotece C++. Wiedza o tym, czym są „przestrzenie nazw", przydatna jest w nieco dalszym etapie programowania w C++, wykraczającym poza proste algorytmy zawarte w tej książce, teraz po prostu przyjmij do wiadomości, że programy nie będą działały, gdy tego nie zrobisz.


Ostrzeżenie

> W dalszych przykładach obie te dyrektywy będą dla oszczędności miejsca pomijane, ale nie zapomnij o nich w prawdziwych programach. Warto też wiedzieć, że w starszych programach w C++ stosowana była tylko instrukcja `include <iostream.h>`, obecnie kompilatory już nie akceptują tej formy.

Pliki dołączane zawierają zazwyczaj deklarację często używanych stałych i typów. Plik dołączany opisany w nawiasach <> jest plikiem standardowym z biblioteki kompilatora. Pliki, których nazwa jest otoczona cudzysłowem "", są plikami stworzonymi przez programistę, zazwyczaj są one umieszczane w tym samym katalogu, co pliki z kodem C++.

Podprogramy

W języku C++, podobnie zresztą jak i w klasycznym C, wszystkie podprogramy są nazywane *funkcjami*. Odpowiednikiem znanej z Pascala *procedury* jest funkcja, która niczego nie zwraca — używamy słowa `void`.

Procedury

Oto przykład definicji i użycia procedur w Pascalu i C++:

```
program pr9;
procedure proc1(a,b:integer;
      var m:integer
      )

{zmienna lokalna:}
var c:integer;
begin
   c:=a+b;
   writeln(c);
   m:=c*a*b
end;

var i,j,k:integer;
begin
   i:=10;
   j:=20;
   proc1(i,j,k)
end.
```

```
void proc1(int a,
           int b,
           int& m.
           )
{
//zmienna lokalna:
int c;
c=a+b;
cout << c << endl;
m=c*a*b;
}

int i,j,k;
int main()
{
i=10;
j=20;
proc1(i,j,k);
}
```

- C++ nie umożliwia tworzenia procedur i funkcji lokalnych.
- Zdefiniowane funkcje i procedury są ogólnie dostępne w całym programie.
- Odpowiednikiem deklaracji typu `var` w nagłówku funkcji jest w C++ zasadniczo tzw. referencja (&), np. `Fun( var i:integer;...)` jest równoważne funkcjonalnie formie `Fun(int& i....)`.
- W C++ tablice są z założenia przekazywane przez adres. Przykładowo zapis `Fun(int tab[3])` oznacza chęć użycia jako parametru wejściowego tablicy elementów typu `int`. Podczas wywołania funkcji `Fun` tablica użyta jako parametr jest przekazywana poprzez swój adres i jej zawartość może być fizycznie zmodyfikowana (patrz też strona 300).

Funkcje

Zasadnicze różnice pomiędzy funkcjami w C++ i w Pascalu dotyczą sposobu zwracania wartości:

```
program pr10;
function   plus2(a:integer):integer;
begin
   plus2:=a+2
end;
var i:integer;
begin
   i:=10;
   writeln(plus2(i))
end.
```

```
int plus2(int a)
{
return a+2;
}

int i;
int main()
{
i=10;
cout<<plus2(i)<<endl;
}
```

- W C++ instrukcja `return(v)` powoduje natychmiastowy powrót z funkcji z wartością `v`. Przykładowo: po instrukcji `if(v) return(val)` nie trzeba używać `else`[1] — w przypadku prawdziwości warunku `v` ewentualna dalsza część procedury już nie zostanie wykonana.
- Dobrym zwyczajem programisty jest używanie tzw. *nagłówków funkcji*, czyli informowanie kompilatora o swoich intencjach, o typach parametrów wejściowych i wyjściowych. Nagłówek funkcji jest tym wszystkim, co zostaje z funkcji po usunięciu z niej jej definicji i *nazw* parametrów wejściowych. Przykładowo: jeśli gdzieś w programie jest zdefiniowana funkcja:

  ```
  void f(int k, char* s[3]){...tutaj kod...}.
  ```

  to tuż za dyrektywami `#include` możemy dopisać linię:

  ```
  void f(int, char*[]); // tu średnik!
  ```

Deklaracje nagłówkowe często grupuje się w plikach *.h* dołączanych dyrektywą `#include` jest to zwłaszcza praktykowane w przypadku definiowania tzw. klas (patrz strona 305). Nagłówki są bardzo ważne, gdyż pozwalają już na etapie wstępnych kompilacji uniknąć wielu błędów związanych z wywołaniem funkcji ze złymi parametrami. Notabene, niektóre kompilatory z założenia nie tolerują ich braku, nie pozwalając na skomplikowanie kodu z nieznaną sobie funkcją, czyli taką, której definicji lub deklaracji nagłówka nie odnalazły w pierwszym „przebiegu" procesu kompilacji.

Oto przykład błędnego (bez użycia nagłówka funkcji) i poprawnego kodu C++:

```
// kod błędny:
void fun() //definicja
{...}      //
int main()
{
fun();
```

```
// kod poprawny:
void fun2(int a); //nagłówek
void fun()
{...}
int main()
{
```

[1] Ale nie jest to, oczywiście, zabronione.

```
fun2(5); //kompilator nie zna 'fun2'!
}
void fun2(int a) // definicja
{...}
```

```
fun();
fun2(5);
}
void fun2(int a) // definicja
{...}
```

Operacje arytmetyczne

Niewielkie różnice dotyczą pewnych operatorów, które w Pascalu nazywają się nieco inaczej niż w C++. To co może nas uderzyć przy pierwszym spojrzeniu na program napisany w C++, to duża liczba skrótów w zapisie operacji arytmetycznych. Wywołują one wrażenie małej czytelności, jednak po zaznajomieniu się ze składnią języka ten efekt szybko mija. Mam tu przede wszystkim na myśli operatory ++, -- oraz całą rodzinę wyrażeń typu:

```
zmienna OPERATOR = wyrażenie;
```

Należy podkreślić, iż stosowanie tych form nie jest obowiązkowe, a mimo to wskazane — kod wynikowy programu będzie dzięki temu nieco efektywniejszy.

```
const pi=3.14;
program pr2;
var a,b,c:integer;{globalne}
begin
   a:=1;
   b:=1;
   a:=a+1; {inkrementacja}
   b:=b-2
end.
```

```
const float pi=3.14;
// lub double dla większej precyzji
int a,b,c;
int main()
{
a=1;
b=1;
a++; //inkrementacja
b-=2; // ŚREDNIK!
}
```

- Miejsce deklarowania zmiennych w C++ jest dowolne. Można to uczynić *przed*, *za* i *w ciele* niektórych instrukcji.
- Przypisanie wartości zmiennej odbywa się za pomocą =, a nie := .
- Znanym z Pascala `div` i `mod` odpowiadają w C++ odpowiednio / i %.

Zwróćmy uwagę na często używane w C++ operatory zwiększania o 1 i zmniejszania o 1 (++, --). Zastosowane w wyrażeniu są uwzględniane w pierwszej kolejności, jeśli są użyte *przedrostkowo*, natomiast w przypadku użycia *przyrostkowego* priorytet ma wyrażenie.

Przykład:

```
int a = 2;
int b = 5;
int n = a + b++;          // n = 7 (priorytet ma dodawanie)
cout << n << endl;        // wypisuje 7
cout << b << endl;        // wypisuje 6
cout << a + ++b << endl; // wypisuje 9 (priorytet ma inkrementacja)
```

- Zapis `zmienna` *op* `= wyrażenie` jest równoważne klasycznemu zapisowi:

  `zmienna = zmienna` *op* `wyrażenie`,

  gdzie *op* oznacza pewien operator dwuargumentowy.

Operacje logiczne

Podobnie jak arytmetyczne, operacje logiczne także mają swoje osobliwości. Na szczęście nie jest ich aż tak wiele. Programiści pascalowi powinni zwrócić szczególną uwagę na różnicę pomiędzy = w Pascalu, a == w C++. Niestety kompilator nie wykaże błędu, jeśli w C++ spróbujemy skompilować instrukcję: `if (a=1) a=a-3` zamiast `if (a==1) a=a-3`.

Przykład poprawnych instrukcji logicznych:

```
Program pr3;
var a:boolean;
begin
   a:=true;
   if a=true then
writeln('true')
   else
writeln('false')
   end.
```

```
int main()
{
bool a;
a=(2>3);
if (a==true)
   cout << "Prawda!\n";
else
   cout << "Fałsz!\n";
}
```

W C++ słowa: `bool`, `true` i `false` są słowami kluczowymi języka.

Zwróć uwagę na rolę średnika w C++, który oznacza *koniec* danej instrukcji. Z tego powodu nawet instrukcja znajdująca się przed `else` musi być nim zakończona!

Niektóre operatory logiczne używane w porównaniach są odmienne w obu językach. Przedstawia je tabela A.1.

Tabela A.1. *Porównanie operatorów Pascalu i C++*

Pascal	C++
=	==
not	!
<>	!=
OR	||
AND	&&

Wskaźniki i zmienne dynamiczne

C++ umożliwia stosowanie różnych paradygmatów programowania o wysokim poziomie abstrakcji (jest to przecież tzw. język strukturalny) z możliwościami zbliżającymi go do języka asemblera. Umiejętne wykorzystanie zarówno jednych, jak i drugich umożliwia łatwe programowanie efektywnych aplikacji. Zmienne dynamiczne, adresy i wskaźniki są kluczem do dobrego poznania C++ i trzeba je dobrze opanować. Poniższy przykład ukazuje sposób tworzenia zmiennych dynamicznych i operowania nimi.

```
program pr4;
type example=^real;
var p:example;
begin
   new(p);
   p^:=3.13;
   dispose(p)
end.
```

```
int main()
{
float *p, q=3.14;
p=new float;
*p=q;
delete p;
}
```

- W C++ operacje wskaźnikowe (na adresach) nie są ograniczone do zmiennych dynamicznych.
- Jeśli chcemy wyłuskać ze zmiennej wskaźnikowej *wartość*, na którą wskazuje, poprzedźmy ją symbolem gwiazdki (przykładowo `p` zawiera adres `*p` — wartość, oczywiście pod warunkiem że zmienna `p` została wcześniej zainicjowana).
- Operator & wyłuskuje ze zmiennej jej adres i stanowi, jak widać, przeciwieństwo operatora * (np. `int *p=&m` spowoduje wskazywanie `p` na komórkę pamięci zawierającą zmienną `m`).
- Adres dowolnej zmiennej w C++ może być z niej pobrany poprzez poprzedzenie jej nazwy operatorem &.

Przykład:

```
int k=12;
int *wsk=&k;
cout << *wsk << endl; // program wypisze 12
```

- W C++ nie ma pozaskładniowych ograniczeń co do operacji na adresach, zmiennych wskaźnikowych, dynamicznych przydziałach pamięci, etc.

Referencje

Osobom słabiej znającym C++ polecam zapoznanie się z pojęciem *referencji*, mechanizmu nieużywanego w tej książce, jednak często spotykanego w praktyce do przekazywania obiektów przez adres, np. do funkcji.

Cechą tego mechanizmu jest przekazywanie do funkcji *adresu* zmiennej, a nie jej kopii lokalnej — wszelkie operacje wykonywane na przekazanej zmiennej będą modyfikowały jej oryginał, a nie kopię lokalną.

Popatrzmy na przykład wywołania funkcji przez referencję:

```
void fun(int& x)
{
x=5;
}

int main()
{
int m=6;
fun(m);
cout << m << endl;
}
```

Ten nieco „wydumany" przykład pokazuje, jak naturalnie używa się zmiennych przekazywanych przez referencję — jedyna różnica składniowa występuje w nagłówku funkcji (znak &). Zmienna `m` zostanie zmodyfikowana w wyniku wywołania w funkcji `fun` i program wyświetli 5. Referencje są o tyle wygodne, że nie zmuszają do używania operatora wyłuskiwania (*) wywodzącego się z języka C.

Typy złożone

W języku C++ występuje komplet typów prostych i złożonych, dobrze znanych z języków strukturalnych. Należą do nich między innymi tablice i rekordy. W porównaniu z Pascalem C++ oferuje tu pozornie mniejsze możliwości. Podstawowe ograniczenie tablic dotyczy zakresu indeksów: zawsze zaczynają się one od zera. Nie jest możliwe również deklarowanie rekordów „z wariantami". Te niedogodności są oczywiście do obejścia, ale nie w sposób bezpośredni.

Tablice

Indeksy w tablicach deklarowanych w C++ startują zawsze od zera. Tak więc deklaracja tablicy `t` o rozmiarze 4 oznacza w istocie *4* zmienne: `t[0]`, `t[1]`, `t[2]` i `t[3]`. Aby uzyskać zgodność indeksów w programach napisanych w Pascalu i w C++, konieczne jest zastosowanie właściwej translacji tychże!

```
Program pr5;
type tab=array[3..5] of integer;
var t:tab;
```

```
typedef int tab[3];
tab t;
int main()
```

```
begin
   t[3]:=11;
   t[4]:=t[3]+1;
end.
```

```
{
t[0]=11;
*(t+1)=t[0]+1;

}
```

- Język C++ w zasadzie nie zapewnia kontroli przekroczenia granic tablic podczas dostępu do nich za pomocą indeksowania, ufając niejako programiście. Radą na to jest zastosowanie mechanizmów obiektowych, ale w wersji pierwotnej trzeba po prostu uważać, aby nie znaleźć się w malinach[2].
- Nazwa tablicy w C++ jest jednocześnie wskaźnikiem do niej. Przykładowo: `t` wskazuje na pierwszy element tablicy, a `(t+3)` na czwarty. Notacja `*(t+1)` jest równoważna `t[1]`.
- Deklaracja `int *x` jest równoważna `int x[]`.

Rekordy

Prosty przykład pokazuje elementarne operacje na rekordach:

```
program pr6;
type Komorka=
   record
     znak:char;
     a,b,d:integer;
end;
var x:Komorka;
begin
   x.znak:= 'a';
   x.a:=1
end.
```

```
struct Komorka
{
char znak;
int a,b,d;
};
Komorka x;
int main()
{
x.znak= 'a';
x.a=1;
}
```

- Rekordy w C++ są zwane *strukturami*, dostęp do nich jest podobny, jak w przypadku Pascala (notacja z kropką).
- Nie można wprost zadeklarować rekordu z wariantami.
- Jest możliwe, podobnie jak w Pascalu, włożenie tablicy do rekordu i odwrotnie.
- Pole `nazwa_pola` wskaźnika lub rekordu dynamicznego, wskazywanego przez zmienną `y`, nie jest dostępne poprzez `y.nazwa_pola`, lecz przez konstrukcję: `x->nazwa_pola`. Przykład:

```
Komorka x, *y; // zmienna typu Komorka oraz wskaźnik do niej.
x.znak='a';
y=&x; // pobiera adres zmiennej x
y->a=1; // notacja z kropką byłaby błędna!
```

Instrukcja switch

Instrukcja `switch` w C++ różni się w kilku zdradzieckich szczegółach od swojej odpowiedniczki w Pascalu — proszę zatem uważnie przeanalizować podany przykład!

Najważniejsza do zapamiętania informacja jest związana ze słowem kluczowym `break` (*przerwij*). Ominięcie go spowodowałoby wykonanie instrukcji znajdujących się dalej, aż do napotkania jakiegoś innego `break` lub końca instrukcji `switch`.

[2] Tutaj mogłoby to oznaczać „złe adresy".

```
program pr7;
var w:integer;
begin
   w:=2;
   case w of
     1: writeln('1');
     2: writeln('2');
     otherwise:
        writeln('?');
   end
end.
```

```
int w;
int main()
{
w=2;
switch(w)
{
case 1:cout<< "1\n":break;
case 2:cout<< "2\n":break;
default:
       cout<<"?\n"; break;
}
}
```

- W C++ `break` pełni rolę separatora przypadków.

Iteracje

Instrukcje iteracyjne są podobne w obu językach:

```
program pr8;
var i,j:integer;
begin
   j:=1;
   for i:=1 to 5 do
     begin
     writeln(i*j);
     j:=j+1
     end;
   i:=1;
   j:=10;
   while j>i do
     begin
     i:=i+1;
     writeln(i)
  end
end.
```

```
int i,j;
int main()
{
j=1;
for(i=1;i<=5;i++)
{
cout << i*j << endl;
j++;
}
i=1;
j=10;
while (j>i++)
cout << i <<endl;
```

- `endl` oznacza znak powrotu do nowej linii.
- Niewymieniona tu instrukcja `do{. }while(v)` jest wykonywana w C++ dopóki, dopóty wyrażenie `v` jest różne od `zera`[3].
- Elementy instrukcji `for(e1; e2; e3)` oznaczają odpowiednio:
  - `e1`: inicjację pętli,
  - `e2`: warunek wykonania pętli,
  - `e3`: modyfikator zmiennej sterującej (może nim być funkcja, grupa instrukcji oddzielonych przecinkiem — wtedy są one wykonywane od lewej do prawej).

Przykład:

```
for(int i=6; i<100; Insert(tab[i++]), Pisz(i));
```

(`Pisz` i `Insert` są funkcjami, `tab` zaś pewną tablicą.)

[3] Porównaj np. z `repeat... until`.

Struktury rekurencyjne

Przykład następny pokazuje sposób deklarowania rekurencyjnych (odwołujących się do siebie samych) struktur danych.

```
Program prll;
type wsk=^element;
element=record
    wartosc : integer;
    nastepny:wsk
        end;
 var p:wsk;
 begin
  new(p);
  read(p^.wartosc);
  p^.wartosc=nil
 end.
```

```
typedef struct x
{
int wartosc;
struct x* nastepny;
} ELEMENT;

int main()
{
ELEMENT *p;
p=new ELEMENT;
cin >> p->wartosc;
p->nastepny=NULL;
}
```

♦ Odpowiednikiem `nil` w C++ jest `NULL`.

Parametry programu main()

Czasami istnieje potrzeba podania, przed wykonaniem programu w C++, parametrów wejściowych, np. opcji wywołania, nazw plików z danymi wejściowymi itp. Parametry wywołania są ciągami znaków i można dość łatwo się do nich dostać, choć składnia, pochodząca jeszcze z języka C, jest nieco dziwaczna.

Następujący program wypisuje parametry podane przy wywołaniu programu (skompilowanego kodu wykonywalnego) w linii poleceń:

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
for (int i=0; i< argc; i++)
   cout << "Parametr nr " << i << ":"<< argv[i] << endl;
}
```

Parametr `argc` jest licznikiem parametrów, wartość `argv[0]` jest nazwą samego programu wykonywalnego, pozostałe (jeśli istnieją) są już zwykłymi parametrami wywołania.

Operacje na plikach w C++

Operacje na plikach w C++ w porównaniu z językiem C to przysłowiowa bułka z masłem. Popatrzmy na przykład programu, który odczytuje pewien plik wejściowy, kopiuje go (wiersz po wierszu) do pliku wyjściowego oraz wypisuje, znak po znaku, plik wejściowy w wersji znakowej oraz kodami (dziesiętnymi i szesnastkowo). Nasz program operuje strumieniami, występuje tu bardzo podobna koncepcja do innych, znanych nam strumieni (np. cout).

pliki.cpp

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

```
int main()
{
ifstream plik_WEJ ("input.txt");  // plik wejściowy
ifstream plik_BIN ("input.txt");  // ten sam plik wejściowy
ofstream plik_WYJ ("output.txt"); // plik wyjściowy

string s;

while (getline(plik_WEJ,s))     // kopiujemy, linia po linii, plik
   plik_WYJ << s << endl; // input.txt do output.txt

char c; // wypisujemy znak po znaku plik wejściowy:
while ( plik_BIN.read(&c,1) )
  cout << "Znak: "<< c << ". dec "<< dec << (int)c << ".  hex: "
       << hex << (int)c << endl;
cout << endl;
}
```

Oto przykładowy wynik programu: (plik wejściowy *input.txt* zawierał dwa wiersze z napisami: 123 i abA.

```
Znak: 1. dec:49.  hex: 31
Znak: 2. dec:50.  hex: 32
Znak: 3. dec:51.  hex: 33
Znak:
. dec:10.  hex: a
Znak: a. dec:97.  hex: 61
Znak: b. dec:98.  hex: 62
Znak: A. dec:65.  hex: 41
itp.
```

Programowanie obiektowe w C++

Cała siła i piękno języka C++ zawiera się nie w cechach odziedziczonych od swojego przodka[4], lecz w nowych możliwościach udostępnionych przez wprowadzenie elementów obiektowych. W zasadzie po raz pierwszy w historii informatyki mamy do czynienia z przypadkiem aż tak dużego zainteresowania jakimś językiem programowania, jak to miało miejsce z C++. Niegdysiejsza moda staje się już powoli wymogiem chwili: jest to narzędzie tak efektywne, iż niekorzystanie z niego naraża programistę na stanie w miejscu, w momencie gdy świat coraz szybciej podąża do przodu!

Tylko dla formalności przypomnę jeszcze ostrzeżenie zawarte we wstępie: cały ten rozdział służy wyłącznie nauczeniu programisty pascalowego *czytania* i *rozumienia* listingów napisanych w C++. Ograniczona objętość książki, w konfrontacji z rozpiętością tematyki, nie pozwala na omówienie wszystkiego. Mimo to cytowane tu przykłady zostały wybrane ze względu na ich dużą reprezentatywność. Osoby głębiej zainteresowane programowaniem obiektowym w C++ mogą skorzystać np. z **[Poh89]**, **[Eck02]** w celu poszerzenia swojej wiedzy.

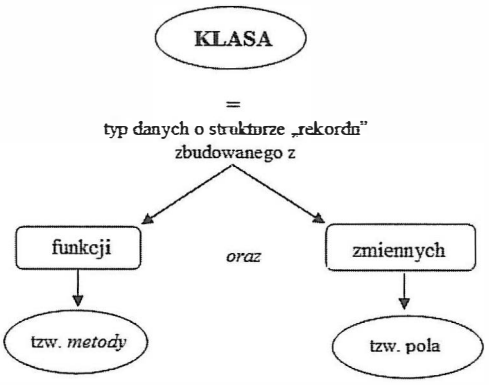
Terminologia

Typowe pojęcia związane z programowaniem obiektowym poglądowo zgrupowano na rysunku A.1.

- Zmienna tego nowego typu danych zwana jest *obiektem*.
- *Metody* są to zwykłe funkcje lub procedury operujące polami, stanowiące jednak własność klasy[5].

[4] Którym jest naturalnie język C.

[5] Tzn. mogą z nich korzystać obiekty danej klasy — inne, zewnętrzne funkcje programu już nie!



Rysunek A.1. *Terminologia w programowaniu obiektowym*

Istnieją dwie metody specjalne:

- *Konstruktor*, który tworzy i inicjalizuje obiekt (np. przydziela niezbędną pamięć, inicjuje w żądany sposób pewne pola, etc.). W deklaracji klasy można bardzo łatwo rozpoznać tę metodę po nazwie — jest ona identyczna z nazwą klasy, ponadto konstruktor ani nie zwraca żadnej wartości, ani nawet nie jest typu `void`.
- *Destruktor*, który niszczy obiekt (zwalnia zajętą przezeń pamięć). Podobnie jak i konstruktor, posiada on specjalną nazwę: identyczną z nazwą klasy, ale poprzedzoną znakiem tyldy (~);

- Każda metoda ma dostęp do pól obiektu, na rzecz którego została aktywowana, poprzez ich nazwy. Inny sposób dostępu jest związany ze wskaźnikiem o nazwie `this` (słowo kluczowe C++): wskazuje on na własny obiekt. Tak więc dostęp do atrybutu `x` może się odbyć albo poprzez `.x`, albo przez `this->x`. Typowo jednak wskaźnik `this` służy w sytuacjach, w których metoda, po uprzednim zmodyfikowaniu obiektu, chce go zwrócić jako wynik (np.: `return *this;`).

Obiekty na przykładzie

Klasa, jako specjalny typ danych, przypomina w swojej konstrukcji rekord, który został wyposażony w możliwość wywoływania funkcji. Definicja klasy może być podzielona na kilka sekcji charakteryzujących się różnym stopniem dostępności dla pozostałych części programu. Najbardziej typowe jest używanie dwóch rodzajów sekcji: *prywatnej* i *publicznej*. W części prywatnej na ogół umieszcza się informacje dotyczące organizacji danych (np. deklaracje typów i zmiennych), a w części publicznej wymienia dozwolone operacje, które można na nich wykonywać. Operacje te mają, oczywiście, postać funkcji, czyli — używając już właściwej terminologii — metod przypisanych klasie.

Spójrzmy na sposób deklaracji klasy, która w sposób dość uproszczony obsługuje tzw. *liczby zespolone*:

Listing

complex.h

```
class Complex
{
public:                          // początek sekcji publicznej
 Complex(double x,double y) // konstruktor klasy
   {
   Re=x;
   Im=y;
   }
```

```
void wypisz();                          // nagłówek funkcji, która wypisuje liczbę urojoną
double Czesc_Rzecz()                    // zwraca część rzeczywistą
{
  return Re;
}
 double Czesc_Uroj ()          // zwraca część urojoną
{
  return Im;
}
// nagłówek funkcji, którą przedefiniowuje operator + (plus), aby
// umożliwić dodawanie liczb zespolonych:
  friend Complex& operator +(Complex,Complex);
// nagłówek funkcji, którą przedefiniowuje operator << aby umożliwić wypisywanie liczb zespolonych:
 friend ostream& operator << (ostream&,Complex);
 private:              // początek sekcji prywatnej
   double Re,Im;       // reprezentacja jako Re+j*Im
 };                    // koniec deklaracji (i częściowej definicji)
                       // klasy Complex
```

Konstrukcja klasy `Complex` informuje o naszych intencjach:

- Wiemy, że liczby zespolone są wewnętrznie widziane jako część rzeczywista i część urojona. Ponieważ sposób budowy klasy jest jej prywatną sprawą, informację o tym umieszczamy w sekcji prywatnej, która redukuje się w naszym przypadku do deklaracji zmiennych `Re` i `Im`.
- Z punktu widzenia obserwatora zewnętrznego (czyli po prostu użytkownika klasy) liczba zespolona jest to obiekt, na którym można wykonywać operację dodawania[6] (mnożenia, dzielenia, etc.) oraz wypisywać ją w pewnej określonej postaci[7].
- W celu dodawania liczb zespolonych przedefiniujemy znaczenie standardowego operatora +, podobnie uczynimy w przypadku wypisywania — tym razem z operatorem <<.

Konstruktor klasy oraz dwie proste metody `Czesc_Rzecz` i `Czesc_Uroj` są zdefiniowane już wewnątrz deklaracji klasy ograniczonej nawiasami klamrowymi { }. Decyzja o miejscu definicji jest najczęściej podyktowana długością kodu: jeśli metoda ma pokaźną objętość[8], to zwykle przemieszcza się ją na zewnątrz, w środku pozostawiając tylko nagłówek.

Deklaracja przykładowego obiektu `20+10*j` ma w programie postać:

- **przypadek 1**: (niejawne tworzenie obiektu poprzez jego deklarację):

```
Complex NazwaObiektu(20, 10);
```

- **przypadek 2** : (jawne tworzenie obiektu poprzez `new`):

```
Complex *NazwaObiektu_Ptr = new Complex(20,10);
```

Wywoływanie metod odbywa się za pomocą standardowej notacji „z kropką":

```
NazwaObiektu.NazwaMetody(parametry); // przypadek 1
```

lub

```
NazwaObiektu_Ptr->NazwaMetody(parametry); //przypadek 2
```

Wiedząc już, jak to wszystko powinno działać, popatrzmy, jak zrealizować brakujące metody.

---

[6] Przykład ogranicza się tylko do dodawania — pozostałe operacje arytmetyczne Czytelnik może z łatwością dopisać samodzielnie.

[7] Reprezentacja za pomocą modułu i fazy zostaje pozostawiona do realizacji Czytelnikowi jako proste ćwiczenie programistyczne.

[8] Powszechnie zalecaną regułą jest nieprzekraczanie jednej strony przy konstrukcji procedury — tak aby całość mogła zostać objęta wzrokiem bez konieczności gorączkowego przerzucania kartek.

Funkcja `wypisz` jest tak trywialna, iż równie dobrze mogłaby być zdefiniowana wprost w ciele klasy. Ponieważ jest to metoda klasy `Complex`, musimy o tym poinformować kompilator przez poprzedzenie jej nazwy nazwą klasy zakończoną operatorem `::` (wymóg składniowy). Jako metoda klasy procedura ta ma dostęp do prywatnych pól obiektu, na rzecz którego została aktywowana. Gdyby jednym z parametrów tej metody był inny obiekt klasy `Complex` (np. `Complex x`), to dostęp do jego pól odbywałby się za pomocą notacji z kropką. Przykład: `x.Re`.


Listing

complex.cpp

```
void Complex::wypisz()
 {
 cout << Re << "+j*" << Im << endl;
 }
```

Język C++ umożliwia łatwe *przedefiniowanie znaczenia operatorów standardowych*, tak aby operacje na obiektach uczynić możliwie najprostszymi. Ponieważ liczby zespolone nieco inaczej dodaje się niż te zwykłe, celowe będzie ukrycie sposobu dodawania w funkcji, a w świecie zewnętrznym pozostawienie do tego celu operatora +. Najwygodniejszym sposobem przedefiniowania operatora dwuargumentowego jest użycie do tego celu tzw. *funkcji zaprzyjaźnionej*: jest to specjalna funkcja, która — nie będąc metodą[9] pewnej określonej klasy — może operować obiektami należącymi do niej. Dotyczy to również dostępu do pól prywatnych!

Nasza funkcja zaprzyjaźniona ma następujące działanie: dwa obiekty `x` i `y` są przekazywane jako parametry. Odczytując wartości ich pól `Re` i `Im`, możliwe jest skonstruowanie nowego obiektu klasy `Complex` wg prostego wzoru: $(a+j\cdot b)+(c+j\cdot d) = (a+c)+(b+d)\cdot j$. Po utworzeniu nowy obiekt jest zwracany przez referencję — czyli jako w pełni adresowalny obiekt. Może on być przypisany innemu obiektowi, na rzecz którego może być aktywowana jakaś metoda klasy `Complex`, etc. Prawidłowe będą zatem instrukcje:

```
Complex x(1.2),y(2.3),c;     // deklaracje obiektów
c=x+y;                       // c = (1+2)+j·(2+3)
```

Popatrzmy na listing funkcji +:

```
Complex& operator +(Complex x,Complex y)
{
double tmp_Re=x.Czesc_Rzecz()+y.Czesc_Rzecz();
double tmp_Im=x.Czesc_Uroj() +y.Czesc_Uroj();
Complex *NowyObiekt=new Complex(tmp_Re,tmp_Im);
return (*NowyObiekt);
}
```

Warto zwrócić uwagę na fakt, iż obiekt `NowyObiekt` jest tworzony w sposób *jawny* za pomocą `new`. Tego typu postępowanie zapewnia nam, że zwrócona referencja będzie się odnosiła do obiektu trwałego (zwykła instrukcja `Complex NowyObiekt` użyta wewnątrz bloku stworzyłaby obiekt tymczasowy, który zniknąłby po wykonaniu instrukcji zawartych we wspomnianym bloku).

Podobnie jak w przypadku operatora + celowe mogłoby być przedefiniowanie operatora <<, który wysyła sformatowane dane do strumienia wyjściowego. W C++ służy do tego celu klasa o nazwie `ostream`. Bez wnikania w szczegóły[10] proponuję zapamiętać zastosowaną poniżej sztuczkę:

```
ostream& operator << (ostream &str,Complex x)
{
str << x.Czesc_Rzecz()<< "+j*" << x.Czesc_Uroj();
return str;
}
```

[9] W konsekwencji nie mogą być wywoływane za pomocą notacji „z kropką"!

[10] Nie miejsce tu bowiem na omawianie dość złożonej hierarchii bibliotek klas dostarczanych z dobrymi kompilatorami C++. Początkującego fana C++ taki opis mógłby dość skutecznie zanudzić.

Spójrzmy wreszcie na program przykładowy, który tworzy obiekty i manipuluje nimi:

```
#include "complex.h"
int main()
{
Complex c1(1.2),c2(3.4);
cout << "c1=";
c1.wypisz();
cout << "c2=";
c2.wypisz();
cout << "c1+c2="<<(c1+c2) << endl;
Complex *c_ptr=new Complex(1.7);
cout << "a)c_ptr wskazuje na obiekt";
c_ptr->wypisz();
cout << "b) c_ptr wskazuje na obiekt"<<*c_ptr<< endl;
}
```

Dla formalności prezentuję rezultaty wykonania programu:

```
c1=1+j*2
c2=3+j*4
c1+c2=4+j*6
c_ptr wskazuje na obiekt 1+j*7
c_ptr wskazuje na obiekt 1+j*7
```

Składowe statyczne klas

Każdy nowo utworzony obiekt posiada pewne unikatowe cechy (wartości swoich atrybutów). Od czasu do czasu zachodzi jednak potrzeba dysponowania czymś w rodzaju zmiennej globalnej w obrębie danej klasy: służą do tego tzw. *pola statyczne.*

Poprzedzenie w definicji klasy `C` atrybutu `x` słowem `static` spowoduje utworzenie właśnie tego typu zmiennej. Inicjacja takiego pola może nastąpić nawet przed utworzeniem jakiegokolwiek obiektu klasy `C`! W tym celu piszemy po prostu

```
C::x=jakaś_wartość;
```

Zbliżone ideowo jest pojęcie *metody statycznej*: może być ona wywołana jeszcze przed utworzeniem jakiegokolwiek obiektu. Oczywistym ograniczeniem metod statycznych jest brak dostępu do pól *niestatycznych danej klasy*, ponadto wskaźnik `this` nie ma żadnego sensu. W przypadku metody statycznej, jeśli chcemy jej umożliwić dostęp do pól niestatycznych pewnego obiektu, trzeba go jej przekazać jako... parametr!

Metody stałe klas

Metoda danej klasy może zostać przez programistę określona mianem *stałej* (np. `void fun() const;`). Nazwa ta jest dość nieszczęśliwie wybrana, chodzi w istocie o metodę deklarującą, że nigdy nie zmodyfikuje pól obiektu, na rzecz którego została aktywowana.

Dziedziczenie własności

Załóżmy, że dysponujemy starannie opracowanymi klasami `A` i `B`. Dostaliśmy je w postaci *skompilowanych bibliotek*, tzn. oprócz kodu wykonywalnego mamy do dyspozycji tylko szczegółowo skomentowane pliki nagłówkowe, które informują nas o sposobach użycia metod i o dostępnych atrybutach.

Niestety twórca klas `A` i `B` dokonał kilku wyborów, które nas niespecjalnie satysfakcjonują, i zaczęło nam się wydawać, że my zrobilibyśmy to nieco lepiej.

Czy musimy wobec tego napisać własne klasy A i B, a dostępne biblioteki wyrzucić na śmietnik? Powinno być oczywiste dla każdego, że nie zadawałbym tego pytania, gdyby odpowiedź nie brzmiała: *NIE*. Język C++ pozwala na bardzo łatwą „ponowną utylizację" kodu już napisanego (a nawet skompilowanego), przy jednoczesnym umożliwieniu wprowadzenia niezbędnych zmian. Weźmy dla przykładu deklaracje dwóch klas A i B, zamieszczone na listingu poniżej:


Listing

dziedzic.h

```
class C1
{
 protected:
  int x;
 public:
  C1(int n) //konstruktor
  {
   x = n;
  }
 void pisz()
 {
  cout<<"**Stara wersja ";
  cout <<"metody `pisz:x=" << x << endl;
 }
};

class C2
 {
private:
 int y;
public:
 C2(int n) //konstruktor
  {
  y = n;
  }

 int ret_y()
  {
   return y;
  }
};
```

Słowo kluczowe protected (*chroniony*) oznacza, że mimo prywatnego dla użytkownika klasy charakteru informacji znajdujących się w tej sekcji zostaną one przekazane ewentualnej klasie pochodnej (zaraz zobaczymy, co to oznacza). Znaczy to, że klasa dziedzicząca będzie ich mogła używać zwyczajnie poprzez nazwę, ale już użytkownik nie będzie miał do nich dostępu poprzez np. notację „z kropką". Jeszcze większymi ograniczeniami charakteryzują się pola *prywatne*: klasa dziedzicząca traci możliwość używania ich w swoich metodach za pomocą nazwy. Ten brak dostępu można, oczywiście, sprytnie ominąć, definiując wyspecjalizowane metody służące do *kontrolowanego* dostępu do pól klasy.

Dołożenie tego typu ochrony danych znakomicie izoluje tzw. *interfejs użytkownika* od bezpośredniego dostępu do danych, ale to już jest temat na osobny rozdział!

Przeanalizujmy wreszcie konkretny przykład programu. Nowa klasa C dziedziczy własności po klasach A i B oraz dokłada nieco swoich własnych elementów:


Listing

dziedzic.cpp

```
#include "dziedzic.h"
class C3:public C1,C2
{
 int z; // pole prywatne
```

```
public:
 C3(int n)  : C1(n+1),C2(n-1)     // nowy
  {                               // konstruktor
  z=2*n;
  }
 pisz_wszystko()
   {
          cout << "Wszystkie pola:\n";
          cout << "\t x="<<x<<endl;
          cout << "\t y="<< ret_y() <<endl;
          cout << "\t z="<<z<<endl;
    }
};
int main()
{
C3 ob(10);
ob.pisz_wszystko();
}

// wynik:
// Wszystkie pola:
// x = 11
// y = 9
// z = 20
```

Konstruktor klasy `C3`, oprócz tego, że inicjalizuje własną zmienną z, wywołuje jeszcze konstruktory klas `C1` i `C2` z takimi parametrami, jakie mu aktualnie odpowiadają. Kolejność wywoływania konstruktorów jest logiczna: najpierw konstruktory klas bazowych (w kolejności narzuconej przez ich pozycję na liście znajdującej się po dwukropku), a na sam koniec konstruktor klasy `C3`. W naszym przypadku parametry `n+1` i `n-1` zostały wzięte „z kapelusza".

Kod zaprezentowany na powyższych listingach jest poglądowo wyjaśniony na rysunku A.2.

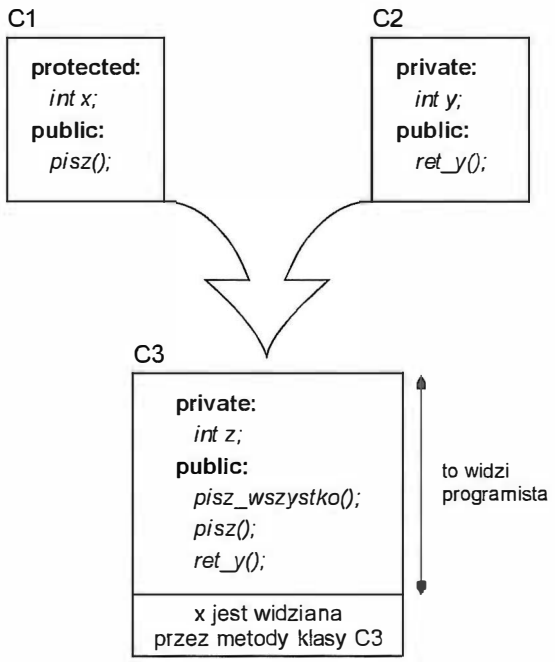
W C++ kilka różnych pod względem zawartości funkcji może nosić taką samą nazwę — taka sytuacja nosi nazwę *przeciążenia*, „ta właściwa" funkcja jest rozróżniana poprzez typy swoich parametrów wejściowych. Przykładowo, jeśli w pliku z programem są zdefiniowane dwie procedury: `void p(char* s)` i `void p(int k)`, to wówczas wywołanie `p(12)` niechybnie będzie dotyczyć tej drugiej wersji.

Mechanizm przeciążania może być zastosowany bardzo skutecznie w powiązaniu z mechanizmami dziedziczenia. Załóżmy, że nie podoba nam się funkcja `pisz`, dostępna w klasie `C3` dzięki temu, że w procesie dziedziczenia przeszła ona z klasy `C1` do `C3`. Z drugiej zaś strony podoba nam się nazwa `pisz` w tym sensie, że chcielibyśmy jej używać na obiektach klasy `C3`, ale do innego celu. Uzupełniamy wówczas klasę `C3` o następującą definicję[11]:

```
void C3::pisz()
{
cout << "nowa wersja metody 'pisz'\n";
}
// wynik użycia ob.C1::pisz(); w main:
// ** stara wersja metody 'pisz': x = 11 **
// wynik użycia ob.pisz(); w main:
// ** nowa wersja metody 'pisz': z = 20 **
```

Teraz instrukcja `ob.pisz()` wywoła nową metodę `pisz` (z klasy `C3`), gdybyśmy zaś koniecznie chcieli użyć starej wersji, to należy jawnie tego zażądać poprzez `ob.C1::pisz()`.

[11] Ponadto należy dodać linię `void pisz();` do sekcji publicznej klasy `C3`.



Rysunek A.2.
Przykład dziedziczenia własności w C++

Nasz przykład zakłada kilka celowych niedomówień. Wynika to z tego, że problematyka dziedziczenia własności w C++ zawiera wiele niuansów, które mogłyby być nużące dla nieprzygotowanego odbiorcy. Mimo to zaprezentowana powyżej wiedza jest już wystarczająca do tworzenia dość skomplikowanych aplikacji zorientowanych obiektowo. Inne mechanizmy, takie jak np. bardzo ważne funkcje *wirtualne* i tzw. *klasy abstrakcyjne*, trzeba już pozostawić wyspecjalizowanym publikacjom — zachęcam Czytelnika do lektury.

Kod warunkowy w C++

W kodach źródłowych (głównie wersja *ftp*), można napotkać na dyrektywy zlecające warunkowe wykonywanie kodu programu.

Jeśli napotkasz fragment kodu o strukturze podobnej do:

```
#define TEST //(*)
// tutaj jakieś deklaracje i instrukcje (1)
#ifdef TEST
// tutaj jakieś inne deklaracje i instrukcje (2)
#endif
```

to kompilator wykona instrukcje oznaczone (1). Gdyby teraz usunąć lub wstawić do komentarza linijkę (*), to wykonałby się kod zawarty w drugiej odnodze (2).

Dodatek B

Systemy obliczeniowe w pigułce

W niniejszym dodatku przedstawię kilka podstawowych informacji dotyczących systemów kodowania — dwójkowego i szesnastkowego i arytmetyki binarnej. Z moich obserwacji wynika bowiem, że nie każdy programista potrafi się przyznać, że ma z tymi pojęciami kłopoty, i ten właśnie suplement pozwoli mu wyrównać tę lukę w wykształceniu.

Kilka definicji

W ramach wstępu wprowadzę pojęcie *systemu pozycyjnego*, w którym liczby zapisuje się cyframi $c_1, c_2, \ldots, c_n$ z pewnego niewielkiego zbioru, a zapis taki interpretuje się jako sumę iloczynów liczb reprezentowanych przez poszczególne cyfry i potęg liczby naturalnej n, nazywanej podstawą systemu, o wykładnikach równych numerowi pozycji cyfry w ciągu. W branży informatycznej popularne są szczególnie podstawy systemu: *2*, *8* i *16* — czyli systemy: dwójkowy, ósemkowy i szesnastkowy.

Powszechnie używane układy cyfrowe, z których składają się komputery, wykonują operacje arytmetyczne i są sterowane z wykorzystaniem zasad tzw. algebry Boole'a, czyli zbioru działań zakładających operowanie tylko dwoma elementami oznaczonymi 0 i 1. Zasady tej algebry można podsumować zbiorem praw przedstawionych w tabeli B.1.

Operacje logiczne występujące w tabeli to: suma (+), iloczyn (·) i dopełnienie (kreska nad elementem).

System dwójkowy

System dwójkowy pozwala na zapis liczb za pomocą dwóch umownych znaków: `0` (zero) i `1` (jedynka). Umowność polega na niewnikaniu w szczegóły, co dla danego komputera (lub sprzętu, np. układu scalonego) oznacza fizycznie każdy z tych stanów — czy jest to zero woltów, czy np. 5 woltów. Z naszego punktu widzenia liczy się tylko fakt, iż ciągle dla wielu zastosowań reprezentacja dwójkowa liczb jest najwygodniejsza, a ponieważ jest ona doskonale wspomagana w językach programowania, np. C++, to tym cenniejsze będzie pełne jej zrozumienie.

Tabela B.1. *Prawa i twierdzenia algebry Boole'a*

Prawa przemienności	**Prawa łączności**	**Prawa rozdzielności**
$x + y = y + x$ $x \cdot y = y \cdot x$	$(x + y) + z = x + (y + z)$ $(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$x + (y \cdot z) = (x + y) \cdot (x + z)$ $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
Prawa identyczności	**Prawa dopełnienia**	**Prawa idempotentności**
$x + 0 = x$ $x \cdot 1 = x$	$x + \overline{x} = 1$ $x \cdot \overline{x} = 0$	$x + x = x$ $x \cdot x = x$ $x + 1 = 1$ $x \cdot 0 = 0$
Prawa pochłaniania:	**Prawa de Morgana**	
$(x \cdot y) + x = x$ $(x + y) \cdot x = x$	$\overline{x + y} = \overline{x} \cdot \overline{y}$ $\overline{x \cdot y} = \overline{x} + \overline{y}$	

Popatrzmy, jak w naturalnym dla człowieka systemie dziesiętnym można przedstawić liczbę 1624:

$$1\,624 = 1 \cdot 1\,000 + 6 \cdot 100 + 2 \cdot 10 + 4 = 1 \cdot \mathbf{10}^3 + 6 \cdot \mathbf{10}^2 + 2 \cdot \mathbf{10}^1 + 4 \cdot \mathbf{10}^0$$

Sumujemy „wagi" (1, 6, 2 i 4) mnożone przez potęgi podstawy systemu obliczeniowego (tutaj: 10) i wychodzi nam dowolna liczba, którą chcemy przedstawić. Albo na odwrót: to właśnie dowolna liczba może zostać rozłożona na podaną wyżej reprezentację.

Wadą systemu dziesiętnego jest kłopot w budowie sprzętu elektronicznego, który operowałby liczbami tego systemu. Nie tylko komplikuje się reprezentacja wewnętrzna (aż dziesięć stanów do zakodowania na poziomie technicznym, czyli np. bramek i tranzystorów), ale jeszcze obliczenia w systemie dziesiętnym są dość pamięciożerne.

System dwójkowy używa do reprezentacji liczb dwóch znaków: 0 i 1, a podstawą systemu jest dwójka.

Popatrzmy, jak łatwo przekształca się liczbę dwójkową na jej reprezentację dziesiętną.

$$1101 = 1 \cdot 2^3 + 1 \cdot \mathbf{2}^2 + 0 \cdot \mathbf{2}^1 + \mathbf{1} \cdot 2^0 = 8 + 4 + 0 + 1 = 13$$

W rzeczywistych komputerach mamy do czynienia z wewnętrznymi rejestrami, które pozwalają zapamiętywać i operować na grupach bitów. Najbardziej znaną jednostką, w której podaje się tak długość takiej grupy, jest *bajt*, czyli osiem bitów.

Bajt może reprezentować liczby od 00000000 do 11111111, czyli dziesiętnie od 0 do 255.

W przypadku systemu dwójkowego poważnym problemem okazało się reprezentowanie liczb ujemnych. Pierwszy pomysł, aby najbardziej znaczący bit (ten wysunięty najbardziej na lewo) oznaczał znak, np. 1 to liczba ujemna, a 0 — dodatnia: 0011 = 3, 1011 = -3. Z przyczyn technicznych system ten się nie przyjął, gdyż niejednoznaczność związana ze znakiem zera (1000 = +0, 0000 = -0???) rodziła sporo kłopotów. Matematycy przyszli wówczas z pomocą, wymyślając *system uzupełnienia dwójkowego*, w którym liczba ujemna jest uzyskiwana poprzez zanegowanie wszystkich bitów (0 na 1, 1 na 0) i dodanie liczby jeden do wyniku.

Przykład:

1101	+13
0010	negujemy bity
+0001	dodajemy 1
0011	wynik -13.

Zaletą systemu uzupełnienia jest rozwiązanie problemu znaku zera i brak potrzeby posiadania operacji odejmowania — wystarczy negacja bitów i dodawanie jedynki (inkrementacja).

Operacje arytmetyczne na liczbach dwójkowych

Arytmetyka liczb dwójowych jest podobna do dziesiętnych (stosujesz reguły „słupkowe" poznane w szkole podstawowej!). Popatrz na kilka prostych przykładów zawartych w tabeli B.2.

Tabela B.2. *Arytmetyka liczb dwójkowych na przykładzie*

	Przykład	Komentarz
Dodawanie	A = 11011 = 27_{10} B = 11001 = 25_{10} A+B = *110100*	Pamiętaj O przeniesieniu 1 dla (1+1)!
Odejmowanie	A = 11010 = 26_{10} B = 10001 = 17_{10} A-B = 1001	W przypadku odejmowania (0–1) musimy dokonać „zapożyczenia" 1 na następnej pozycji liczby.
Mnożenie	A = 101 = 5_{10} B = 010 = 2_{10} 000 101 A*B = *1010*	Tak jak w systemie dziesiętnym (mnożysz przez kolejne cyfry i przesuwasz się w lewo).
Dzielenie	110 1100:10 = 12_{10}:6_{10}=2_{10} -10 0100 10 000	Tak jak w systemie dziesiętnym (odejmujesz przesunięty na lewo dzielnik od dzielnej aż do uzyskania 0 lub reszty).

Operacje logiczne na liczbach dwójkowych

Podstawowe operacje logiczne wykonywane na liczbach dwójkowych są przedstawione w tabeli B.3. W programach często używa się kodowania bitowego do oznaczania opcji wywołania funkcji — zazwyczaj niskopoziomowych, np. na plikach.

Tabela B.3. *Podstawowe operacje na liczbach dwójkowych*

	Zasada obliczania wyniku na dwóch bitach	Przykład (zapisany w konwencji C++)
LUB (ang. *OR*) — suma logiczna	Jeśli choć jeden z bitów jest jedynką, to wynik jest równy 1.	`1101 | 0001 = 1101`
I (ang. *AND*) — iloczyn logiczny	Jeśli choć jeden z bitów jest zerem, to wynik jest równy 0.	`1001 & 1100 = 1000`
Różnica symetryczna[1] (ang. *XOR*)	Jeśli oba bity są równe, to wynik jest zerem.	`1100 ^ 1001 = 0101`
Negacja	1 staje się 0, 0 staje się 1	`~1101=0010`
Przesunięcie w lewo o *n* bitów	-	`zmienna << n`
Przesunięcie w prawo o *n* bitów	-	`zmienna >> n`

[1] Inna popularna nazwa to „suma modulo 2".

```
#define OPCJA_1     1       // 0001
#define OPCJA_2     2       // 0010
#define OPCJA_3     4       // 0100
#define OPCJA_4     8       // 1000
...
int opcje = 0;
opcje = OPCJA_1 | OPCJA_4; //„włączenie" opcji 1 i 4, wynik: 1001
```

Aby zaprzyjaźnić Czytelnika z operacjami bitowymi, proponuję analizę następującego programu, który pokazuje kilka typowych operacji bitowych i przy okazji zaznajamia Czytelnika z prostym sposobem wyświetlania liczby w postaci dwójkowej.


Listing

bit_operations.cpp

```
#include <iostream>
using namespace std;
void showbits(unsigned char s)
// funkcja pokazuje reprezentację binarną znaku
{
unsigned char wagi[8]={1,2,4,8,16,32,64,128}; // maska bitu wagi
for(int i=7; i >= 0; i--)
   {
   int bit = (wagi[i] & s);
   if (bit !=0 )
      cout << '1';
   else
      cout << '0';
  }
}
// --Przykładowe operacje na bitach
int main()
{
cout << "i\tbinarnie\tprzes.w lewo\tnegacja\n";
for (int i=0; i<16; i++)
   {
   cout << i << "\t"; showbits(i); cout << "\t"; // dec oraz binarnie
                                                  // przesunięcie o 1 bit w lewo:
   int j= i << 1;
   showbits(j);
   cout << "\t";
   int k= ~i;
   showbits(k);
   cout << endl; // negacja bitowa
   }
}
```

Wynik działania programu:

```
i       binarnie        przes.w lewo    negacja

0       00000000        00000000        11111111
1       00000001        00000010        11111110
2       00000010        00000100        11111101
3       00000011        00000110        11111100
4       00000100        00001000        11111011
5       00000101        00001010        11111010
6       00000110        00001100        11111001
7       00000111        00001110        11111000
8       00001000        00010000        11110111
9       00001001        00010010        11110110
10      00001010        00010100        11110101
11      00001011        00010110        11110100
12      00001100        00011000        11110011
```

13	00001101	00011010	11110010
14	00001110	00011100	11110001
15	00001111	00011110	11110000

System ósemkowy

Liczba ósemkowa jest liczbą zapisaną w pozycyjnym systemie ósemkowym, tj. za pomocą ośmiu cyfr, od 0 do 7, np. 074, 0322. W C++ liczba poprzedzona zerem jest traktowana jak liczba ósemkowa.

Zapis ósemkowy łatwo zamienia się na dwójkowy (i odwrotnie): wystarczy zamieniać cyfry ósemkowe na trzy cyfry dwójkowe lub odwrotnie. Przykład:

$(78)_{10} = (\mathbf{001}001\mathbf{110})_2 = (116)_8$

System szesnastkowy

Liczba szesnastkowa pozwala zapisać w zwięzłej postaci długie liczby binarne. System szesnastkowy do oznaczania cyfr używa czwórek bitów (półbajtów), które zastępują ciągi bitów wg podanej poniżej tabeli B.4.

Tabela B.4. *Tabela konwersji liczb szesnastkowych na dwójkowe i dziesiętne*

Cyfra szesnastkowa	Wartość dziesiętna	Wartość dwójkowa
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

W języku C++ liczby szesnastkowe oznacza się przedrostkiem `0x`.

Przykład konwersji liczby dwójkowej na szesnastkową:

$(78)_{10} = (0100\mathbf{1110})_2 = (4E)_{16}$

Zmienne w pamięci komputera

Wiemy już, że komputer działa i funkcjonuje w systemie dwójkowym. Dla naszej wygody w programach czasami stosuje się zapisy liczbowe z wykorzystaniem systemu ósemkowego lub szesnastkowego, ale wynika to wyłącznie z chęci uproszczenia zapisu liczb i zminimalizowania

ryzyka błędów (liczby dwójowe są długie i niezbyt czytelne dla większości osób). Możemy jednak zadać w tym miejscu pytanie, jak się ma zasada zapisu binarnego do języków programowania wykorzystujących przecież całkiem zrozumiałe, logiczne typy danych, takie jak `integer`, `char`, `float` itp.? Jak komputer przechowuje *liczbę* `15`. a jak *ciąg znaków* „15"?

Tak jak wspomniałem o tym w rozdziale 5., lista typów podstawowych oraz ich precyzja jest w C++ określona przez standard tego języka, ale ich konkretna implementacja, np. liczba bitów zajmowanych przez zmienną określonego typu, zależy już od konkretnej technologii (sprzęt) i systemu operacyjnego. Architektura sprzętu (typ procesora komputera i obsługiwane przezeń operacje na tzw. rejestrach — komórkach pamięci specjalnego przeznaczenia) determinują zasady adresowania danych i kodu. Bezpośrednie programowanie procesora i zasobów sprzętowych umożliwia język niskiego poziomu, tzw. asembler. Biegłe programowanie w asemblerze obecnie jest rzadkością nawet w środowiskach informatyków.

System operacyjny można traktować jako swego rodzaju nakładkę na sprzęt, pozwalającą izolować programy tworzone przez programistów od warstwy fizycznej zasobów komputera. Jedną z cech wymaganych od systemu operacyjnego jest zapewnienie bezpieczeństwa dostępu do zasobów sprzętowych: programy użytkownika są izolowane od programów systemowych, teoretycznie nie powinno być możliwe bezpośrednie adresowanie zasobów sprzętowych, np. wysyłanie komend do karty graficznej lub nagrywarki DVD. Oczywiście w praktyce bywa inaczej, ale warto mieć świadomość, że programy ingerujące w sprzęt są nieprzenośne i ich czas życia bywa dość krótki.

Po tych uwagach nie jest trudno się domyślić, że to właśnie kompilator danego języka programowania, przygotowany pod konkretny sprzęt (np. procesory Intel) i system (np. Windows XP/Vista) dokonuje „przekodowania" umownych, logicznych typów danych, na ich wersje „fizyczne", umiejscowione „gdzieś w pamięci". Programista nie musi się już martwić zasadami składowania zmiennych w pamięci operacyjnej, oczywiście pod warunkiem że nie ma takiej jawnej potrzeby (optymalizacja kodu, programowanie sprzętu).

Kodowanie znaków

Każdy użytkownik komputera na co dzień pracuje, używając klawiatury komputerowej, i w ogóle nie zastanawia się, w jaki sposób naciśnięcie klawisza z symbolem np. litery *A* powoduje wyświetlenie takiej litery na ekranie monitora. Dla nikogo nie powinno być już niespodzianką, że komputer nie zapisuje bezpośrednio znaków w pamięci, tylko używa do tego pewnej umownej reprezentacji dwójkowej. Kodowaniem i dekodowaniem znaków zajmuje się sprzęt, np. klawiatura „wie", że po naciśnięciu znaku A powinna wysłać określony kod (zazwyczaj 65), podobnie ekran (karta graficzna) potrafi „narysować" określony znak na podstawie otrzymanego kodu.

Jednym z najczęściej stosowanych standardów kodowania znaków jest tzw. kod ASCII (ang. *American Standard Code for Information Interchange*). Kod ten jest 7-bitowy, co oznacza, że do dyspozycji mamy liczby z zakresu 0 – 127, którym przypisano litery alfabetu angielskiego, cyfry, znaki interpunkcyjne (tabela B.6) i polecenia sterujące sposobem wyświetlania znaków na drukarce lub terminalu znakowym (np. znak nowej linii, tabulacja, backspace) — (tabela B.5).

Tabela B.5. *Kod ASCII — kody sterujące*

Dec	Hex	Znak	Skrót
0	*00*	Null	NUL
1	*01*	Start Of Heading	SOH
2	*02*	Start of Text	STX
3	*03*	End of Text	ETX

Tabela B.5. *Kod ASCII — kody sterujące — ciąg dalszy*

Dec	Hex	Znak	Skrót
4	*04*	End of Transmission	EOT
5	*05*	Enquiry	ENQ
6	*06*	Acknowledge	ACK
7	*07*	Bell	BEL
8	*08*	Backspace	BS
9	*09*	Horizontal Tab	HT
10	*0A*	Line Feed	LF
11	*0B*	Vertical Tab	VT
12	*0C*	Form Feed	FF
13	*0D*	Carriage Return	CR
14	*0E*	Shift Out	SO
15	*0F*	Shift In	SI
16	*10*	Data Link Escape	DLE
17	*11*	Device Control 1 (XON)	DC1
18	*12*	Device Control 2	DC2
19	*13*	Device Control 3 (XOFF)	DC3
20	*14*	Device Control 4	DC4
21	*15*	Negative Acknowledge	NAK
22	*16*	Synchronous Idle	SYN
23	*17*	End of Transmission Block	ETB
24	*18*	Cancel	CAN
25	*19*	End of Medium	EM
26	*1A*	Substitute	SUB
27	*1B*	Escape	ESC
28	*1C*	File Separator	FS
29	*1D*	Group Separator	GS
30	*1E*	Record Separator	RS
31	*1F*	Unit Separator	US
32	*20*	Spacja	
127	*7F*	Delete	DEL

Tabela B.6. *Kod ASCII — cyfry, litery alfabetu, znaki interpunkcyjne*

Dec	Hex	Znak	Dec	Hex	Znak	Dec	Hex	Znak
33	*21*	!	*65*	*41*	A	*97*	*61*	a
34	*22*	"	*66*	*42*	B	*98*	*62*	b
35	*23*	#	*67*	*43*	C	*99*	*63*	c
36	*24*	$	*68*	*44*	D	*100*	*64*	d
37	*25*	%	*69*	*45*	E	*101*	*65*	e
38	*26*	&	*70*	*46*	F	*102*	*66*	f
39	*27*	'	*71*	*47*	G	*103*	*67*	g
40	*28*	(	*72*	*48*	H	*104*	*68*	h
41	*29*	)	*73*	*49*	I	*105*	*69*	i
42	*2A*	*	*74*	*4A*	J	*106*	*6A*	j
43	*2B*	+	*75*	*4B*	K	*107*	*6B*	k

Tabela B.6. *Kod ASCII — cyfry, litery alfabetu, znaki interpunkcyjne — ciąg dalszy*

Dec	Hex	Znak	Dec	Hex	Znak	Dec	Hex	Znak
44	*2C*	,	*76*	*4C*	L	*108*	*6C*	l
45	*2D*	-	*77*	*4D*	M	*109*	*6D*	m
46	*2E*	.	*78*	*4E*	N	*110*	*6E*	n
47	*2F*	/	*79*	*4F*	O	*111*	*6F*	o
48	*30*	0	*80*	*50*	P	*112*	*70*	p
49	*31*	1	*81*	*51*	Q	*113*	*71*	q
50	*32*	2	*82*	*52*	R	*114*	*72*	r
51	*33*	3	*83*	*53*	S	*115*	*73*	s
52	*34*	4	*84*	*54*	T	*116*	*74*	t
53	*35*	5	*85*	*55*	U	*117*	*75*	u
54	*36*	6	*86*	*56*	V	*118*	*76*	v
55	*37*	7	*87*	*57*	W	*119*	*77*	w
56	*38*	8	*88*	*58*	X	*120*	*78*	x
57	*39*	9	*89*	*59*	Y	*121*	*79*	Y
58	*3A*	:	*90*	*5A*	Z	*122*	*7A*	Z
59	*3B*	;	*91*	*5B*	[	*123*	*7B*	{
60	*3C*	<	*92*	*5C*	\	*124*	*7C*	|
61	*3D*	=	*93*	*5D*	]	*125*	*7D*	}
62	*3E*	>	*94*	*5E*	^	*126*	*7E*	~
63	*3F*	?	*95*	*5F*	_			
64	*40*	@	*96*	*60*	`			

Siedem bitów w kodzie ASCII stanowiło pewną zaszłość historyczną i ponieważ komputery operowały już słowami 8-bitowymi, szybko uzupełniono kod ASCII o kody powyżej 127, gdzie wstawiono znaki semigraficzne, służące do tworzenia obramowań (np. ╔, ╤ itp.) oraz znaki narodowe. Rozszerzona część tablicy ASCII zawiera zestaw zależny od kraju i czasami producenta sprzętu komputerowego (drukarki, terminale znakowe). W celu obsługi polskich znaków stworzono wariant kodu ASCII o nazwie ISO 8859-2 (tzw. ISO Latin-2).

Niestety w międzyczasie powstało kilkadziesiąt innych standardów, o czym łatwo się przekonać, oglądając różne programy i strony internetowe (czasami w miejsce polskich znaków pojawiają się dziwne symbole). Świetne, tabelaryczne zestawienie standardów kodowania polskich znaków znajduje się na stronie internetowej:

http://pl.wikipedia.org/wiki/Sposób_kodowania_polskich_znaków.

Wadą standardu ASCII jest zamieszanie wprowadzone przez producentów oprogramowania i sprzętu (w zasadzie tylko część kodów 0... 127 jest jednoznaczna) i brak obsługi wielu języków na raz (do dyspozycji w wersji rozszerzonej standardu ASCII mamy tylko 8 bitów, co daje 256 możliwości).

Alternatywą dla ASCII staje się powoli 16-bitowy system kodowania Unicode, pozwalający na reprezentowanie do 65 tysięcy znaków w sposób jednoznaczny, nie są potrzebne żadne informacje dodatkowe, pozwalające rozróżniać języki czy wręcz pojedyncze znaki. Jako ciekawostkę można podać, że w pliku zgodnym z Unicode kolejność znaków określa kolejność ich faktycznego czytania, a nie oglądania (pomyśl o np. tekście artykułu gazetowego zawierającego tekst w języku angielskim z wtrąconymi słowami lub zdaniami arabskimi).

Pojemność Unicode wydaje się ogromna, choć znając historię wieży Babel, być może kiedyś i ten kod się zapełni w wyniku twórczej działalności człowieka!

Dodatek C

Kompilowanie programów przykładowych

Zawartość archiwum ZIP na ftp

Wszystkie programy przykładowe i materiały uzupełniające są umieszczone na serwerze ftp wydawnictwa Helion pod adresem *ftp://ftp.helion.pl/przyklady/algo4.zip*. Pobierz i rozpakuj te programy na dysk swojego komputera, aby je modyfikować lub sprawdzać ich działanie.

Struktura plików i folderów dołączonych do książki jest następująca:

- Folder *CPP* zawiera programy do kompilacji (rozszerzenie *.cpp*) i dwa pliki wsadowe, dokonujące hurtowej kompilacji przy pomocy GCC:
  - `kompiluj_gcc.bat` (dla systemu DOS/Windows)
  - `kompiluj_gcc.sh` (dla Linuxa).

  Powyższe pliki wsadowe niczym się nie różnią, ale dla wygody przygotowałem wersje w obu standardach zapisu znaków nowej linii.
- Folder *Visual* zawiera gotowe projekty programów do kompilacji z użyciem kompilatora Microsoft Visual C++ Express Edition. Projekty zawarte w tym folderze używają plików źródłowych zawartych w *CPP*.
- Folder *Dodatki* zawiera materiały pobrane z Internetu, znalazło się tam m.in. dodatkowe oprogramowanie freeware, które powinno zainteresować zwłaszcza Czytelników pragnących eksperymentować z kodowaniem i kompresją danych. Oprócz tego znajdziesz tam listę komend GCC i inne uzupełniające materiały.

Darmowe kompilatory C++

Do kompilacji większości programów przykładowych powinno wystarczyć proste środowisko np. GCC obejmujące kompilatory C, C+ itp. GCC, czyli GNU Compiler Collection, dawniej znany pod nazwa GNU C Compiler, jest to zestaw kompilatorów różnych języków programowania, rozwijany w ramach projektu GNU i udostępniany na licencji GPL. Systemy oparte na Uniksie używają GCC jako podstawowego kompilatora, ale GCC może działać również w systemach Windows lub Mac OS (ten ostatni zresztą korzeniami tkwi w BSD Unix).


Uwaga

W systemach typu Linux/Unix kompilator C++ często jest już wstępnie zainstalowany. Aby to zweryfikować, spróbuj wykonać komendę c++ lub g++ z poziomu terminala tekstowego. Jeśli komenda nie zadziała, wejdź do instalatora Twojej dystrybucji Linuxa (np. YAST w Suse) i dokonaj wyboru pakietu do programowania w C++. W Linuksie Ubuntu, gdy brakuje kompilatora, wystarczy z poziomu konsoli zainstalować go, uruchamiając polecenie `sudo apt-get install g++`.

Aby pobrać GCC, zajrzyj pod adres *http://gcc.gnu.org*, gdzie w sekcji Download/ Binaries możesz wybrać wersję dla swojego systemu (np. Windows, Linux). Jeśli używasz Windows, to polecam instalację MinGW (*http://www.mingw.org*) — nazwa jest skrótem od „Minimalist GNU for Windows". Pakiet instaluje się domyślnie w folderze C:\MinGW. Aby go używać, wystarczy dodać folder C:\MinGW\bin do zmiennej środowiskowej *Path* i od tego momentu można kompilować programy z poziomu konsoli[1].

Jeśli używasz komputera Apple i systemu Mac OS, to polecam rejestrację w serwisie Apple Developer Connection (*http://connect.apple.com*). Można tam np. pobrać całe środowisko graficzne Xcode, w ramach którego dostaniemy komplet kompilatorów, także najnowszy kompilator C++. Xcode jest ciężki w użyciu i generuje dużo nadmiarowych plików, ale możemy poprzestać na używaniu kompilatora C++ z linii poleceń terminala.

Użytkownikom Windows polecam jednak zainstalowanie świetnego, darmowego kompilatora Microsoft Visual C++ Express Edition, który można pobrać pod adresem *http:// www.microsoft.com/express/download*. Na stronie producenta można pobrać go w wersji online (Web Install) lub jako obraz dysku DVD-ROM, np. do wypalenia programem Nero (Offline Install). Plik, który musimy w tym przypadku pobrać, zajmuje ok. 750 MB, zatem bez stałego dostępu do Internetu się nie obejdzie. Jeśli kogoś jednak przeraża rozmiar tej instalacji, to może skorzystać z darmowego kompilatora Dev C++, który choć dość stary (ostatnia stabilna wersja pochodzi z 2005 roku), jest nakładką na kompilator GNU C++ (MinGW) i oferuje polski interfejs użytkownika! Pakiet można pobrać pod adresem: *http://www.bloodshed.net*, sekcja *Download*. Instalacja jest bardzo prosta i sprowadza się do uruchomienia pliku instalacyjnego, a potem wybrania języka polskiego jako domyślnego.

Kompilacja i uruchamianie

Jak skompilować i uruchomić program z tej książki?

Pierwsze, co należy zrobić, to skopiować i rozpakować do dowolnego, roboczego katalogu pliki pobrane z *ftp* (pliki są zawarte w archiwum ZIP). Reszta zależy już od systemu operacyjnego i środowiska, które posiadamy. Ponieważ celem tej książki nie jest kurs użycia kompilatorów, opiszę tylko podstawowe warianty kompilacji i uruchamiania programów, które powinny okazać się wystarczające dla Czytelników używających różnych systemów i narzędzi deweloperskich.

GCC

Opisywany w niniejszym punkcie sposób kompilacji jest niezależny od systemu operacyjnego. Zakładam też, że chcesz pracować, używając tylko komend z linii poleceń (terminal tekstowy), co w dobie graficznych systemów operacyjnych jest dla wielu osób wielkim wyzwaniem!

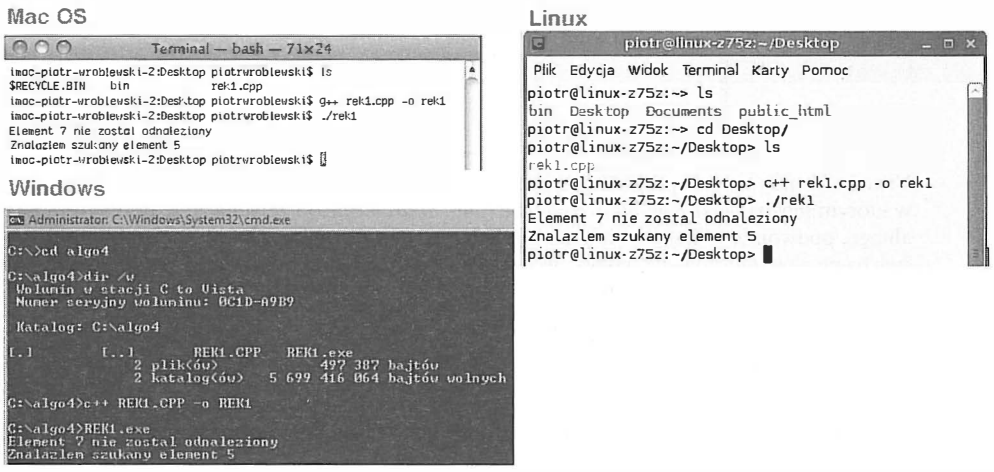
Wykonaj następujące czynności (zakładam, że kompilator GCC jest już zainstalowany):

[1] Wejdź do *Panel sterowania/System i konserwacja/System*, następnie *Zaawansowane ustawienia systemu* i *Zmienne środowiskowe*, aby dopisać po średniku nową ścieżkę.

- Uruchom terminal (okienko linii poleceń). W systemie Windows można to zrobić, wywołując z menu *Start* program o nazwie *cmd* (*Start/Wszystkie programy/Akcesoria/Wiersz polecenia*), w Mac OS uruchom program *Terminal*. W Linuksie wyszukaj w swoim środowisku graficznym program Terminal (np. w Suse może to być *Komputer/Pokaż wszystkie programy/Terminal GNOME*).
- Przejdź do folderu, w którym znajduje się interesujący Cię plik w języku C++ rozpakowany z archiwum ZIP dołączonego do tej książki. Do nawigacji w linii poleceń możesz używać komend: `cd \`, `cd..`, `cd nazwa` pozwalających na, odpowiednio: przejście do folderu głównego, przejście do folderu nadrzędnego, przejście do podfolderu *nazwa*). W Ubuntu wskaż myszką folder, np. *CPP* i prawym klawiszem myszki wywołaj polecenie *Otwórz w terminalu*.
- Wpisz z linii poleceń komendę kompilacji:

```
c++ -o plik plik.cpp
```

- Powyższe polecenie skompiluje przykład w C++ zawarty w pliku o nazwie `plik.cpp`, tworząc w tym samym katalogu jego wersję wykonywalną o tej samej nazwie (w zależności od systemu może zostać doklejone rozszerzenie, np. dla Windows będzie to *.exe*).
- Skompilowany do postaci binarnej program możesz uruchomić już klasycznie, klikając go dwukrotne myszką lub wywołując z tej samej konsoli tekstowej (przykłady na rysunku C.1). Oprócz komendy `cd` możesz używać `ls` do wyświetlania zawartości katalogu. Uruchomienie pliku wykonywalnego wymaga poprzedzenia go symbolami *./* (kropka i ukośnik), co stanowi cechę charakterystyczną terminala opartego na powłoce bash. W linii poleceń Windows wystarczy, że wpiszesz nazwę pliku wykonywalnego łącznie z rozszerzeniem *.exe*.



Rysunek C.1. *Kompilacja i uruchamianie programu w konsoli tekstowej*

Visual C++ Express Edition

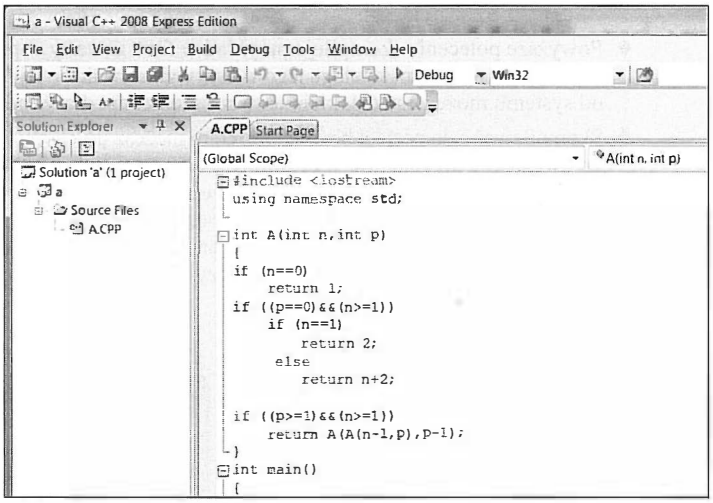
W tym punkcie opisuję przypadek użycia graficznego środowiska Visual C++ Express Edition dostępnego dla systemu Windows, które pozwala na kompilowanie programów zarówno graficznych, jak i pracujących w trybie tekstowym.

Dla ułatwienia na *ftp* zawarłem gotowe projekty pod to środowisko, opiszę jednak także, jak utworzyć nowy konsolowy projekt w tym kompilatorze na podstawie gotowego pliku CPP.

Otwieramy gotowy projekt

Aby otworzyć istniejący projekt stworzony w Visual C++ Express Edition, należy wykonać następujące czynności (rysunek C.2):

- Uruchom program Visual C++ z menu Start.
- Wybierz z menu *File/Open*, a następnie *Project/Solution* (możesz też użyć skrótu *Ctrl+Shift+O*).
- Wskaż na dysku plik projektu (rozszerzenie *.vcproj*) i otwórz go (przycisk *Otwórz*).
- Komplikacja i utworzenie kodu wynikowego: klawisz *F7* (także menuBuild/*Build Solution*).
- Uruchomienie kodu: klawisz *F5*.



Rysunek C.2. *Otwieramy gotowy projekt Visual C++ Express Edition*

Visual C++ po otwarciu projektu powinien w lewym panelu pokazać drzewko *Solution Explorer*, w którym można wyszukać pliki źródłowe (ang. *Source Files*) i załadować je do edytora wizualnego, podwójnie klikając w ich nazwę. Edytor w środowisku Visuala jest znakomity, podświetlanie słów kluczowych, pomoc w formatowaniu (wcięcia) i pomoc składniowa na pewno ułatwią pracę każdemu programiście.

Jeśli panel *Solution Explorer* został przez przypadek zamknięty i straciliśmy możliwość efektywnej pracy z kompilatorem, to nie stało się nic strasznego, w menu *View* można ponownie włączyć ten widok.

Warto mieć świadomość, że program konsolowy po uruchomieniu z poziomu środowiska Visual C++ może otworzyć swoje okno i szybko je zamknąć.

Aby go „zatrzymać" na ekranie, można do kodu funkcji `main` wprowadzić na koniec np. prostą instrukcję pobierania danych:

```
char klawisz; cin >> klawisz ;
```

lub

```
cin.get();
```

W systemie DOS lub Windows można użyć funkcji getch():

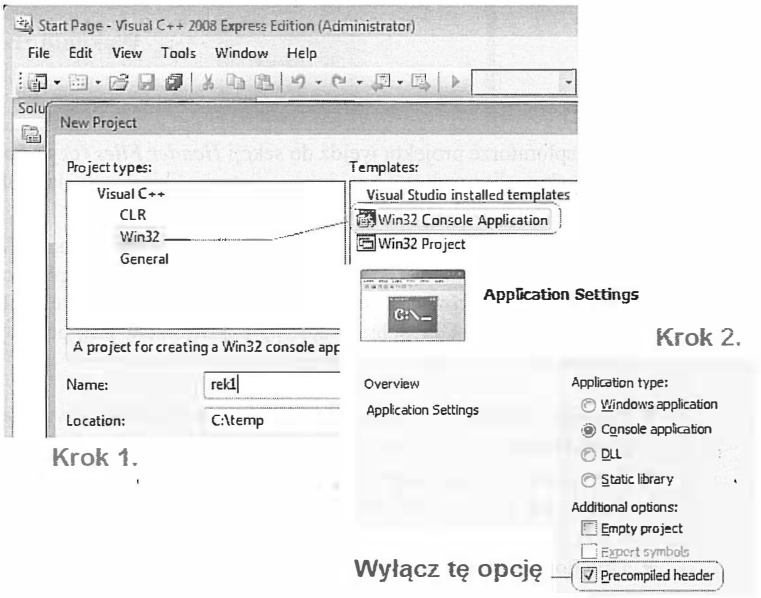
```
#include <conio.h>  // z powodu getch
...
getch();
```

Tworzymy nowy projekt konsolowy w Visual C++

Na pewno prędzej czy później przyda nam się wiedza, jak utworzyć projekt w Visual C++. Niekoniecznie musimy dysponować plikiem źródłowym, ale załóżmy na początek, że takowy posiadamy i chcemy go np. dostosować do naszych potrzeb.

Aby na podstawie gotowego pliku *.cpp* utworzyć nowy projekt „gołego" języka C++, który będzie działał w konsoli tekstowej, należy wykonać następujące czynności:

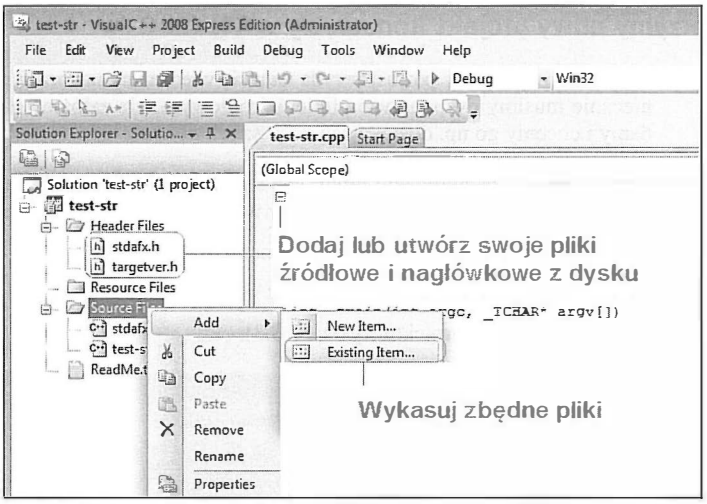
- Uruchom program Visual C++ z menu Start.
- Wybierz z menu *File/New/Project/Win32* typ *Win32 Console Application.*
- Wpisz nazwę projektu (rysunek C.3).



Rysunek C.3. *Utworzenie tzw. projektu w Visual C++ Express Edition*

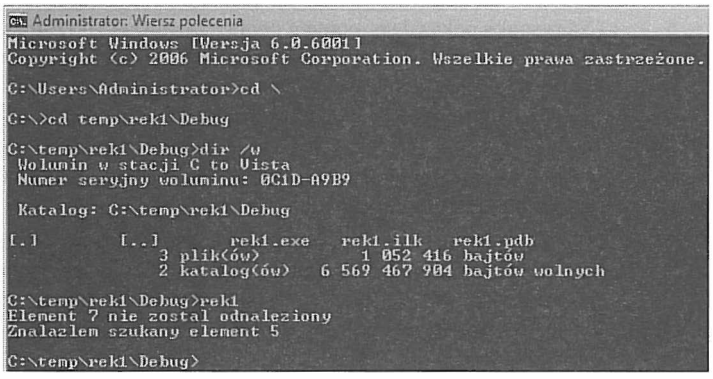
(Na rysunku pokazałem moment utworzenia projektu o nazwie *rek1*, którego pliki będą zawarte w folderze *c:\temp*. Dla uproszczenia nazwałem projekt tak samo, jak plik źródłowy programu, którego w nim użyję, ale oczywiście nie jest to obowiązkowe).

- W okienku *Win 32 Application Wizard*, który pojawi się na ekranie, wciśnij klawisz *Next* i wyłącz opcję *Precompiled headers*. To samo możesz zrobić później, w menu Project\<nasza nazwa projektu>\Properties, w sekcji C/C++/Precomplied Headers (działa dopiero po załadowaniu pliku *.cpp*).
- W kolejnym etapie (rysunek C.4) znajdziesz się już na ekranie znanego nam deweloperskiego środowiska graficznego (IDE), oferującego wygodny eksplorator plików projektowych i świetny edytor, wyposażony w funkcje wyróżniania słów kluczowych języka C++ i mechanizmy poprawiania formatowania kodu (np. wcięcia).
- W eksploratorze projektu wejdź do sekcji *Source Files* (czyli po polsku „pliki źródłowe") i wykasuj (klawisz *Delete*) znajdujące się tam pliki (wersje na dysku twardym pozostaną nienaruszone; jeśli chcesz, to również je usuń).



Rysunek C.4. *Kompilacja projektu w Visual C++ Express Edition*

- W eksploratorze projektu wejdź do sekcji *Header Files* (czyli po polsku „pliki nagłówkowe") i wykasuj znajdujące się tam pliki (wersje na dysku twardym pozostaną nienaruszone; jeśli chcesz, to również je usuń).
- Od tego momentu możesz dodawać do projektu swoje własne pliki źródłowe: kliknij prawym klawiszem myszki na sekcji *Source Files* i z menu podręcznego wywołaj polecenie *Add/Existing item* (po polsku: dodaj istniejący element). Wskaż interesujący Cię plik źródłowy CPP pobrany np. z *ftp* lub stworzony przy pomocy zwykłego edytora (systemowy Notepad wystarczy!).
- Naciśnij klawisz *F7*, aby zbudować plik wykonywalny zawierający kod programu przykładowego. Jeśli kompilator wykryje błędy, popraw je[2].
- Przy pomocy menu Start uruchom program `cmd` (*Wiersz polecenia*) i przejdź do folderu Debug lub Release zawartego w lokalizacji podanej w polu Location na rysunku C.3). W naszym przykładzie będzie to `c:\temp\rek1\Debug`.
- Uruchom skompilowany program (rysunek C.5).

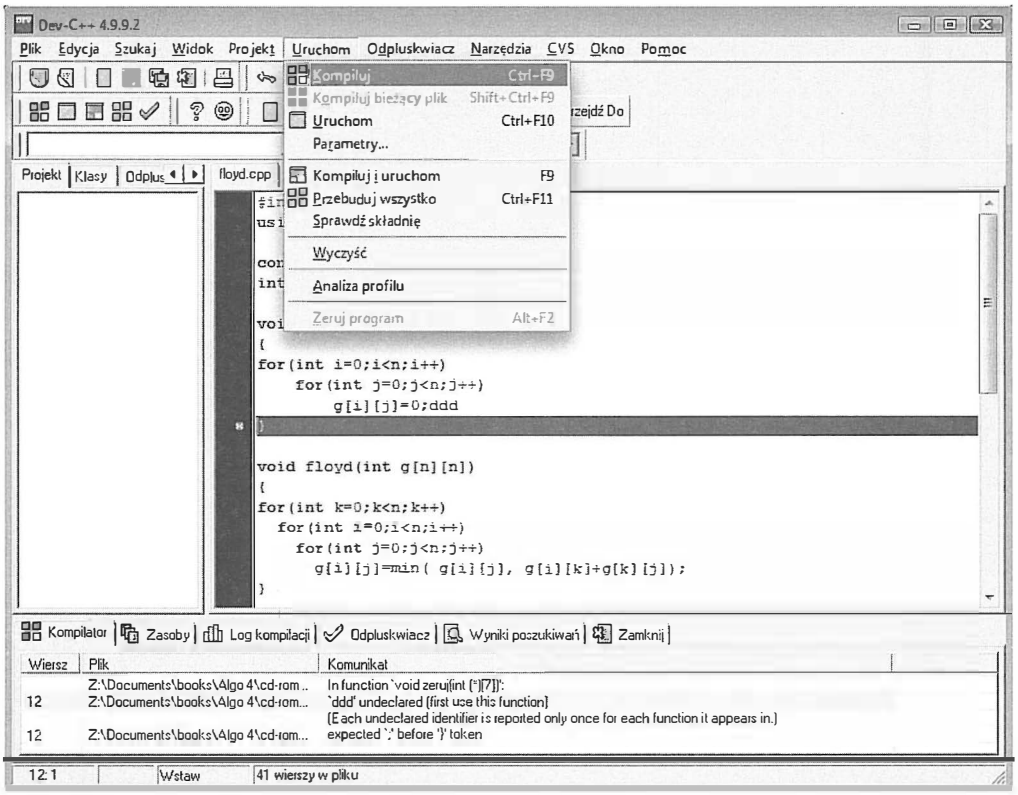


Rysunek C.5. *Uruchamianie skompilowanego programu konsolowego w systemie Windows*

[2] Programy są oczywiście sprawdzone przeze mnie, ale możesz je modyfikować, co czasami prowadzi do błędów.

Dev C++

Środowisko Dev C++ (rysunek C.6) jest dość popularne wśród początkujących programistów, gdyż nie wymaga zbyt dużo miejsca (pliki instalacyjne zajmują około 10 MB, a po rozpakowaniu na dysku około 60 MB). Podczas instalacji jedyne, na co trzeba uważać, to okienko konfiguracyjne, w którym możemy wybrać język polski dla komunikatów i graficznego interfejsu użytkownika.



Rysunek C.6. *Darmowe środowisko Dev C++*

Kompilacja programu przykładowego nie wymaga nawet założenia pliku projektu. Wystarczy otworzyć plik w edytorze (*Plik/Otwórz*) i skompilować go przy pomocy polecenia *Uruchom/Kompiluj* (*Ctrl+F9*). Jeśli w programie nie było błędów, to w tym samym folderze, w którym znajduje się plik źródłowy *.cpp*, zostanie zapisany plik wykonywalny (*.exe*), który można uruchomić np. z linii poleceń lub z menu *Uruchom/Kompiluj* (*Ctrl+F10*). Polonizacja Dev C++ nie jest stuprocentowa, ale nawet osoba nieznająca angielskiego z łatwością zrozumie ostrzeżenia i komunikaty o błędach w programach, gdyż program wskazuje linijki kodu, w których jest błąd.