C++ Wydanie II

Andrzej Stasiewicz

Helion

TABLICE INFORMATYCZNE

BUDOWA PROGRAMU

Najprostszy program w C++

Standard C++ wymaga nagłówków biblioteki standardowej bez rozszerzenia *.h*, starych nagłówków w wersji z literą c (np. `cmath`) i dołączenia przestrzeni nazw `std`. Funkcja `main()` niemusisię kończyćfrazą `return`.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
 double a;
 cin >> a;
 cout << "sinus(" << a << ") = "
 << sin(a);
}
```

Najprostszy program w C++

Starsze kompilatory mogą wymagać następującej treści:

```
#include <iostream.h>
#include <math.h>
int main()
{
 double a;
 cin >> a;
 cout << "sinus(" << a << ") = "
 << sin(a);
 return 0;
}
```

Pliki źródłowe

Treść deklaracji zwyczajowo umieszczamy w plikach o rozszerzeniu *.h* (tzw. nagłówkach). Treść implementacji znajduje się w plikach o rozszerzeniu *.cpp*. W pierwszych liniach plików implementacyjnych *.cpp* zazwyczaj sytuujemy dyrektywy `#include <nazwa nagłówka>` (porównaj podrozdział „Dyrektywy preprocesora"). Proste programy przygotowujemy bez wyodrębniania części *.h*, spisując deklaracje w początkowych fragmentach pliku *.cpp*.

Przykład

Zawartość pliku *nazwa_pliku.h*:

```
#define MAXX 640
const double pi = 3.14;
double srednia(double a, double b);
```

Zawartość pliku *nazwa_pliku.cpp*:

```
#include "nazwa_pliku.h"
using namespace std;
int main()
{
...
}
double srednia(double a, double b)
{
...
}
```

DYREKTYWY PREPROCESORA

Preprocesor przetwarza tekst programu przed kompilacją. Wszystkie dyrektywy preprocesora zaczynają się od znaku `#`.

`#include <nazwa_pliku>`	Wstawia treść pliku (zazwyczaj nagłówkowego) biblioteki
`#include "nazwa_pliku"`	Wstawia treść pliku (zazwyczaj nagłówkowego) użytkownika
`#define WERSJA_1`	Określa napis `WERSJA_1` (do kompilacji warunkowej)
`#ifdef WERSJA_1`	Kompiluje, jeśli napis `WERSJA_1` jest określony
`#ifndef WERSJA_1`	Kompiluje, jeśli napis `WERSJA_1` nie jest określony
`#endif`	Kończy obszar zapoczątkowany przez `#ifdef` albo `#ifndef`
`#undef WERSJA_1`	Odwołuje napis `WERSJA_1`
`#define MAXX 640`	Napisy `MAXX` zastępuje napisem `640`
`#define MAX(a,b) ((a)>(b)?(a):(b))`	Definiuje makropolecenie, tutaj maksimum dwóch liczb

Przestrzenie nazw

Aby zapobiec konfliktom nazw w obrębie tekstu źródłowego (np. podczas pracy zespołowej), wprowadzono słowo kluczowe `namespace`.

`namespace Kowalski {treść programu}`	Zamknięcie fragmentu programu w swojej przestrzeni nazw
`Malinowski :: wydruk();`	Odwołanie do elementu określonego w innej przestrzeni nazw
`using namespace Malinowski;`	Trwałe podłączenie do innej przestrzeni nazw

Wszystkie identyfikatory biblioteki standardowej są zdefiniowane w przestrzeni nazw `std`, stąd w zasadzie każdy współczesny program zaczyna się od deklaracji `using namespace std;`.

Przykład

```
#include <iostream>
using namespace std;
int main()
{
 std::cout << "Jan Kowalski";
}
```

Standardowe wejście i wyjście

W pliku nagłówkowym `iostream` biblioteki standardowej są zadeklarowane klasy oraz operatory realizujące pobieranie danych z klawiatury i wypisywanie danych na ekran.

`cin >> a;`	Pobranie wartości do zmiennej `a` (która musi być zadeklarowana)
`cin >> a >> b >> c;`	Pobranie kaskadowe kilku wartości
`cout << a;`	Wypisanie (wyprowadzenie) wartości zmiennej `a`
`cout << "Wartosc a = " << a;`	Wypisanie kaskadowe kilku wartości
`cout.width(20);`	Ustalenie szerokości pola do wyprowadzenia zmiennej
`cout.fill('*');`	Ustalenie znaku wypełniającego nadmiar szerokości
`cout.precision(2);`	Ustalenie liczby miejsc po przecinku wyprowadzanej zmiennej

Przykład

```
#include <iostream>
using namespace std;
int main()
{
 double a;
 cin >> a;
 cout.width(20);
 cout.precision(2);
 cout << "Liczba = " << a;
}
```

INSTRUKCJE STERUJĄCE

Instrukcja grupująca (blok instrukcji)

```
{
 instrukcja 1;
 instrukcja 2;
 ...
}
```

Zbiór instrukcji ujętych w instrukcji grupującej jest traktowany jak jedna instrukcja. Taka instrukcja umożliwia deklarowanie danych lokalnych, widocznych tylko w jej obrębie. Stosowana jest głównie w warunkach logicznych i pętlach.

Przykład

```
if(a < 0)
{
 cout << "a jest mniejsze od zera
...";
 a = 10;
}
```

Instrukcja wykonania warunkowego if ... else

```
if(warunek logiczny)
{
 instrukcje A;
}
else
{
 instrukcje B;
}
```

Realizuje polecenie: „jeśli warunek jest spełniony — wykonaj instrukcje A, w przeciwnym wypadku — instrukcje B". Części od `else` w dół może nie być. Instrukcje grupujące `{...}` są potrzebne, jeśli mamy wykonać warunkowo więcej instrukcji.

Przykład

```
if(a < 100)
   b = 0;
else
{
   b = 1;
   c = 0;
}
```

Częste błędy

- Umieszczenie średnika za frazą `if(...)` i przed instrukcją grupującą.
- Pominięcie instrukcji grupującej, jeżeli jest niezbędna.

Zwrotnica wielokierunkowa switch() { case ...}

```
switch(wyrażenie_klucz)
{
 case wartosc_1: instrukcje;
                 break;
 case wartosc_2: instrukcje;
                 break;
 ...
 default: instrukcje;
}
```

Dopasowuje wartość klucza do etykietek we frazach `case` i realizuje instrukcje z odpowiedniej szufladki. Sformułowanie `wyrażenie_klucz` musi być typu wyliczeniowego (znak, liczba całkowita). Klamry są obowiązkowe — w tym wypadku nie oznaczają instrukcji grupującej. Wariant `default` jest realizowany wtedy, gdy klucz nie pasuje do etykietki żadnego wariantu `case`. Wariant `default` nie jest konieczny. Szufladki nie muszą być spisywane w jakimś ustalonym porządku.

Przykład

```
switch(a)
{
 case 0:
 case 1:
        cout << "Jan Kowalski";
        break;
 case 17:
        b = 1;
        break;
 default:
        cout << "Niewlasciwa
        wartosc !!!";
}
```

Częste błędy

- Brak w którejś szufladce frazy `break` na zakończenie algorytmu (od razu wykona się następna szufladka).
- Ta sama wartość etykiety kilku szufladek.

Pętla for (...; ...; ...)

```
for(wyrażenie inicjujące; warunek
logiczny; wyrażenie modyfikujące)
{
 instrukcje;
}
```

Ma w nagłówku dwa średniki, które wyznaczają trzy pola. Pierwsze pole wykonuje się jednorazowo przy wejściu do pętli — zazwyczaj zawiera instrukcję inicjowania licznika obrotów. Drugie pole wykonuje się przed rozpoczęciem każdego obrotu pętli i zawiera warunek logiczny, warunkujący wykonanie obrotu. Trzecie pole wykonuje się na zakończenie każdego obrotu i zazwyczaj zawiera modyfikację licznika obrotów.

Przykład

```
for(i = 0; i < 100; ++i)
{
   cout << "Obrót pętli nr "
   << i + 1;
}
```

Częsty błąd

- Postawienie średnika tuż za pętlą, a przed instrukcjami, które mają być powtarzane.

Pętla for (... : ...) dla tablic i kontenerów

```
for(zmienna iterująca : tablica)
{
 instrukcje;
}
```

Dostępna w standardzie c++11.

Przykład

```
int tablica[3] = {1,2,3};
for(int &x : tablica)
{
   cout << "Element " << x << endl;
   x = x + 3;
}
```

Pętla while()

```
while(warunek logiczny)
{
   instrukcje;
}
```

Pętla o tym samym charakterze co pętla `for`, jednak bez zaimplementowanych pól inicjowania i kończenia obrotu. Zazwyczaj stosuje się ją tam, gdzie nie wiadomo z góry, ile obrotów zostanie wykonanych.

Przykład

```
i = 0;
while (i < 100)
{
   cout << " Obrót pętli nr "
   << i + 1;
   i = i + 1;
}
```

Porównaj analogiczny przykład dla pętli `for`.

Częste błędy

- Postawienie średnika za nagłówkiem pętli, a przed instrukcjami, które mają być powtarzane.
- Pominięcie klamer instrukcji grupującej, mimo że powtarzanych ma być kilka instrukcji.

Pętla do ... while()

```
do
{
   instrukcje;
}while(warunek logiczny);
```

Pętla sprawdza warunek logiczny *po* wykonaniu instrukcji, zatem zawsze wykona się przynajmniej jeden raz. Dlatego nie może zastępować pętli `for` i `while`, które sprawdzają warunki logiczne *przed* wykonaniem instrukcji.

Przykład

```
do
{
   cin >> c;
   a = a + 1;
} while(c != 'k');
```

Częsty błąd

- Umieszczenie w algorytmie, który wymaga pętli `for` lub `while`.

Instrukcja break

```
break;
```

Przerywa działanie każdej pętli. Zobacz także instrukcję `switch`, w której instrukcja `break` wyznacza koniec algorytmu szufladki `case`.

Przykład

```
while(i < 100)
{
   i = i + 1;
   if(i > 20)
      break;
}
```

Instrukcja continue

```
continue;
```

Przerywa działanie bieżącego obrotu pętli i przechodzi do następnego.

Przykład

```
while(i < 100)
{
   i = i + 1;
   if(i < 20)
      continue;
   cout << " Obrót pętli nr "
   << i + 1;
}
```

Instrukcja „wyrażeniowe if"

```
a = (warunek logiczny) ? wyrażenie_na_
tak : wyrażenie_na_nie;
```

Zbliżona charakterem do instrukcji `if ... else`, może być przez nią zastąpiona. Zwraca wartość, zatem można ją wbudować w wyrażenie arytmetyczne. Dwa warianty wyrażeń muszą dostarczać wartości tego samego typu.

Ciąg dalszy na str. 2

Przykład

```
a = b < 100 ? 1 : cos(pi);
```

Powyższy przykład można zrealizować także za pomocą instrukcji `if ... else`, jednak w dłuższym zapisie:

```
if(b < 100)
   a = 1;
else
   a = cos(pi);
```

Instrukcja ta może być zagnieżdżana, co wymaga uważnego opatrzenia nawiasami:

```
a = b < 100 ? (c < 100 ? 1 : 0) :
cos(pi);
```

Instrukcja goto

```
goto etykieta;
instrukcje;
etykieta:
   instrukcje;
```

Skok do miejsca programu określonego etykietą zakończoną dwukropkiem.

Przykład

```
for(i = 0; i < 100; ++i)
   for(j = 0; j < 100; ++j)
      if(i + j == 123)
         goto AWARIA;
AWARIA:
cout << " Wyjście z pętli ...";
```

Częste błędy

- Nadużywanie `goto`.
- Próba opuszczenia funkcji (niedozwolony przeskok z funkcji do funkcji).

TYPY DANYCH

Typy arytmetyczne

Parametry typów zależą od kompilatora i platformy. Rozmiar minimalny (gwarantowany) określamy w celu zapewnienia przenośności. Dla typów rzeczywistych (`float` i `double`) minimum oznacza zbliżenie do zera.
Charakterystyka typów arytmetycznych (kompilator gcc).

Nazwa	Rozmiar minimalny	Rozmiar	Minimum	Maksimum
`char`	1 bajt	1 bajt	–128	127
`unsigned char`	1	1	0	255
`short int`	2	2	–32768	32767
`unsigned short int`	2	2	0	65535
`int`	2	4	–2147483648	2147483647
`unsigned int`	2	4	0	4294967295
`long int`	4	4	–2147483648	2147483647
`unsigned long int`	4	4	0	4294967295
`long long`	4	8	–9223372036854775808	9223372036854775807
`unsigned long long`	4	8	0	18446744073709551615
`float`	4	4	1.17549e–38	3.40282e+38
`double`	8	8	2.22507e–308	1.79769e+308
`long double`	12	12	brak danych	brak danych

Jak poznać charakterystykę typu arytmetycznego?

Na przykładzie typu `unsigned int`:

```
#include <limits>
using namespace std;
...
cout << "Minimum: " << numeric_
limits< unsigned int > :: min();
cout << "Maksimum: " << numeric_
limits< unsigned int > :: max();
```

We wcześniejszych dialektach C++ na przykładzie typu `long int`:

```
#include <limits.h>
...
cout << "Minimum: " << LONG_MIN;
cout << "Maksimum: " << LONG_MAX;
```

Rozmiar typu lub zmiennej o danym typie zwraca operator `sizeof()`:

```
long double d;
cout << "Rozmiar w bajtach: "
<< sizeof(d);
cout << "Rozmiar w bajtach: "
<< sizeof(long double);
```

Deklarowanie i inicjalizowanie zmiennych arytmetycznych

```
Typ zmienna = napis inicjalizujący;
```

Przykłady dopuszczalnych napisów inicjalizujących dla typów arytmetycznych.

Typ	Napisy inicjalizujące	Komentarz
Wszystkie typy arytmetyczne	`123`, `'a'`, `0xFF`	Inicjalizowanie dziesiętne, znakowe, szesnastkowe
Typy `unsigned`	Jak w wierszu 1. i `123456u`	Inicjalizowanie wartością nieujemną
Typy `long`	Jak w wierszu 1. i `1000000L`	Inicjalizowanie wartością długą
Dodatkowo typy `unsigned long`	Jak w wierszu 2. i `100000uL`	Inicjalizowanie wartością długą nieujemną
`long long`, `unsigned long long`	Jak wszystkie poprzednie i `100LL`, `100uLL`	Inicjalizowanie wartością długą, długą nieujemną
Dodatkowo wszystkie typy zmiennoprzecinkowe	`123.456`, `123.456e–8`	Inicjalizowanie dziesiętne i inżynierskie

Typ tekstowy

Standard C++ definiuje pełnowartościowy typ `string`.

Przykład deklarowania, inicjalizowania, dodawania i wyprowadzania

```
#include <string>
using namespace std;
...
string s1 = "Jan", s2 = "Kowalski";
string txt = s1 + "  " + s2;
cout << txt;
```

Wcześniejsze dialekty C++ nie mają wydzielonego typu tekstowego — jest nim wskaźnik dla ciągu znaków, koniecznie zakończony bajtem zerowym, co pozwala na definiowanie bardzo długich tekstów. Nie ma wydzielonego typu, ale jest inicjalizator tekstowy, dopisujący na końcu zero.

Przykład

```
char *z = "Jan ", zz[9] = "Kowalski";
cout << z << zz;
```

Inne użyteczne typy

bool

Typ dwuwartościowy do oznaczania wartości wyrażeń logicznych.

Przykład

```
bool a, b = true, c = false,
d = (a < 5);
```

enum

Typ wyliczeniowy, definiujący elementy zbioru i przypisujący im wartości w porządku rosnącym.

Przykład

```
enum Dni {Pon=1,Wto, Sro, Czw, Pia,
Sob, Nie, Pozaplanetarne=100};
Dni dd = Pia;
```

void

Typ pusty do oznaczania wskaźników niezainicjalizowanych oraz funkcji niezwracających wartości albo niepobierających argumentów.

Przykład

```
void *wskaznik_pusty;
void procedura(void);
```

auto

Dla standardu c++11. Kompilator sam rozpoznaje typ deklarowanej zmiennej.

Przykład

```
int a = 5;
auto b = a;   // int b = a;
```

decltype()

Dla standardu c++11. Typ taki, jaki ma wcześniej zadeklarowany obiekt.

Przykład

```
double pi = 3.14;
decltype( pi) r;
```

Modyfikatory typów

const

Oznaczenie danej stałej. Dana stała powinna być zainicjowana w momencie deklaracji. Może być oznaczeniem wskaźnikowych lub referencyjnych rezultatów albo argumentów funkcji.

Przykład

```
const double pi = 3.14;
void drukuj(const Student &student);
```

static

Oznaczenie danej, która istnieje przez cały czas życia programu (nawet wtedy, gdy jest zadeklarowana lokalnie). W klasach może być oznaczeniem pola, które jest wspólne dla wszystkich egzemplarzy (obiektów).

Przykład

```
static int liczba_uruchomien;
```

register

Oznaczenie danej, która powinna być przechowywana w pamięci podręcznej (w rejestrze procesora), bo jest intensywnie eksploatowana.

Przykład

```
register int indeks;
```

volatile

Przeciwieństwo `register` — dana, która musi być zawsze pobierana z oryginalnej lokalizacji w pamięci, bo inny proces mógł ją zmodyfikować.

Przykład

```
volatile int liczba_wlaczonych_
komputerów;
```

extern

Oznaczenie danej, której deklaracja znajduje się w innym module (innym pliku źródłowym). W programie może wystąpić tylko jedna deklaracja bez słowa `extern`.

Przykład

```
int MAXX; // w jednym pliku źródłowym
...
extern int MAXX; // we wszystkich
pozostałych plikach źródłowych
```

Dynamiczne deklarowanie zmiennych

```
Typ *adres = new Typ;
...
delete adres;
```

Dynamiczne tworzenie zmiennych określonego typu polega na zadeklarowaniu wskaźnika dla tego typu i utworzeniu pod jego adresem obszaru pamięci przeznaczonego na zmienną. Zmienna utworzona dynamicznie koniecznie musi być zlikwidowana, gdy nie jest już potrzebna.

Przykład

```
double *adres = new double;
*adres = 17.1;
cout << sin(*adres);
delete adres;
```

Częste błędy

- Zwalnianie pamięci przy użyciu operatora `delete[]` (porównaj podrozdział „Tablice"), a nie `delete`.
- Dwukrotne wywołanie operatora `delete`.
- Brak słowa `delete` — czyli tzw. błąd wycieku pamięci.

TABLICE

```
Typ nazwa_tablicy[liczba elementów];
```

Są to spójne grupy zmiennych tego samego typu. Poszczególne zmienne zgromadzone w tablicy są dostępne za pomocą indeksu liczonego od zera do wartości `liczba_elementów – 1`.

Przykład

```
int tab[100];
for(int i = 0; i < 100; ++i)
   tab[i] = 0;
```

Częste błędy

- Operowanie indeksem poza zadeklarowanym obszarem tablicy.
- Nieuwzględnienie faktu, że pierwszy element tablicy ma indeks zerowy.

Deklarowanie i inicjalizacja

Inicjalizacja tablic polega na przytoczeniu w klamrach ciągu wartości oddzielonych przecinkami (wartości te zostaną przypisane do kolejnych elementów). Napisy określające wartości tworzymy według takich samych reguł jak inicjalizowanie zmiennych nietablicowych (porównaj podrozdział „Typy danych").

Przykłady

```
int A[100], B[3] = { 1, '3', 0xFF};
long double R[4] = {0, 1.2, 2.3e-17,
'c'};
```

Kiedy elementów w klamrach jest mniej niż zadeklarowany rozmiar tablicy — zostaną zainicjowane początkowe elementy, a pozostałe będą wyzerowane.

Przykład

```
unsigned int A[100] = {1, 2, 3, 123456u};
```

Przy inicjalizowaniu tablicy nie musimy określać jej rozmiaru — zostanie on ustalony według liczby elementów inicjalizujących.

Przykład

```
int A[] = {1, 2, 3};
```

Tablice wielowymiarowe

```
Typ nazwa_tablicy[liczba elementów]
[liczba elementów] [...];
```

Obowiązują tutaj te same reguły deklarowania i inicjalizowania, a także określania liczby elementów na podstawie postaci inicjalizatora.

Przykład

```
int A[10][20], B[2][2][2];
int C[][5] = {{1, 2, 3}, {2, 3, 4, 5,
6}};  // tablica B[2][5]
double D[][] = {{1, 2}, {1, 2, 3, 4,
5}, {1}, {1}};  // tablica C[4][5]
```

Tablice dynamiczne i operatory new[] i delete[]

```
Typ *adres = new Typ[liczba
elementów];
...
delete[] adres;
```

Dynamiczne tworzenie tablic określonego typu polega na zadeklarowaniu wskaźnika dla tego typu i utworzeniu pod jego adresem takiego obszaru pamięci, by zmieściła się w nim planowana liczba elementów. Tablica utworzona dynamicznie powinna być zlikwidowana, gdy nie jest już potrzebna (nie jest likwidowana automatycznie i powstaje błąd zwany *wyciekiem pamięci*).

Przykład

```
double *adres = new double[10];
adres[0] = 3.14;
cout << adres[0];
delete[] adres;
```

Przykład

Tablica typu `int` o *10* wierszach i *20* kolumnach:

```
int **adr;
adr = new int *[10];
for(int i = 0; i < 10; ++i)
   adr[i] = new int[20];
...
for(int i = 0; i < 10; ++i)
   delete[] adr[i];
delete[] adr;
```

Częste błędy

- Brak pewności, czy pamięć pod tablicę została przydzielona.
- Zwalnianie pamięci za pomocą operatora `delete`, a nie `delete[]`, albo odwrotnie.
- Przeoczenie zwolnienia pamięci lub dwukrotne zwolnienie pamięci.

Tablice informatyczne. C++. Wydanie II

Tablice w standardzie C++

```
#include <vector>
using namespace std;
...
vector<Typ> tablica;
```

Standard C++ dostarcza wzorca tablicy (porównaj podrozdział „Szablony (wzorce) funkcji i klas") dowolnego typu, zwanego *wektorem*.

Przykład

Wektor liczb `double`:

```
#include <iostream>
#include <vector>
using namespace std;
...
vector <double> Q(10, 3.14);
// tablica 10 liczb double,
zainicjowanych wartością 3.14
cout << Q[0] << endl;
Q.push_back(-3.14);  // poszerzenie
tablicy przez dopisanie elementu
```

WSKAŹNIKI

Wskaźnik jest *adresem* uzupełnionym informacją o *typie zmiennej*, która znajduje się pod owym adresem.

Operatory * i &

```
zmienna = * wskaznik;
wskaznik = & zmienna;
```

Operator *wyłuskania* * poprzedza wskaźnik i zwraca wskazywaną przez niego wartość zmiennej.
Operator *adresacji* & poprzedza zmienną i zwraca wskaźnik do niej.

Przykład

```
int *adr, a;
a = *adr;     // wyłuskanie wartości
spod wskaźnika adr
adr = &a;     // uzyskanie wskaźnika
do zmiennej a
```

Deklarowanie i inicjalizowanie wskaźników

```
Typ *nazwa_zmiennej_wskaźnikowej;
```

Przykład

```
double *adr1, *adr2;
```

Wskaźniki inicjalizuje się albo przez przypisanie ich do istniejącej zmiennej tego samego typu, albo za pomocą operatora żądania pamięci `new` lub wielu komórek pamięci `new[]` (porównaj podrozdział „Tablice dynamiczne i operatory `new[]` i `delete[]`").

Przykład

```
int il_mies = 12;
int *adr1 = &il_mies, *adr2 = new
int, *adr3 = new int[10];
```

Częste błędy

- Przydzielenie wskaźnikowi pamięci, ale niezwolnienie jej przy użyciu operatora `delete` lub `delete[]`.
- Pomylenie operatorów `delete` i `delete[]`.
- Operowanie na niezainicjalizowanym wskaźniku.

Wskaźnik shared_ptr<> i unique_ptr<>

```
#include <memory>
shared_ptr<Typ> a( new int);
unique_ptr<double b( new double);
```

Tylko w standardzie c++11. Zaletą nowych wskaźników obiektowych jest odstąpienie od ich niszczenia (nie wywołuje się instrukcji `delete`). Wiele wskaźników `shared_ptr` może wskazywać na ten sam obiekt. Tylko jeden `unique_ptr` może wskazywać na obiekt.

Przykład

```
shared_ptr<int> adr1( new int( 6));
unique_ptr<int[]> adr2( new int[3]);
*adr1 = 17;
adr2[ 0] = -11;
```

OPERATORY

Operatory arytmetyczne

Operator	Działanie
`a * b`	Mnożenie
`a / b`	Dzielenie
`a + b`	Dodawanie
`a – b`	Odejmowanie
`a % b`	Reszta z dzielenia (modulo)
`++a`	Preinkrementacja (zwiększenie o 1)
`a++`	Postinkrementacja (zwiększenie o 1)
`--a`	Predekrementacja (zmniejszenie o 1)
`a--`	Postdekrementacja (zmniejszenie o 1)

Operatory arytmetyki bitowej

Operator	Działanie
`a >> n`	Przesunięcie bitów w zmiennej `a` o `n` pozycji w prawo, uzupełnienie z lewej zerami
`a << n`	Przesunięcie bitów w zmiennej `a` o `n` pozycji w lewo, uzupełnienie z prawej zerami
`a | b`	Złożenie bitów z dwóch zmiennych za pomocą relacji „lub" na każdym bicie
`a & b`	Złożenie bitów z dwóch zmiennych za pomocą relacji „i" na każdym bicie
`a ^ b`	Złożenie bitów rozłączne (XOR)
`~a`	Negacja bitów w zmiennej

Operatory przypisania

Operator	Działanie
`a = b = c`	Przypisanie, także w formie ciągu
`a *= b`	$a = a * b$
`a /= b`	$a = a / b$
`a += b`	$a = a + b$
`a –= b`	$a = a - b$
`a %= n`	$a = a \% b$
`a >>= n`	$a = a >> n$
`a <<= n`	$a = a << n$
`a &= b`	$a = a \& b$
`a |= b`	$a = a \mid b$
`a ^= b`	$a = a \wedge b$

Operatory logiczne

Operator	Działanie
`a && b`	`true`, gdy `a = true` i `b = true`
`a || b`	`true`, gdy `a = true` lub `b = true`
`!a`	`true`, gdy `a = false`
`a < b`	`true`, gdy `a` mniejsze od `b`
`a <= b`	`true`, gdy `a` mniejsze lub równe `b`
`a > b`	`true`, gdy `a` większe od `b`
`a >= b`	`true`, gdy `a` większe lub równe `b`
`a == b`	`true`, gdy `a` równe `b`
`a != b`	`true`, gdy `a` różne od `b`

Operator rozmiaru sizeof()

Zwraca rozmiar (w bajtach) danej lub typu, co jest w C++ szczególnie potrzebne, rozmiary danych zależą bowiem od kompilatora i platformy. Porównaj też podrozdział „Typy danych".

Przykład

```
int a;
cout << "Rozmiar w bajtach: "
<< sizeof(int);
cout << "Rozmiar w bajtach: "
<< sizeof(a);
```

Operatory do działania na wskaźnikach

Porównaj też podrozdział „Wskaźniki".

Operator i fraza języka	Działanie
`a = *adr;`	Wyłuskuje wartość zmiennej spod wskaźnika `adr`
`adr = &a;`	Podaje wskaźnik do zmiennej `a`

Operatory przydzielania i zwalniania pamięci

Operator i fraza języka	Działanie
`Typ *adr = new Typ;`	Przydziela pamięć do wskaźnika
`Typ *adr = new Typ[n];`	Przydziela `n` komórek pamięci
`delete adr;`	Zwalnia pamięć spod wskaźnika
`delete[] adr;`	Zwalnia wiele komórek pamięci spod wskaźnika

Inne operatory

Operator	Działanie
`wskaźnik_do_obiektu -> element`	Dostarcza składnika obiektu zadanego wskaźnikowo
`obiekt.element`	Dostarcza składnika obiektu
`tablica[a]`	Dostęp do tablic za pomocą indeksu
`()`	Argumenty funkcji, otaczanie nawiasami złożonych wyrażeń
`(Typ)a` albo `Typ(a)`	Konwersja typu danej `a` na `Typ`

Priorytety operatorów

W razie jakiejkolwiek wątpliwości należy zastosować otoczenie wyrażenia nawiasami. Operatory na początku listy mają największy priorytet.

Operatory	Komentarz
`(), [], ->, .`	Otaczanie wyrażeń nawiasami, element tablicy, element obiektu danego wskaźnikiem, element obiektu
`sizeof, ++, --, ~, !, &, *, new, new[], delete, delete[], ()`	Rozmiar, inkrementacja, dekrementacja, negacja bitów, negacja logiczna, pobranie wskaźnika, wyłuskanie wartości spod wskaźnika, przydział pamięci, zwalnianie pamięci, konwersja typu
`*, /, %`	Operatory arytmetyczne
`+, -`	Operatory arytmetyczne
`<<, >>`	Przesunięcia bitów
`<, <=, >, >=`	Relacje logiczne
`==, !=`	Relacje logiczne
`&, ^, |`	Operacje na bitach
`&&, ||`	Operatory (spójniki) logiczne
`=, *=, /=, +=, -=, <<=, >>=, &=, |=, ^=`	Przypisania, w tym z modyfikacją

FUNKCJE

Deklaracja i definicja

```
TypRezultatu nazwa_
funkcji(TypArgumentu1 argument1, ...)
throw(TypWyjatku1, ...);
```

Deklaracją funkcji jest jej nagłówek zakończony średnikiem. Nagłówek składa się z oznaczenia typu rezultatu, jaki zwraca funkcja, nazwy funkcji i listy argumentów ujętej w nawiasy. Zalecamy (choć nie wymagamy) oznaczanie typów wyjątków wyrzucanych przez funkcję (porównaj podrozdział „Obsługa sytuacji wyjątkowych").

Przykład

```
int suma(int a, int b);
double dzielenie(double a, double b)
throw(EDzieleniePrzezZero);
```

Definicją funkcji jest jej nagłówek, za którym następują nawiasy klamrowe (czyli instrukcja grupująca) z ciągiem instrukcji.

Przykład

```
int suma(int a, int b)
{
 int c = a + b;
 return c;
}
```

Deklaracje nie są wymagane, gdy definicje poprzedzają wywołanie funkcji w zasadniczym algorytmie.

Argumenty funkcji

Przekaz przez wartość

```
TypRezultatu nazwa_
funkcji(TypArgumentu1 argument1, ...);
```

Funkcja sporządza kopie argumentów. Ewentualne zmiany wartości argumentów wewnątrz funkcji nie mają wpływu na stan zmiennych poza funkcją. Jest to bezpieczne, ale niezbyt wydajne, bo wymaga kopiowania danych przy każdym wywołaniu funkcji.

Przykład

```
int suma(int a, int b)
{
 int c = a + b;
 return c;
}
```

Przekaz przez referencje (zalecany)

```
TypRezultatu nazwa_
funkcji(TypArgumentu1 & argument1,
...);
```

Funkcja pracuje na oryginalnych danych. Przekaz przez referencję jest wydajniejszy w stosunku do przekazu przez wartość, ale niebezpieczny, bo w razie modyfikacji argumentu w funkcji zmianie ulegają dane znajdujące się poza nią. Gdy taka zmiana jest niepożądana, zaleca się na liście argumentów stosować modyfikatory `const`, uniemożliwiające zmianę wartości argumentu wewnątrz funkcji.

Przykład

```
double dzielenie(double &a, double &b);
int suma(const int &a, const int &b)
{
 return a + b;
}
```

Przekaz przez wskaźniki (przez adres)

```
TypRezultatu nazwa_funkcji(TypArgumentu1
*argument1, ...);
```

Równie wydajny i równie niebezpieczny jak poprzedni przekaz. Wewnątrz funkcji zazwyczaj wymaga wyłuskiwania danych spod wskaźników.

Przykład

```
int suma(int *a, int *b)
{
 return *a + *b;
}
```

Argumenty domyślne

```
TypRezultatu nazwa_funkcji(...,
TypArgumentuN argumentN = wartość);
```

Z prawej strony listy argumentów w deklaracji albo definicji funkcji (ale nie i tu, i tu) mogą znaleźć się wartości przygotowane z góry (wartości domyślne). Jeśli w momencie wywołania funkcji nie określimy wartości tych argumentów, za ich wartości zostaną przyjęte wartości domyślne.

Przykład

```
int suma(int a, int b, int c = 0,
int d = 0)
{
 return a + b + c + d;
}
...
cout << suma(1, 2);
```

Rezultat funkcji

Nagłówek funkcji określa typ wartości przez nią zwracanej. Jeśli nie jest to typ pusty `void`, ciało funkcji musi kończyć się poleceniem `return`.

Przykład

Funkcja nie zwraca wartości:

```
void punkt(int x, int y)
{
 putpixel(x, y);
}
```

Przykład

Funkcja zwraca wartość typu `double`:

```
double srednia(double x, double y)
{
 return (x + y) / 2;
}
```

Funkcja może uczestniczyć w wyrażeniach, jeśli występuje w nich na takiej pozycji, że po wyliczeniu jej rezultatu nie dojdzie do kolizji typów.

Przykład

Funkcje `sin()` i `cos()` muszą dostarczać wartość arytmetyczną:

```
alfa = 1 + sin(x) * sin(x) + cos(x)
* cos(x);
```

Funkcje przeciążone

Funkcje o identycznej nazwie, ale o zróżnicowanych typach zwracanych i/lub listach argumentów, są traktowane jak oddzielne algorytmy. Przeciążanie podnosi czytelność programów.

Przykład

```
int suma(int a, int b);
int suma(int a, int b, int c);
double suma(double a, double b);
...
a = suma(1, 2) + suma(1, 2, 3) +
suma(1.2, 1.3);
```

Częsty błąd

- Wywołanie funkcji z takim zestawem argumentów, że kompilator nie może jednoznacznie stwierdzić, który egzemplarz ma zostać wywołany.

Wskaźniki na funkcje

```
TypRezultatu (*nazwa_funkcji)
(TypArgumentu1 argument1, ...);
```

Funkcje mogą być wywoływane za pośrednictwem wskaźników. Deklarując wskaźnik, zawsze trzeba określić cechy funkcji — zwracany rezultat, liczbę i typy jej argumentów. Funkcje różniące się tymi cechami dostarczają wskaźników o różnych typach. Wskaźnikiem na funkcję jest też nazwa funkcji.

Przykład

```
double srednia(double x, double y)
{
 return (x + y) / 2;
}
...
double (*adres_sredniej)(double,
double ), z;
adres_sredniej = srednia;
z = adres_sredniej(1, 2);
```

KLASY

Klasa jest typem złożonym, składającym się z danych i funkcji (zwanych metodami).

Deklaracja i definicja

```
class NazwaTypu
{
 deklaracje danych i funkcji;
};
```

Deklaracja jest zapowiedzią klasy i polega na przytoczeniu jej ustroju w klamrach umieszczonych za słowem kluczowym `class`. Definicja klasy oznacza definicję jej funkcji. Definicje można przytaczać bezpośrednio w deklaracji, a także poza nią.

Przykład

Jedna funkcja zdefiniowana w deklaracji, druga poza nią:

```
class Wymierna
{
 private:
    int licznik, mianownik;
 protected:
 public:
    Wymierna(int l, int m){licznik =
    l; mianownik = m;};
    void wypisz(void);
};
void Wymierna :: wypisz(void)
{
 cout << licznik << "/" << mianownik;
}
```

Częsty błąd

- Funkcja definiowana poza klasą nie jest zaopatrzona w etykietę przynależności (tutaj `Wymierna ::`).

Ograniczanie zasięgu elementów klasy

`private`	Elementy dostępne tylko dla funkcji wchodzących w skład klasy
`protected`	Tak jak `private`; dodatkowo dostępne w klasach będących potomkami (pochodnymi) klasy (porównaj podrozdział „Dziedziczenie")
`public`	Elementy dostępne dla wszystkich funkcji

Funkcje zaprzyjaźnione

```
Typ funkcja(lista argumentów);
...
class Nazwa
{
 friend Typ funkcja(lista argumentów);
...
};
```

Ograniczenia widoczności elementów klasy wyjątkowo naruszają funkcje zewnętrzne (pozaklasowe), które — oprócz swojej normalnej deklaracji i definicji — są w klasie zadeklarowane jako zaprzyjaźnione z nią.

Przykład

```
class Wymierna
{
 friend int suma(Wymierna w);    //
 dostęp do składników niepublicznych
 private:
    int licznik, mianownik;
...
};
int suma(Wymierna w)
{
 return w.licznik + w.mianownik;
}
```

Dane statyczne klasy

```
class Nazwa
{
 static Typ nazwa_danej;
 ...
};
```

Wszystkie egzemplarze (obiekty) klasy mogą mieć wspólne elementy. Zmiana takiego elementu w jednym obiekcie natychmiast dotyczy wszystkich obiektów. Elementy statyczne muszą być zadeklarowane (i mogą być zainicjalizowane) jako dane globalne. Modyfikować je można albo poprzez poszczególne obiekty, albo przez nazwę klasy.

Przykład

```
class Punkt
{
 public:
   static int MAXX, MAXY;
   // elementy statyczne
};
int Punkt::MAXX, MAXY;
// ich deklaracja globalna
...
int main()
{
 Punkt p[100];
 p[0].MAXX = 640;
 // zmiana przez obiekt
cout << p[17].MAXX << endl; // 640
Punkt::MAXY = 480;
// zmiana przez nazwę klasy
cout << p[17].MAXY << endl; // 480
...
```

Częsty błąd

- Brak deklaracji statycznego składnika poza klasą jako danej globalnej.

Funkcje statyczne klasy

```
class Nazwa
{
 static Typ nazwa_funkcji(lista
 argumentów);
 ...
};
```

Funkcja, która operuje wyłącznie na danych statycznych (tzn. odwołuje się do nich), też może być zadeklarowana jako statyczna. Funkcję statyczną można wywoływać zarówno na rzecz klasy, jak i dowolnego jej obiektu.

Przykład

Nawiązanie do poprzedniego:

```
class Punkt
{
   static int MAXX, MAXY;
 public:
   static void zmien_
   rozdzielczosc(int maxx, int maxy){
   MAXX = maxx; MAXY = maxy;}
};
...
int main()
{
 Punkt p[100];
 p[0].zmien_rozdzielczosc(640, 480);
 // wywołanie przez obiekt
 Punkt:: zmien_rozdzielczosc(800,
 600);// i przez nazwę klasy
...
```

Pola bitowe

```
class Nazwa
{
 Typ NazwaPola1 : LiczbaBitów1,
                  ...
     NazwaPolaN : LiczbaBitówN;
};
```

Gęsto upakowane dane, jednak ze zwiększonym kosztem dostępu do nich. Przy projektowaniu pól bitowych musimy znać bitową rozpiętość typu. Do pól bitowych odwolujemy się tak jak do zwykłych danych, jednak nie możemy uzyskiwać wskaźników do nich.

Przykład

```
class Port
{
 private:
  unsigned char in1 : 1,
                in2 : 1,
              clock : 1,
               data : 4;
};
int main()
{
 Port LPT1;
 LPT1.in1 = 1;
...
```

Częste błędy

- Przepełnienie źle zaprojektowanego (za małego) pola bitowego podczas wprowadzania wartości.
- Przepełnienie zmiennej, przeznaczonej podczas projektowania klasy na zbiór pól bitowych.

Wskaźnik this

Wskaźnik do bieżącego egzemplarza klasy (obiektu). Wskaźnik `this` jest intensywnie eksploatowany podczas definiowania operatorów w klasach.

Przykład

```
class Wymierna
{
...
   void wypisz(void)
   { cout << this -> licznik << "/"
   << this -> mianownik;}
};
```

Konstruktory klasy

```
class Nazwa
{
   Nazwa(lista argumentów);
   Nazwa(inna lista argumentów);
...
};
```

Funkcja o nazwie identycznej z nazwą klasy i niezwracająca rezultatu (nawet `void`). Zazwyczaj klasa ma kilka konstruktorów.

Przykład

```
class Punkt
{
 public:
   Punkt(void);
};
```

Uwagi

- Jeśli dla klasy nie zadeklarowano żadnego konstruktora, to system dodaje deklarację konstruktora domyślnego o pustym algorytmie, który wystarcza do deklarowania obiektów (najczęściej o zaśmieconym ustroju).
- Jeśli klasa nie ma konstruktora kopiującego, to system go dodaje.
- Konstruktor konwertujący (jednoargumentowy) może być zadeklarowany z modyfikatorem `explicit` — wtedy nie będzie używany do niejawnych konwersji i będzie traktowany jak konstruktor merytoryczny (porównaj podrozdział „Konwersje typów").
- Szczególnie starannie muszą być napisane konstruktory klas, które dynamicznie przydzielają pamięć.

Częste błędy

- Konstruktor kopiujący zadeklarowany bez argumentu referencyjnego.
- Brak konstruktora kopiującego, gdy klasa dynamicznie operuje na pamięci (brak mechanizmu kopiowania przydzielonych obszarów pamięci).

Lista inicjalizacyjna konstruktorów

Dane należące do klasy najwydajniej inicjalizujemy za pomocą listy. Elementy stałe `const` można zainicjalizować tylko przy użyciu listy. Elementów wspólnych dla wszystkich obiektów (`static`) nie wolno inicjalizować za pomocą listy.

Przykład

```
class Punkt
{
 private:
   int x, y;
   const int kolor;
 public:
   Punkt(int A, int B) : x(A), y(B),
kolor(RED){};
 };
```

Destruktor klasy

```
class Nazwa
{
 public:
   ~Nazwa();
...
};
```

Destruktor jest pozbawioną argumentów i zwracanego rezultatu funkcją o nazwie takiej jak nazwa klasy poprzedzona znakiem tyldy `~`. Jest wywoływany automatycznie lub jawnie, wtedy gdy obiekt typu omawianej klasy jest usuwany z programu. Jeśli klasa nie deklaruje destruktora, dodawany jest destruktor systemowy. Jeśli klasa ma *funkcje wirtualne* (będzie źródłem *polimorficznego* drzewa klas), powinna mieć *wirtualny* destruktor (porównaj podrozdział „Dziedziczenie").

Przykład

```
class Punkt
{
  Punkt(void);
 ~Punkt();
};
...
int main()
{
 Punkt p1, p2[100]; // utworzenie
 obiektów za pomocą konstruktora
 Punkt *adr = new Punkt [33];
...
 delete[] adr; // jawne wywołanie
 33 destruktorów
} // pozostałe 101 destruktorów
  wywołane automatycznie
```

Częsty błąd

- Brak destruktora lub źle napisany destruktor w klasie, której konstruktor dynamicznie przydzielał pamięć.

Przeciążanie operatorów

Egzemplarze klas (obiekty) mogą upodobnić się do danych typów wbudowanych — np. mogą znajdować się w wyrażeniach arytmetycznych.

Przykład

Zakładamy istnienie klasy `LiczbaZespolona` z operatorami przypisania i mnożenia liczby całkowitej przez liczbę zespoloną:

```
LiczbaZespolona z1, z2(1, 1);
z1 = 3 * z2;
```

Zasady przeciążania operatorów

1. Można zupełnie zmieniać sens operatorów.
2. Nie można wymyślać nowych operatorów.
3. Nie można zmieniać liczby argumentów operatorów.
4. Nie można zmieniać priorytetów działania operatorów.

Klasyfikacja konstruktorów na przykładzie klasy `Punkt`.

Nazwa umowna	Postać nagłówka	Komentarz
Domyślny	`Punkt(void);`	Deklaracje bezargumentowe: `Punkt p1;` Deklaracje tablic: `Punkt p[100];` Deklaracje dynamiczne: `Punkt *adr = new Punkt[10];`
Kopiujący	`Punkt(const Punkt &p);`	Deklaracje „na wzór i podobieństwo": `Punkt p1(p2);` Deklaracje z inicjalizowaniem: `Punkt p1 = p2;` Przekaz obiektu do funkcji: `void funkcja(Punkt p);` Zwrot obiektu z funkcji: `Punkt funkcja(void);`
Merytoryczne	`explicit Punkt(int x);` `Punkt(int x, int y);`	Deklaracje z inicjowaniem: `Punkt p1(100), p2(100, 200), *adr = new Punkt(100);`
Konwertujące	`Punkt(InnyTyp A);` `Punkt(const InnyTyp &A);`	Przekształcanie typów, tutaj `InnyTyp` na `Punkt`

Ciąg dalszy na str. 5

Przeciążanie operatorów wewnątrz klasy

```
class Nazwa
{
   ZwracanyTyp operator
SymbolOperatora(lista argumentów);
...
};
```

Operator jest funkcją zadeklarowaną wewnątrz klasy. Wtedy jedynym (lewym) argumentem operatora automatycznie jest obiekt macierzysty (wskaźnik `this`). Operatory deklarowane w klasie mają dostęp do jej prywatnych składników we wszystkich obiektach uczestniczących w algorytmie.

Przykład

Unarny — jednoargumentowy minus:

```
class LiczbaZespolona
{
 public:
   double a, b;
   LiczbaZespolona operator -(void)
   {
      LiczbaZespolona tmp;
      tmp.a = -a; tmp.b = -b;
      return tmp;
   }
};
```

Przykład

Dwuargumentowy operator +:

```
class LiczbaZespolona
{
 public:
   double a, b;
   LiczbaZespolona operator +(const
   LiczbaZespolona &p)
   {
      LiczbaZespolona tmp;
      tmp.a = a + p.a; tmp.b = b +
      p.b;
      return tmp;
   }
};
```

Przykład

Operator przypisania umożliwiający ciąg przypisań `z1 = z2 = z3`:

```
class LiczbaZespolona
{
 public:
   double a, b;
   LiczbaZespolona & operator
   =(const LiczbaZespolona &p)
   {
      if( &p != this)
      {
         a = p.a; b = p.b;
      }
      return *this;
   }
};
```

Globalne przeciążanie operatorów

```
ZwracanyTyp operator
SymbolOperatora(lista argumentów);
...
class Nazwa
{
 ...
};
```

Operator jest funkcją zadeklarowaną poza klasą. Wszystkie argumenty znajdują się na liście argumentów operatora. Operator globalny zawsze ma o jeden argument więcej od swojego odpowiednika deklarowanego w klasie. Operatory globalne nie mają dostępu do prywatnych składników obiektów uczestniczących w algorytmie. Dlatego zazwyczaj klasy deklarują globalne funkcje operatorowe jako *zaprzyjaźnione*, co daje im dostęp do ich prywatnych składników.

Przykład

Przeciążenie operatora `<<`, by wyprowadzał obiekt typu `Wektor3d` w formacie `[1, 2, 3]`:

```
#include <iostream>
using namespace std;
class Wektor3d
{
 friend ostream & operator <<
 (ostream &os, const Wektor3d &w);
 private:
   double x, y, z;
};
ostream & operator << (ostream &os,
const Wektor3d &w)
{
 os << "[" << w.x << ", "  << w.y
 << ", " << w.z << "]";
 return os;
}
int main()
{
 Wektor3d w;
 cout << w;
}
```

DZIEDZICZENIE

```
class Pochodna : rodzaj_dziedziczenia
Bazowa
{
 deklaracje danych i funkcji;
};
```

Jest to mechanizm uzyskiwania nowych klas (zwanych *pochodnymi* lub *potomnymi*) z już istniejących (zwanych *bazowymi*) bez konieczności powtórnego przepisywania kodu. Powstaje tzw. *hierarchia* klas.

Przykład

```
class Monitor_LCD : public Monitor
{
 ...
};
```

Częsty błąd

- Użycie dziedziczenia w sytuacji, gdy typ pochodny *zawiera* typ bazowy, a nie *jest jego rodzajem*.

Rodzaje dziedziczenia i zasięgi

Dziedziczenie publiczne

```
class Pochodna : public Bazowa
{ ...
```

Zasięg elementu w klasie bazowej	Zasięg odziedziczony w klasie pochodnej
private	niedostępne
protected	private
public	public

Wskaźniki i referencje do obiektów klasy pochodnej mogą być traktowane tak, jakby prowadziły do obiektów klasy bazowej (np. na listach argumentów funkcji). Zatem klasa pochodna *jest rodzajem* klasy bazowej.

Dziedziczenie protected

```
class Pochodna : protected Bazowa
{ ...
```

Zasięg elementu w klasie bazowej	Zasięg odziedziczony w klasie pochodnej
private	niedostępne
protected	protected
public	protected

Wskaźniki i referencje do obiektów klasy pochodnej nie mogą być traktowane tak, jakby prowadziły do obiektów klasy bazowej. Zatem klasa pochodna *nie jest rodzajem* klasy bazowej.

Dziedziczenie private

```
class Pochodna : private Bazowa
{ ...
```

Zasięg elementu w klasie bazowej	Zasięg odziedziczony w klasie pochodnej
private	niedostępne
protected	private
public	private

Wskaźniki i referencje do obiektów klasy pochodnej nie mogą być traktowane tak, jakby prowadziły do obiektów klasy bazowej. Zatem klasa pochodna *nie jest rodzajem* klasy bazowej.

Wykluczenia w dziedziczeniu

W klasie pochodnej należy zdefiniować (bo nie podlegają dziedziczeniu):

- konstruktory,
- operator przypisania =,
- destruktor.

W nowych definicjach zawsze należy starać się wykorzystywać algorytmy odpowiednich elementów z klasy bazowej.

Konstruktory klasy pochodnej

```
class Pochodna : rodzaj_dziedziczenia
Bazowa
{
 Pochodna(lista argumentów);
 Pochodna(inna lista argumentów);
};
```

Funkcje nazywające się tak jak klasy i niezwracające rezultatu są konstruktorami (porównaj podrozdział „Klasy"). Najpierw wywołują wskazany konstruktor klasy bazowej, co należy zaznaczyć w nagłówku ich definicji. Nie trzeba wskazywać wywołania konstruktora domyślnego (bezparametrowego) klasy bazowej.

Przykład

```
class Figura
{
 private:
   int kolor;
 public:
   Figura(int Akolor) {kolor =
   Akolor;}
};
class Punkt : public Figura
{
 private:
   int x, y;
 public:
   Punkt(int Ax, int Ay, int Akolor) :
   Figura(Akolor)
   {x = Ax; y = Ay;}
};
```

Konstruktor kopiujący

```
class Pochodna : rodzaj_dziedziczenia
Bazowa
{
 Pochodna(const Pochodna &p);
};
```

Przykład

Nawiązanie do poprzedniego:

```
class Punkt : public Figura
{
 public:
    Punkt(const Punkt & p) :
    Figura(p)
    {x = p.x; y = p.y;}
 ...
```

Operator przypisania w klasie pochodnej

```
class Pochodna : rodzaj_dziedziczenia
Bazowa
{
 Pochodna & operator = (const
 Pochodna &p);
};
```

Pewnym problemem jest wywołanie z klasy bazowej operatora przypisania, który ma dokonać przypisań w części bazowej. Wywołuje się go tak jak zwykłą funkcję ze wskazaniem przynależności — tutaj do klasy bazowej.

Przykład

Nawiązanie do poprzedniego:

```
class Punkt : public Figura
{
 public:
    Punkt & operator=(const Punkt
    & p)
    {
     if( &p != this)
     {
        Figura::operator=(p); x =
        p.x; y = p.y;
     }
     return *this;
    }
 ...
```

Destruktor w klasie pochodnej

```
class Pochodna : rodzaj_dziedziczenia
Bazowa
{
~Pochodna();
};
```

Destruktor jest funkcją niemającą żadnych argumentów, o nazwie takiej jak nazwa klasy poprzedzona znakiem tyldy ~. Klasa pochodna powinna mieć destruktor operujący na własnych elementach (czyli nieodziedziczonych) — w szczególności zwalniający pamięci przydzielone operatorami `new` w klasie pochodnej. Jeśli destruktora nie ma, to zostanie dodany przez system. Najpierw wchodzi do gry destruktor klasy pochodnej, potem bazowej.
Jeśli klasa ma funkcje wirtualne (będzie źródłem *polimorficznego* drzewa klas), powinna mieć wirtualny destruktor.

Funkcje wirtualne

```
class Nazwa
{
    virtual ZwracanyTyp
    NazwaFunkcji(lista argumentów);
...
};
```

Modyfikator `virtual` oznacza te funkcje w klasie bazowej, które mogą (ale nie muszą) zostać zastąpione innymi algorytmami w klasie pochodnej.

Przykład

```
class Figura
{
 public:
    virtual void rysuj(void) { cout
    << "Nie wiadomo...";}
};
class Punkt : public Figura
{
 public:
    void rysuj(void) { cout
    << "Punkt";}
};
class Linia : public Figura
{
 public:
    void rysuj(void) { cout
    << "Linia";}
};
```

Funkcje czysto wirtualne i klasy abstrakcyjne

```
class Nazwa
{
    virtual ZwracanyTyp
    NazwaFunkcji(lista argumentów)
    = 0;
...
};
```

Są to takie funkcje wirtualne, które w klasie bazowej nie mają definicji — są tylko deklaracje przyrównane do zera (symbolika ta nie ma nic wspólnego z arytmetyką). Klasy zawierające funkcje czysto wirtualne nazywają się *abstrakcyjnymi* — nie wolno deklarować obiektów według takich typów i zawsze należy deklarować klasy pochodne definiujące swoje egzemplarze funkcji w miejsce czysto wirtualnych.

Przykład

```
class Figura
{
 public:
    virtual void rysuj(void) = 0;
};
class Kwadrat : public Figura
{
    void rysuj(void) { cout
    << "Kwadrat";};
};
```

Polimorfizm

Jest to mechanizm zastępowania funkcji wirtualnych przez egzemplarze zdefiniowane w klasach pochodnych. Stosujemy go wtedy, kiedy do klas pochodnych odwołujemy się za pomocą *wskaźnika* albo *referencji* do klasy bazowej. Pozwala przy oszczędnym interfejsie (wskaźnik lub referencja *tylko* do klasy bazowej) uzyskać bogatą różnorodność wywoływania odmiennych funkcji. Mechanizm ten nazywa się *późnym wiązaniem* — właściwe funkcje są wyszukiwane podczas pracy programu, a nie w trakcie jego kompilacji.

Przykład

Nawiązanie do poprzednich:

```
void pokaz(Figura &f)
{
 f.rysuj();
}
int main()
{
 Punkt p;
 Linia l;
 pokaz(p);
 pokaz(l);
}
```

Częsty błąd

- Brak wirtualnego destruktora w klasie bazowej.

KONWERSJE TYPÓW

Zagadnienie modyfikowania (dopasowywania) typów nazywa się konwersją.

Przykłady

```
int a = 12e-17;
char c = 1410;
```

Operator konwersji

Nazwa nowego typu — ujęta w nawiasy albo, alternatywnie, nazwa nowego typu i para nawiasów za nią — jest operatorem konwersji, zwanym też operatorem rzutowania.

Przykłady

```
int a = (int)12e-17;
char c = char(12e-17);
```

Operator konwersji jest także wywoływany automatycznie (niejawnie), gdy zachodzi niedopasowanie typów w wyrażeniach, argumentach funkcji lub podczas zwracania rezultatu funkcji.

Zalecenie

- Operatory konwersji mogą być bardzo intensywnie wywoływane niejawnie, zatem muszą być przemyślane.

Tablice informatyczne. C++. Wydanie II

Definiowane operatorów konwersji w klasach

```
class Nazwa
{
 public:
    operator InnyTyp();
...
};
```

Operator konwersji zadeklarowany w klasie przekształca typ tej klasy w inny typ. Klasa może zawierać wiele funkcji tej postaci, czym zapewnia przekształcanie swojego typu na typy wskazane za pomocą rodziny operatorów.

Przykład

```
class Liczba_zespolona
{
 private:
    double a, b;
 public:
    operator double(){ return a * a
    + b * b;}
};
int main()
{
 Liczba_zespolona z;
 cout << (double)z;
 cout << double(z);  // alternatywa
}
```

Konstruktory konwertujące

```
class Nazwa
{
 public:
    Nazwa(InnyTyp dana);
...
};
```

Konwersję w drugim kierunku (od innych typów do typu klasy) zapewniają jednoargumentowe konstruktory (porównaj podrozdział „Konstruktory klasy"). Konstruktory takie tworzą egzemplarz klasy na podstawie typu i wartości argumentu.

Przykład

```
class Liczba_zespolona
{
 private:
    double a, b;
 public:
   Liczba_zespolona(double a) : a(a)
   { b = 0;}
};
int main()
{
 double r = 20;
 Liczba_zespolona z(r);
 // jawna konwersja
 z = r;  // niejawna konwersja
}
```

Konstruktor z zabronioną konwersją niejawną

```
class Nazwa
{
 public:
    explicit Nazwa(InnyTyp dana);
...
};
```

Słowo kluczowe `explicit` powoduje, że konstruktor jednoargumentowy nie może brać udziału w niejawnych konwersjach. Taki konstruktor jest więc traktowany jak zwykły konstruktor merytoryczny (a więc umożliwia deklarowanie obiektów).

Operatory konwersji selektywnej w standardzie C++

Zaleca się stosowanie poniższych konwerterów, które od operatora konwersji różnią się tym, że wybiórczo modyfikują określone cechy konwertowanego typu.

static_cast

```
Typ a = static_cast<Typ>(b);
```

Uruchamia omówione wcześniej mechanizmy konwersji (konstruktor konwertujący lub operator konwersji).

Przykład

```
double r = 20;
Liczba_zespolona z = static_
cast<Liczba_zespolona>(r);
```

const_cast

```
Typ a = const_cast<Typ>(b);
```

Usuwa lub dodaje modyfikator `const` lub `volatile` (porównaj podrozdział „Typy danych"). Najczęściej występuje w wywołaniach funkcji, gdzie argument wchodzi w konflikt z oczekiwanym typem z powodu niezgodności modyfikatorów `const`.

Przykład

```
void wypisz(int &a)
{
 cout << a << endl;
}
int main()
{
 const int a = 10;
wypisz(a);  // błąd
 wypisz(const_cast<int &>(a));
}
```

dynamic_cast

```
Typ a = dynamic_cast<Typ>(b);
```

Wykorzystywany do przekształcania wskaźników albo referencji typów w obrębie polimorficznego drzewa dziedziczenia. Zwraca rezultat, zatem umożliwia testowanie zależności genealogicznych między typami.

Przykład

Klasa `Punkt` jest pochodną klasy `Figura`:

```
Figura *f_adr;
Punkt p;
if(f_adr = dynamic_cast<Figura
*>(&p))  // Uda się konwersja
na klasę bazową?
{
 ...
}
```

reinterpret_cast

```
Typ a = reinterpret_cast<Typ>(b);
```

Ryzykowne przekształcenie wskaźników lub referencji z całkowitą zmianą typów.

Przykład

Przekształcenie wskaźnika obiektu `Punkt` na wskaźnik obiektu `Liczba_zespolona`:

```
Punkt p;
Liczba_zespolona *z_adr;
z_adr = reinterpret_cast<Liczba_
zespolona *>(&p);
```

Deklarowanie i definiowanie wzorca klasy

Deklaracja i definicja szablonu klasy operują symbolicznymi typami — parametrami szablonu.

Przykład

```
template <typename T> class
TypNumeryczny
{
 public:
 T suma(T a, T b) { return a + b;}
 T kosinus(T a) {return (T)
 cos((double)a);}
};
int main()
{
 TypNumeryczny <int> A;
 cout << A.kosinus(5) << A.suma(1, 2);
}
```

Przykład powyższy wymaga, by typ (tutaj `int`) podstawiany w miejsce typu symbolicznego realizował operację dodawania `+`, miał zdefiniowaną konwersję na typ `double` i z typu `double` (porównaj podrozdział „Konwersje typów"), a także aby miał konstruktor kopiujący w celu przekazania argumentów do funkcji i zwrócenia z niej rezultatu.

Wzorzec vector

Biblioteka standardowa dostarcza kilkunastu wzorców klas, zwanych kontenerami. Służą one do manipulowania danymi typu ustalanego w momencie konkretyzacji. Szablon `vector` przechowuje obiekty w dynamicznej tablicy i dostarcza zestawu funkcji do jej obsługi. Typ, którym konkretyzuje się szablon, powinien dysponować publicznym konstruktorem domyślnym (standard c++11 tego nie wymaga), kopiującym, operatorem przypisania i destruktorem.
W tabeli przedstawiono wybrane funkcje kontenera `vector`, demonstrowane na przykładowej konkretyzacji typem `double`.

Przykład

Wypełnienie kontenera przez 1000 wartości pseudolosowych i wyprowadzenie ich na ekran:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
 vector <int> A;
 for(int i = 0; i < 1000; ++i)
    A.push_back(rand());
 for(int i = 0; i < 1000; ++i)
    cout << A[i] << ", ";
}
```

Postać	Komentarz
`vector <double> A;`	Deklaracja pustego kontenera
`vector <double> A(1000);`	Deklaracja tablicy 1000 niezainicjowanych liczb
`vector <double> A(1000, 3.14);`	Deklaracja tablicy 1000 liczb o wartości 3.14
`A.size();`	Aktualna liczba elementów w kontenerze A
`A.clear();`	Opróżnienie kontenera A
`A[123];`	Tablicowy dostęp do elementu nr 123
`A.push_back(3.14);`	Dopisanie elementu o wartości 3.14 na końcu tablicy
`A.pop_back();`	Usunięcie ostatniego elementu (zmniejszenie tablicy)
`A.insert(A.begin()+123, 3.14);`	Dodanie elementu o wartości 3.14 na pozycji 123, licząc od początku (poszerzenie tablicy)
`A.erase(A.begin()+123);`	Usunięcie elementu na pozycji 123, licząc od początku (zmniejszenie tablicy)
`A.resize(2000);`	Przykrojenie lub poszerzenie tablicy

SZABLONY (WZORCE) FUNKCJI I KLAS

```
template <class T1, class T2, ...>
deklaracja_funkcji;
template <class T1, class T2, ...>
deklaracja_klasy;
```

Słowo `class` jest niefortunne — nie ma nic wspólnego z właściwym znaczeniem. Standard C++ wprowadza tutaj lepsze słowo kluczowe:

```
template <typename T1, typename T2,
...> deklaracja;
```

Deklarowanie i definiowanie wzorca funkcji

Wzorzec funkcji to schemat jej nagłówka i algorytmu, zdefiniowany z nieznanymi, symbolicznie oznaczonymi typami, zwanymi parametrami wzorca. W momencie konkretyzacji wzorca (tworzenia prawdziwej funkcji) symboliczne typy zostają zastąpione typami rzeczywistymi. Typy rzeczywiste muszą realizować wszystkie operacje zaimplementowane na typach symbolicznych.

Przykład

Szablon funkcji zwracającej większy element:

```
template <class T> T maksimum(T a,
T b)
{
 return a > b ? a : b;
}
int main()
{
 double x = 1, y = 2;
 cout << maksimum(x, y);
}
```

Powyższy przykład wymaga, by typ (tutaj `double`) podstawiany w miejsce typu symbolicznego realizował operację porównania `>`, a także aby miał konstruktor kopiujący w celu przekazania argumentów do funkcji i zwrócenia z niej rezultatu.

OBSŁUGA SYTUACJI WYJĄTKOWYCH

Zgłaszanie wyjątków

Funkcje zgłaszają wyjątki za pomocą słowa kluczowego `throw` (wyrzuć). Zaleca się (choć nie jest to obowiązkowe) uwidacznianie typu wyjątku w nagłówku funkcji, co podnosi czytelność kodu:

```
typ funkcja(argumenty) throw(typ_
wyjatku)
{
  ...
  if(sytuacja krytyczna)
  {
     typ_wyjatku wyjatek;
     throw wyjatek;
  }
 ...
}
```

Można też zaznaczyć, że funkcja nie wyrzuca żadnego wyjątku:

```
typ funkcja(argumenty) throw()
{ ...
```

Odbieranie sygnałów o wyjątkach

Algorytm, który może wyrzucić wyjątek (zawiera instrukcje `throw`), musi znaleźć się w klamrach `try {...}` (próbuj wykonać). Wystąpienie wyjątku spowoduje przeskok do najbliższej sekcji `catch {...}` (łap, gdy błędy), oznaczonej typem wyjątku zgodnym z typem wyjątku wyrzuconego i argumentem wyjątku:

```
try
{
 ryzykowne instrukcje
}
catch(Typ_wyjatku1 wyjatek1)
{
 instrukcje obsługi wyjątku 1
}
catch(Typ_wyjatku2 wyjatek2)
{
instrukcje obsługi wyjątku 2
...
```

Stosuje się (np. kiedy funkcja zwraca wyjątek jednego typu) nieselektywną obsługę wyjątków, gdy wszystkie wyjątki — niezależnie od ich typów — są kierowane do tej samej sekcji `catch{...}`:

```
catch(...)
{
instrukcje obsługi wszystkich wyjątków
...
```

Przykład

```
double dziel(double a, double b)
throw(int)
{
 if(b == 0)
    throw 17;
 return a / b;
}
int main()
{
 try
 {
    dziel(1, 0);
    cout << "Nie było wyjątku"
    << endl;
 }
 catch(...)
 {
    cout << "Wyjątek - dzielenie
    przez zero!" << endl;
 }
}
```


Wydawnictwo Helion
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl
Informatyka w najlepszym wydaniu

Sprawdź najnowsze promocje: http://helion.pl/promocje
Książki najchętniej czytane: http://helion.pl/bestsellery
Zamów informacje o nowościach: http://helion.pl/nowosci

Aby ocenić tę tablicę, zajrzyj pod adres:
http://helion.pl/user/opinie?ticpp2

helion.pl księgarnia internetowa

ISBN 978-83-246-3925-0