Programowanie, część I

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

6 marca 2012

Kontakt

- http://www.fuw.edu.pl/~rwys/pz/
- rwys@fuw.edu.pl
- tel. 22 55 32 263
- ul. Hoża 69, pok. 142

Materiał na ćwiczenia

- Język C++ – klasy i dziedziczenie.
- Programowanie grafiki z użyciem biblioteki *Qt*.
- Algorytmy numeryczne w C++.

Język C++ – powtórzenie i standardowe klasy

- Podstawowe własności języka C++.
- Proste typy danych, operatory, wskaźniki, referencje.
- Klasy, przeciążanie operatorów, dziedziczenie, modyfikatory dostępu.
- Wirtualne metody, klasy abstrakcyjne i hierarchie klas.
- Szablony.
- Wyjątki.
- Klasy do obsługi wejścia-wyjścia.
- Klasa `string`.
- Biblioteka STL.

Biblioteka *Qt*

- Struktura programu z oknami.
- Wątek obsługi zdarzeń, sygnały i sloty.
- Klasy z biblioteki *Qt* i ich powiązania.
- Tworzenie prostej grafiki.
- Konstruowanie graficznego interfejsu użytkownika (GUI).

Algorytmy numeryczne

- Równania różniczkowe zwyczajne i metoda Rungego-Kutty.
- Równania liniowe – eliminacja Gaussa i Gaussa-Jordana.
- Rozwiązywanie równań nieliniowych metodą Newtona.

Jakim językiem programowania jest C++

Język wysokiego poziomu

Kod źródłowy zapisywany w abstrakcyjnej formie.

Jakim językiem programowania jest C++

Język wysokiego poziomu

Kod źródłowy zapisywany w abstrakcyjnej formie.

Język kompilowany

Kod wykonywalny generowany na podstawie kodu źródłowego przez kompilator (*ang. compiler*).

Jakim językiem programowania jest C++

Język wysokiego poziomu

Kod źródłowy zapisywany w abstrakcyjnej formie.

Język kompilowany

Kod wykonywalny generowany na podstawie kodu źródłowego przez kompilator (*ang. compiler*).

Natywny kod wykonywalny

Kod wykonywalny składa się z rozkazów dla procesora, a zatem nie jest przenośny między różnymi architekturami komputerowymi.

Jakim językiem programowania jest C++

Język wysokiego poziomu

Kod źródłowy zapisywany w abstrakcyjnej formie.

Język kompilowany

Kod wykonywalny generowany na podstawie kodu źródłowego przez kompilator (*ang. compiler*).

Natywny kod wykonywalny

Kod wykonywalny składa się z rozkazów dla procesora, a zatem nie jest przenośny między różnymi architekturami komputerowymi.

Wywodzi się od języka C

Jest nazywany nadzbiorem (*ang. superset*) języka C.

Struktura kodu źródłowego w C++

Musi zawierać co najmniej jendą funkcję (*ang. function*)

Wykonywanie programu zaczyna się od (skompilowanej) funkcji `main()`.

Struktura kodu źródłowego w C++

Musi zawierać co najmniej jendą funkcję (*ang. function*)

Wykonywanie programu zaczyna się od (skompilowanej) funkcji `main()`.

Funkcje w C++

- Każda funkcja składa się z nagłówka (*ang. header*) i treści (*ang. body*).
- Nagłówek określa typ zwracanego wyniku, nazwę i listę argumentów funkcji.
- Treść stanowi (abstrakcyjną) reprezentację pewnego ciągu rozkazów dla procesora.
- Jeżeli treść funkcji reprezentuje obliczenia, to wynik zwracany przez nią powinien reprezentować wynik tych obliczeń.
- Każda funkcja w kodzie źródłowym jest kompilowana oddzielnie.

Przykład funkcji w C++

```
int main(int argc, char *argv[])  // Nagłówek.
{  // Treść.
        QApplication app(argc, argv);

        QLabel hello("Hello world!");
        hello.resize(100, 30);

        hello.show();
        return app.exec();
}
```

Ta funkcja ma nazwę `main`, zwraca wynik typu `int` i ma dwa argumenty, `argc` typu `int` oraz `argv`, który jest tablicą wskaźników (`char` jest typem danych dla zmiennych wskazywanych przez nie).

Instrukcje w C++

Instrukcja (*ang. statememt*)

Najprostszy możliwy do skompilowania zapis w treści funkcji.

Instrukcje w C++

Instrukcja (*ang. statememt*)

Najprostszy możliwy do skompilowania zapis w treści funkcji.

Podstawowe rodzaje instrukcji w C++

1. Przypisanie (*ang. assignment*): symbol =.
2. Modyfikacja: symbole +=, -=, . . . , ++, --.
3. Sterujące:
   - Pętle (`while`, `for`, `do/while`).
   - Przerwanie (`break`) i kontynuacja (`continue`).
   - Skok (`goto`).
4. Warunkowe (`if`, `if/else`) i wyboru (`switch`).
5. Obsługa wyjątków (`try`, `catch`).
6. Wywołanie funkcji (*ang. function call*).

Bloki w C++

W C++ pojedynczą instrukcję można zastąpić blokiem (*ang. block*)

Blok zaczyna się otwierającym nawiasem klamrowym { i kończy się zamykającym nawiasem klamrowym }.

Bloki w C++

W C++ pojedynczą instrukcję można zastąpić blokiem (*ang. block*)

Blok zaczyna się otwierającym nawiasem klamrowym { i kończy się zamykającym nawiasem klamrowym }.

```
for (i = 1; i <= N; i++)
    cout << i << endl;
```

```
for (i = 1; i <= N; i++) {
    suma += i;
    cout << suma << endl;
}
```

Bloki w C++

W C++ pojedynczą instrukcję można zastąpić blokiem (*ang. block*)

Blok zaczyna się otwierającym nawiasem klamrowym { i kończy się zamykającym nawiasem klamrowym }.

```
for (i = 1; i <= N; i++)
    cout << i << endl;
```

```
for (i = 1; i <= N; i++) {
    suma += i;
    cout << suma << endl;
}
```

Poza instrukcjami blok może zawierać deklaracje zmiennych lub stałych.

Zmienne w C++

Zmienna (*ang. variable*)

Symbol reprezentujący pewną wartość, która może zmieniać się w trakcie wykonywania programu przez procesor.

Zmienne w C++

Zmienna (*ang. variable*)

Symbol reprezentujący pewną wartość, która może zmieniać się w trakcie wykonywania programu przez procesor.

C++ jest językiem wymuszającym sztywne reguły stosowania typów danych (*ang. strong typing*).

Zmienne w C++

Zmienna (*ang. variable*)

Symbol reprezentujący pewną wartość, która może zmieniać się w trakcie wykonywania programu przez procesor.

C++ jest językiem wymuszającym sztywne reguły stosowania typów danych (*ang. strong typing*).

Deklaracje zmiennych

Każda zmienna w C++ musi być zadeklarowana zanim zostanie użyta po raz pierwszy. W deklaracji zmiennej należy podać jej nazwę (*ang. name*) oraz typ danych (*ang. data type*).

Zmienne w C++

Zmienna (*ang. variable*)

Symbol reprezentujący pewną wartość, która może zmieniać się w trakcie wykonywania programu przez procesor.

C++ jest językiem wymuszającym sztywne reguły stosowania typów danych (*ang. strong typing*).

Deklaracje zmiennych

Każda zmienna w C++ musi być zadeklarowana zanim zostanie użyta po raz pierwszy. W deklaracji zmiennej należy podać jej nazwę (*ang. name*) oraz typ danych (*ang. data type*).

Typy danych są to zbiory wartości, jakie mogą być reprezentowane przez zmienne (tzn. jakie mogą być nadawane zmiennym).

Fizyczna reprezentacja zmiennych

W czasie wykonywania programu przez procesor zmienne są utożsamiane z określonymi obszarami pamięci (o znanych adresach), zawierającymi wzory bitowe (*ang. bit pattern*) o ustalonej długości i dobrze zdefiniowanej interpretacji.

Fizyczna reprezentacja zmiennych

W czasie wykonywania programu przez procesor zmienne są utożsamiane z określonymi obszarami pamięci (o znanych adresach), zawierającymi wzory bitowe (*ang. bit pattern*) o ustalonej długości i dobrze zdefiniowanej interpretacji.

Długości tych wzorów bitowych (rozmiary zmiennych) oraz ich interpretacja są wyznaczane przez typy danych przypisane tym zmiennym.

Fizyczna reprezentacja zmiennych

W czasie wykonywania programu przez procesor zmienne są utożsamiane z określonymi obszarami pamięci (o znanych adresach), zawierającymi wzory bitowe (*ang. bit pattern*) o ustalonej długości i dobrze zdefiniowanej interpretacji.

Długości tych wzorów bitowych (rozmiary zmiennych) oraz ich interpretacja są wyznaczane przez typy danych przypisane tym zmiennym.

Każda wartość należąca do typu danych przypisanego zmiennej musi być reprezentowana przez jeden z takich wzorów bitowych (niektóre wartości mogą być reprezentowane przez dwa wzory bitowe lub większą ich liczbę).

Typy danych w C++

W C++ każda wartość będąca wynikiem obliczeń lub wykorzystywana w obliczeniach zawsze ma określony typ danych.

Typy danych w C++

W C++ każda wartość będąca wynikiem obliczeń lub wykorzystywana w obliczeniach zawsze ma określony typ danych.

W C++ obliczenia zawsze są przeprowadzane na wartościach tego samego typu. Mogą one pochodzić od wartości innych typów, z których są otrzymywane na drodze rzutowania typów danych (*ang. type casting*).

Typy danych w C++

W C++ każda wartość będąca wynikiem obliczeń lub wykorzystywana w obliczeniach zawsze ma określony typ danych.

W C++ obliczenia zawsze są przeprowadzane na wartościach tego samego typu. Mogą one pochodzić od wartości innych typów, z których są otrzymywane na drodze rzutowania typów danych (*ang. type casting*).

Proste i złożone typy danych

W C++ zdefiniowana jest pewna liczba prostych lub podstawowych (*ang. fundamental*) typów danych, z których można konstruować (dowolnie skomplikowane) złożone typy danych.

Proste typy danych w C++

Podstawowe

1. Numeryczne całkowite:
   - Bezznakowe (`unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`).
   - Ze znakiem (`char`, `short`, `int`, `long`, `long long`).
2. Numeryczne zmiennoprzecikowe (`float`, `double`, `long double`).
3. Boole'owski (`bool`).
4. Poszerzony znakowy (`wchar_t`) (*ang. wide character*).

Wskaźnikowe (*ang. pointer*)

- Czyste adresy (`void *`).
- Związane z typami podstawowymi (np. `(char *)`, `(int *)` itd.).
- Związane z typami złożonymi.

Automatyczne rzutowanie typów danych – numeryczne

1. `char`
2. `unsigned char`
3. `short int`
4. `unsigned short int`
5. `int`
6. `unsigned int`
7. `long int`
8. `unsigned long int`
9. `long long int`
10. `unsigned long long int`
11. `float`
12. `double`
13. `long double`

Rzutowanie typów danych na `bool`

Zasada 0 = `false` dla numerycznych typów danych

Wartość typu `bool` zawsze może być zastąpiona przez wartość o numerycznym typie danych, która jest automatycznie rzutowana na typ `bool`. Wynikiem tego rzutowania jest wartość `false`, jeżeli rzutowaną numeryczną wartością jest 0. W przeciwnym wypadku wynikiem tego rzutowania jest wartość `true`.

Rzutowanie typów danych na bool

Zasada 0 = false dla numerycznych typów danych

Wartość typu bool zawsze może być zastąpiona przez wartość o numerycznym typie danych, która jest automatycznie rzutowana na typ bool. Wynikiem tego rzutowania jest wartość false, jeżeli rzutowaną numeryczną wartością jest 0. W przeciwnym wypadku wynikiem tego rzutowania jest wartość true.

Zasada NULL = false dla wskaźników

Jest analogiczna do powyższej, tylko wynikiem rzutowania jest true dla wszystkich wartości wskaźnikowych oprócz NULL (czyli „zera").

Wyrażenia w C++

Wyrażenia są fragmentami kodu źródłowego reprezentującymi obliczenia.

Wyrażenia w C++

Wyrażenia są fragmentami kodu źródłowego reprezentującymi obliczenia.

Na ich podstawie kompilator tworzy ciągi rozkazów dla procesora, po wykonaniu których otrzymuje się określone wyniki (odpowiadające wynikom wyrażeń zapisanych w kodzie źródłowym).

Wyrażenia w C++

Wyrażenia są fragmentami kodu źródłowego reprezentującymi obliczenia.

Na ich podstawie kompilator tworzy ciągi rozkazów dla procesora, po wykonaniu których otrzymuje się określone wyniki (odpowiadające wynikom wyrażeń zapisanych w kodzie źródłowym).

W wyrażeniach występują wartości, na podstawie których obliczany jest wynik oraz operatory, czyli symbole reprezentujące czynności do wykonania.

Wyrażenia w C++

Wyrażenia są fragmentami kodu źródłowego reprezentującymi obliczenia.

Na ich podstawie kompilator tworzy ciągi rozkazów dla procesora, po wykonaniu których otrzymuje się określone wyniki (odpowiadające wynikom wyrażeń zapisanych w kodzie źródłowym).

W wyrażeniach występują wartości, na podstawie których obliczany jest wynik oraz operatory, czyli symbole reprezentujące czynności do wykonania.

Wynik wyrażenia zawsze jest określonego typu, który zależy od typów wartości, na podstawie których został on obliczony.

Operatory w C++

1. Arytmetyczne (+, -, *, /, %).
2. Bitowe (*ang. bit pattern*) (|, &, ^, ~, <<, >>).
3. Porównania (==, !=, <, >, <=, >=).
4. Logiczne (*ang. boolean*) (||, &&, !).
5. Operator trójargumentowy (? :).
6. Wskaźnikowe (*ang. pointer*) (+, -, *, &, ->, []).
7. Wyboru składnika złożonego typu danych (.).
8. Inkrementacji i dekrementacji (++, --).
9. Rezerwowania i zwalniania pamięci (`new`, `delete`).
10. `sizeof()`.
11. Przypisania (=) i modyfikacji (+=, -=, *=, ...).

Priorytety operatorów

Priorytet (*ang. priority*) operatora

Waga określająca w jakiej kolejności w stosunku do innych operacji w wyrażeniu będzie przeprowadzona operacja reprezentowana przez dany operator.

Priorytety operatorów

Priorytet (*ang. priority*) operatora

Waga określająca w jakiej kolejności w stosunku do innych operacji w wyrażeniu będzie przeprowadzona operacja reprezentowana przez dany operator.

Zasady

1. W pierwszej kolejności przeprowadzane są operacje reprezentowane przez operatory o najwyższych priorytetach.
2. Dla operatorów o jednakowych priorytetach kolejność przeprowadzania operacji nie jest jednoznacznie określona (najlepiej jest używać nawiasów w celu określenia jej).
3. Do wymuszania określonej kolejności operacji służą nawiasy okrągłe.

Tabela priorytetów (siły wiązania) operatorów

Siła wiązania (priorytet) operatorów w porządku malejącym

```
  !     ~     -
  *     /     %
  +     -
 <<    >>
  <     <=    >=    >
 ==    !=
  &
  ^
  |
 &&
 ||
 ?  :
```

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Na ogół wygodnie jest używać funkcji (procedur) do logicznej separacji części kodu przeznaczonych do różnych zadań.

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Na ogół wygodnie jest używać funkcji (procedur) do logicznej separacji części kodu przeznaczonych do różnych zadań.

Metody w klasach są specjalnym rodzajem funkcji.

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Na ogół wygodnie jest używać funkcji (procedur) do logicznej separacji części kodu przeznaczonych do różnych zadań.

Metody w klasach są specjalnym rodzajem funkcji.

Aby funkcje mogły spełniać swoje zadania, muszą operować na określonych danych, które trzeba przekazywać do kodu zapisanego w postaci funkcji.

Przykład – dodawanie macierzy 2x2

```
class Matrix22 {
  public:
    double w1k1, w1k2, w2k1, w2k2;
    Matrix22(void): w1k1(0), w1k2(0), w2k1(0), w2k2(0) {}
    Matrix22(double r): w1k1(r), w1k2(0), w2k1(0), w2k2(r) {}
    Matrix22(double a, double b, double c, double d):
             w1k1(a), w1k2(b), w2k1(c), w2k2(d) {}
    double det(void) const { return w1k1*w2k2 - w1k2*w2k1; }
    Matrix22 operator +(Mattix22 m)
    {
        return Matrix22(w1k1 + m.w1k1, w1k2 + m.w1k2,
                        w2k1 + m.w2k1, w2k2 + m.w2k2);
    }
};

int main()
{
    Matrix22 A(1), B(2), C;

    C = A + B;
    cout << C.w1k1 << " " << C.w1k2 << " " << C.w2k1 << " " << C.w2k2 << endl;
    A = C + B;
    cout << A.w1k1 << " " << A.w1k2 << " " << A.w2k1 << " " << A.w2k2 << endl;
    B = A + C;
    cout << B.w1k1 << " " << B.w1k2 << " " << B.w2k1 << " " << B.w2k2 << endl;
    return 0;
}
```

Przekazywanie argumentu przez wartość

Aby uniknąć wielokrotnego powtarzania tego samego kodu wprowadzamy funkcję `drukuj()`:

```
void drukuj(Matrix22 M)
{
    cout << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}

int main()
{
    Matrix22 A(1), B(2), C;

    C = A + B;
    drukuj(C);
    A = C + B;
    drukuj(A);
    B = A + C;
    drukuj(B);
    return 0;
}
```

Przekazywanie argumentu przez wartość

Aby uniknąć wielokrotnego powtarzania tego samego kodu wprowadzamy funkcję `drukuj()`:

```
void drukuj(Matrix22 M)
{
    cout << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}

int main()
{
    Matrix22 A(1), B(2), C;

    C = A + B;
    drukuj(C);
    A = C + B;
    drukuj(A);
    B = A + C;
    drukuj(B);
    return 0;
}
```

Przy każdym wywołaniu `drukuj()` tworzona jest kopia obiektu przekazywanego jako argument (używana w funkcji pod nazwą `M`).

Wady przekazywania argumentów funkcji przez wartość

Każdym wywołanie funkcji `drukuj()` wymaga

1. zarezerwowania pamięci na 4 zmienne typu `double`,
2. przeprowadzenia operacji kopiowania 4 wartości typu `double` (poświęcamy czas).

Ponosimy zatem koszty związane z użyciem funkcji.

Wady przekazywania argumentów funkcji przez wartość

Każdym wywołanie funkcji `drukuj()` wymaga

1. zarezerwowania pamięci na 4 zmienne typu `double`,
2. przeprowadzenia operacji kopiowania 4 wartości typu `double` (poświęcamy czas).

Ponosimy zatem koszty związane z użyciem funkcji.

Koszty te nie byłyby ponoszone, gdyby można było „powiedzieć” funkcji, że ma posługiwać się tym samym obiektem, który jest przekazywany jako argument (bez kopiowania go).

Wady przekazywania argumentów funkcji przez wartość

Każdym wywołanie funkcji `drukuj()` wymaga

1. zarezerwowania pamięci na 4 zmienne typu `double`,
2. przeprowadzenia operacji kopiowania 4 wartości typu `double` (poświęcamy czas).

Ponosimy zatem koszty związane z użyciem funkcji.

Koszty te nie byłyby ponoszone, gdyby można było „powiedzieć" funkcji, że ma posługiwać się tym samym obiektem, który jest przekazywany jako argument (bez kopiowania go).

Do tego celu służą referencje (*ang. reference*).

Przekazywanie argumentu przez referencję

Aby uniknąć kopiowania obiektu przekazywanego jako argument do funkcji można użyć referencji:

```
void drukuj(Matrix22& M)
{
    cout << M.w1k1 << " " << M.w1k2 << " "
         << M.w2k1 << " " << M.w2k2 << endl;
}
```

Przekazywanie argumentu przez referencję

Aby uniknąć kopiowania obiektu przekazywanego jako argument do funkcji można użyć referencji:

```
void drukuj(Matrix22& M)
{
    cout << M.w1k1 << " " << M.w1k2 << " "
         << M.w2k1 << " " << M.w2k2 << endl;
}
```

Wtedy funkcja będzie używać tego samego obiektu, który został przekazany jako argument (bez kopiowania go), ale pod inną nazwą.

Przekazywanie argumentu przez referencję

Aby uniknąć kopiowania obiektu przekazywanego jako argument do funkcji można użyć referencji:

```
void drukuj(Matrix22& M)
{
    cout << M.w1k1 << " " << M.w1k2 << " "
         << M.w2k1 << " " << M.w2k2 << endl;
}
```

Wtedy funkcja będzie używać tego samego obiektu, który został przekazany jako argument (bez kopiowania go), ale pod inną nazwą.

Wówczas modyfikacje tego obiektu przez funkcję zostaną zachowane i będą „widoczne” po zakończeniu wykonywania jej.

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Coś takiego oznacza, że funkcja przekazuje kontrolę nad obiektem, którym posługiwała się, do funkcji wywołującej ją.

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Coś takiego oznacza, że funkcja przekazuje kontrolę nad obiektem, którym posługiwała się, do funkcji wywołującej ją.

Ma to sens jedynie w przypadku obiektów, o których wiadomo, że będą istniały po zakończeniu wykonywania funkcji zwracających referencje do nich (tzn. nie wolno zwracać referencji do obiektów będących zmiennymi lokalnymi w funkcji zwracającej referencję do obiektu).

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Coś takiego oznacza, że funkcja przekazuje kontrolę nad obiektem, którym posługiwała się, do funkcji wywołującej ją.

Ma to sens jedynie w przypadku obiektów, o których wiadomo, że będą istniały po zakończeniu wykonywania funkcji zwracających referencje do nich (tzn. nie wolno zwracać referencji do obiektów będących zmiennymi lokalnymi w funkcji zwracającej referencję do obiektu).

```
ostream& drukuj(ostream& out, Matrix22& M)
{
    out << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
    return out;  // Przekazanie kontroli nad (obiektem reprezentowanym przez) out z powrotem.
}
```

Zwracanie referencji przez funkcje (c. d.)

Zapis funkcji `drukuj()` z użyciem referencji pozwala na wykonanie następującej operacji bez kopiowania obiektów (najpierw drukowane są elementy macierzowe `A`, później – `B`, a na końcu – `C`):

```
drukuj(drukuj(drukuj(cout, A), B), C);
```

Zwracanie referencji przez funkcje (c. d.)

Zapis funkcji `drukuj()` z użyciem referencji pozwala na wykonanie następującej operacji bez kopiowania obiektów (najpierw drukowane są elementy macierzowe `A`, później – `B`, a na końcu – `C`):

```
drukuj(drukuj(drukuj(cout, A), B), C);
```

Można zapisać `drukuj()` w postaci operatora <<:

```
ostream& operator <<(ostream& out, Matrix22& M)
{
    return out << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}
```

Zwracanie referencji przez funkcje (c. d.)

Zapis funkcji `drukuj()` z użyciem referencji pozwala na wykonanie następującej operacji bez kopiowania obiektów (najpierw drukowane są elementy macierzowe `A`, później – `B`, a na końcu – `C`):

```
drukuj(drukuj(drukuj(cout, A), B), C);
```

Można zapisać `drukuj()` w postaci operatora <<:

```
ostream& operator <<(ostream& out, Matrix22& M)
{
    return out << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}
```

Wtedy drukowanie elementów macierzowych `A`, `B` i `C` (kolejno) można zapisać jako:

```
cout << A << B << C;
```

Kiedy referencje sprawiają problemy

Funkcja `operator <<()` z poprzedniego slajdu nie nadaje się do przeprowadzenia następującej operacji:

```
cout << (A + B);
```

Kiedy referencje sprawiają problemy

Funkcja `operator <<()` z poprzedniego slajdu nie nadaje się do przeprowadzenia następującej operacji:

```
cout << (A + B);
```

Jest tak dlatego, że metoda `operator +()` z klasy `Matrix22` zwraca wynik przez wartość i wymaga skopiowania go do jakiegoś istniejącego obiektu (tzn. obiekt tworzony podczas wykonywania instrukcji `return` w tej metodzie istnieje tylko do czasu skopiowania go do innego obiektu).

Kiedy referencje sprawiają problemy

Funkcja `operator <<()` z poprzedniego slajdu nie nadaje się do przeprowadzenia następującej operacji:

```
cout << (A + B);
```

Jest tak dlatego, że metoda `operator +()` z klasy `Matrix22` zwraca wynik przez wartość i wymaga skopiowania go do jakiegoś istniejącego obiektu (tzn. obiekt tworzony podczas wykonywania instrukcji `return` w tej metodzie istnieje tylko do czasu skopiowania go do innego obiektu).

Rozwiązanie tego problemu wymaga przekazywania `M` do funkcji `operator <<()` przez wartość:

```
ostream& operator <<(ostream& out, Matrix22 M)
{
    return out << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}
```

Do czego są potrzebne wskaźniki (w C++)

W C++ wskaźniki (tzn. zmienne zawierające adresy innych zmiennych, np. obiektów) są *głównie* potrzebne do operowania zmiennymi tworzonymi na żądanie (dynamicznie) z pomocą `new`.

Do czego są potrzebne wskaźniki (w C++)

W C++ wskaźniki (tzn. zmienne zawierające adresy innych zmiennych, np. obiektów) są *głównie* potrzebne do operowania zmiennymi tworzonymi na żądanie (dynamicznie) z pomocą `new`.

Zmienne tworzone na żądanie (z pomocą `new`) są „anonimowe” (tzn. nie mają nazw), więc ich położenie w pamięci jest znane tylko dzięki adresom przechowywanym we wskaźnikach.

Do czego są potrzebne wskaźniki (w C++)

W C++ wskaźniki (tzn. zmienne zawierające adresy innych zmiennych, np. obiektów) są *głównie* potrzebne do operowania zmiennymi tworzonymi na żądanie (dynamicznie) z pomocą `new`.

Zmienne tworzone na żądanie (z pomocą `new`) są „anonimowe” (tzn. nie mają nazw), więc ich położenie w pamięci jest znane tylko dzięki adresom przechowywanym we wskaźnikach.

Przypomnienie

Wynik zwracany przez `new` jest adresem zmiennej (lub tablicy) utworzonej na żądanie (dynamicznie) lub ma wartość `NULL` (czyli 0), jeżeli zmienna nie mogła być utworzona.

Wskazywane zmienne

Jeżeli `wsk` jest wskaźnikiem, to `*wsk` oznacza referencję do zmiennej, której adres jest wartością `wsk`. Nazywana się ją zmienną wskazywaną przez `wsk` lub zmienną pod adresem `wsk`.

Wskazywane zmienne

Jeżeli `wsk` jest wskaźnikiem, to `*wsk` oznacza referencję do zmiennej, której adres jest wartością `wsk`. Nazywana się ją zmienną wskazywaną przez `wsk` lub zmienną pod adresem `wsk`.

Jeżeli `wsk` jest wskaźnikiem, to `*wsk` może zastępować nazwę zmiennej we wszystkich sytuacjach, w których może ona być użyta, a w szczególności:

- po lewej stronie operatorów przypisania i modyfikacji,
- przy przekazywaniu argumentu do funkcji przez referencję,
- przy zwracaniu referencji przez funkcję.

Wskazywane zmienne

Jeżeli `wsk` jest wskaźnikiem, to `*wsk` oznacza referencję do zmiennej, której adres jest wartością `wsk`. Nazywana się ją zmienną wskazywaną przez `wsk` lub zmienną pod adresem `wsk`.

Jeżeli `wsk` jest wskaźnikiem, to `*wsk` może zastępować nazwę zmiennej we wszystkich sytuacjach, w których może ona być użyta, a w szczególności:

- po lewej stronie operatorów przypisania i modyfikacji,
- przy przekazywaniu argumentu do funkcji przez referencję,
- przy zwracaniu referencji przez funkcję.

Operator obliczania adresu &

Dla zmiennej o nazwie `var` symbol `&var` oznacza adres tej zmiennej. Zatem `*(&var)` oznacza to samo, co `var`.

Wskaźnikowe typy danych

Wszystkie wskaźniki mają jednakowe rozmiary, ale przypisuje się im typy danych określające typy danych dla wskazywanych zmiennych.

Wskaźnikowe typy danych

Wszystkie wskaźniki mają jednakowe rozmiary, ale przypisuje się im typy danych określające typy danych dla wskazywanych zmiennych.

Nazwa wskaźnikowego typu danych składa się z nazwy typu danych dla wskazywanych zmiennych i symblu * (rozdzielnych znakami przerwy), np.:

`(int *)` – wskazywane zmienne są typu `int`.

`(double *)` – wskazywane zmienne są typu `double`.

`(Matrix22 *)` – wskazywane zmienne są obiektami klasy `Matrix22`.

Wskaźnikowe typy danych

Wszystkie wskaźniki mają jednakowe rozmiary, ale przypisuje się im typy danych określające typy danych dla wskazywanych zmiennych.

Nazwa wskaźnikowego typu danych składa się z nazwy typu danych dla wskazywanych zmiennych i symblu * (rozdzielnych znakami przerwy), np.:

`(int *)` – wskazywane zmienne są typu `int`.

`(double *)` – wskazywane zmienne są typu `double`.

`(Matrix22 *)` – wskazywane zmienne są obiektami klasy `Matrix22`.

W deklaracjach wskaźników nazwy odpowiadających im typów danych występują bez nawiasów, np.:

```
int *wsk;  // Wskaźnik zawierający adresy zmiennych typu int.
Matrix22 *ptr;  // Wskaźnik zawierający adresy obiektów klasy Matrix22.
```

Wskaźnikowe typy danych i wskazywane zmienne

Dzięki wskaźnikowym typom danych wiadomo, co oznacza `*wsk` dla danego wskaźnika `wsk`.

Wskaźnikowe typy danych i wskazywane zmienne

Dzięki wskaźnikowym typom danych wiadomo, co oznacza `*wsk` dla danego wskaźnika `wsk`.

`*wsk`

Zmienna typu określonego przez typ danych wskaźnika `wsk` znajdująca się pod adresem będącym wartością `wsk`.

Wskaźnikowe typy danych i wskazywane zmienne

Dzięki wskaźnikowym typom danych wiadomo, co oznacza `*wsk` dla danego wskaźnika `wsk`.

`*wsk`

Zmienna typu określonego przez typ danych wskaźnika `wsk` znajdująca się pod adresem będącym wartością `wsk`.

Operator ->

Jeżeli zmienna wskazywana przez wskaźnik `wsk` jest obiektem (tzn. jej typ danych jest klasą), to operator -> pozwala na odwoływanie się do pól i wywoływanie metod w kontekście tej zmiennej, np.:

```
Matrix22 *ptr;  // Wskaźnik zawierający adresy obiektów klasy Matrix22.

ptr = new Matrix22(1);
ptr->w1k2 = -1;  // Odwołanie do pola w1k2 obiektu wskazywanego przez ptr.
cout << ptr->det();  // Wywołanie metody det() dla obiektu wskazywanego przez ptr.
```

Stałe wskaźnikowe i czyste adresy

Stałe wskaźnikowe

Dla dowolnej zmiennej `var` wynik wyrażenia `&var` jest zawsze taki sam przez cały czas „życia” zmiennej, a zatem jest on stałą wskaźnikową. Podobnie stałą wskaźnikową jest nazwa tablicy jednowymiarowej (reprezentuje ona adres pierwszego elementu tablicy).

Stałe wskaźnikowe i czyste adresy

Stałe wskaźnikowe

Dla dowolnej zmiennej `var` wynik wyrażenia `&var` jest zawsze taki sam przez cały czas „życia" zmiennej, a zatem jest on stałą wskaźnikową. Podobnie stałą wskaźnikową jest nazwa tablicy jednowymiarowej (reprezentuje ona adres pierwszego elementu tablicy).

Czysty adres

Adres miejsca w pamięci nie skojarzony z typem danych zmiennej.

Stałe wskaźnikowe i czyste adresy

Stałe wskaźnikowe

Dla dowolnej zmiennej `var` wynik wyrażenia `&var` jest zawsze taki sam przez cały czas „życia" zmiennej, a zatem jest on stałą wskaźnikową. Podobnie stałą wskaźnikową jest nazwa tablicy jednowymiarowej (reprezentuje ona adres pierwszego elementu tablicy).

Czysty adres

Adres miejsca w pamięci nie skojarzony z typem danych zmiennej.

`(void *)`

Wskaźnikowy typ danych reprezentujący czyste adresy. Dla wskaźników tego typu operacje `*wsk` i `wsk->` nie mają sensu.

Wskaźniki i funkcje

Wskaźniki mogą być argumentami funkcji, np.:

```
void drukuj(Matrix22 *M)
{
    cout << M->w1k1 << " " << M->w1k2 << " " << M->w2k1 << " " << M->w2k2 << endl;
}
```

Wskaźniki i funkcje

Wskaźniki mogą być argumentami funkcji, np.:

```
void drukuj(Matrix22 *M)
{
    cout << M->w1k1 << " " << M->w1k2 << " " << M->w2k1 << " " << M->w2k2 << endl;
}
```

W ten sposób unikamy kopiowania zmiennych przy przekazywaniu ich do funkcji (oszczędność czasu i miejca, podobnie jak przy posługiwaniu się referencjami).

Wskaźniki i funkcje

Wskaźniki mogą być argumentami funkcji, np.:

```
void drukuj(Matrix22 *M)
{
    cout << M->w1k1 << " " << M->w1k2 << " " << M->w2k1 << " " << M->w2k2 << endl;
}
```

W ten sposób unikamy kopiowania zmiennych przy przekazywaniu ich do funkcji (oszczędność czasu i miejca, podobnie jak przy posługiwaniu się referencjami).

Jednak w tym celu musimy znać adres zmiennej, którą funkcja ma posługiwać się lub obliczyć go, np.:

```
Matrix22 *wsk;

wsk = new Matrix22(1);
drukuj(wsk);  // Adres zmiennej jest wartością wsk.
```

```
Matrix22 M(1);

drukuj(&M);  // Jawnie obliczamy adres M.
```

Zgodność typów danych dla wskaźników

Kompilator C++ ostrzega w przypadkach, w których wartość wskaźnikowa jest przypisywana wskaźnikowi o niewłaściwym typie danych, np.:

```
double *wsk;

wsk = double[100];
drukuj(wsk);  // UWAGA! Oczekiwanym typem danych jest (Matrix22 *)!
```

Zgodność typów danych dla wskaźników

Kompilator C++ ostrzega w przypadkach, w których wartość wskaźnikowa jest przypisywana wskaźnikowi o niewłaściwym typie danych, np.:

```
double *wsk;

wsk = double[100];
drukuj(wsk);  // UWAGA! Oczekiwanym typem danych jest (Matrix22 *)!
```

Można zastosować jawną konwersję typów danych w celu wskazania kompilatorowi, że wiemy co robimy, np.:

```
drukuj((Matrix22 *)wsk);
```

Zgodność typów danych dla wskaźników

Kompilator C++ ostrzega w przypadkach, w których wartość wskaźnikowa jest przypisywana wskaźnikowi o niewłaściwym typie danych, np.:

```
double *wsk;

wsk = double[100];
drukuj(wsk);  // UWAGA! Oczekiwanym typem danych jest (Matrix22 *)!
```

Można zastosować jawną konwersję typów danych w celu wskazania kompilatorowi, że wiemy co robimy, np.:

```
drukuj((Matrix22 *)wsk);
```

Jawna konwersja wskaźnikowych typów danych nie jest potrzebna, jeżeli docelowy wskaźnik jest typu `(void *)` lub przypisujemy czysty adres wskaźnikowi dowolnego typu.

Wskaźniki i referencje – różnice

Wskaźniki są zmiennymi, a referencje nie mogą nimi być.

Wskaźniki i referencje – różnice

Wskaźniki są zmiennymi, a referencje nie mogą nimi być.

Użycie referencji oznacza przekazanie kontroli nad obiektem, natomiast wskaźnik określa położenie zmiennej (lub ogólnie czegoś) w pamięci.

Wskaźniki i referencje – różnice

Wskaźniki są zmiennymi, a referencje nie mogą nimi być.

Użycie referencji oznacza przekazanie kontroli nad obiektem, natomiast wskaźnik określa położenie zmiennej (lub ogólnie czegoś) w pamięci.

Dlatego pola obiektów mogą być wskaźnikami (tzn. można stworzyć np. klasę, w której część pól będzie wskaźnikami), ale nie mogą być referencjami.

Rola i rodzaje złożonych typów danych

Pozwalają reprezentować złożone obiekty

Pozwalają tworzyć zmienne, którymi można posługiwać się tak, jak gdyby były zespołami zmiennych o różnych prostych typach danych.

Rola i rodzaje złożonych typów danych

Pozwalają reprezentować złożone obiekty

Pozwalają tworzyć zmienne, którymi można posługiwać się tak, jak gdyby były zespołami zmiennych o różnych prostych typach danych.

Zmienne o złożonych typach danych często nazywane są obiektami (*ang. object*).

Rola i rodzaje złożonych typów danych

Pozwalają reprezentować złożone obiekty

Pozwalają tworzyć zmienne, którymi można posługiwać się tak, jak gdyby były zespołami zmiennych o różnych prostych typach danych.

Zmienne o złożonych typach danych często nazywane są obiektami (*ang. object*).

Rodzaje złożonych typów danych w C++

- Struktury (*ang. structure*)
- Unie (*ang. union*)
- Klasy (*ang. class*)

Struktury

Definicja struktury

1. Słowo kluczowe `struct`.
2. Nazwa struktury (identyfikator złożonego typu danych).
3. Otwierający nawias klamrowy.
4. Lista składników (*ang. member*) – dla struktur są to wyłącznie pola (*ang. field*).
5. Zamykający nawias klamrowy.
6. (Opcjonalnie) lista nazw zmiennych (o tym typie danych).
7. Średnik.

Struktury

Definicja struktury

1. Słowo kluczowe `struct`.
2. Nazwa struktury (identyfikator złożonego typu danych).
3. Otwierający nawias klamrowy.
4. Lista składników (*ang. member*) – dla struktur są to wyłącznie pola (*ang. field*).
5. Zamykający nawias klamrowy.
6. (Opcjonalnie) lista nazw zmiennych (o tym typie danych).
7. Średnik.

Składniki struktury deklaruje się tak, jak zmienne (bez wartości początkowych).

Zmienne o typach danych będących strukturami

Deklaracja zmiennej zawiera słowo kluczowe `struct` i nazwę struktury, które łącznie pełnią rolę typu danych dla zmiennej, np.:

```
struct element zmienna;
struct element *wsk;  // Wskaźnik
```

Zmienne o typach danych będących strukturami

Deklaracja zmiennej zawiera słowo kluczowe `struct` i nazwę struktury, które łącznie pełnią rolę typu danych dla zmiennej, np.:

```
struct element zmienna;
struct element *wsk;  // Wskaźnik
```

Wykorzystanie pamięci

1. Zmienna o typie danych będącym strukturą zajmuje w pamięci co najmniej tyle miejsca, ile wynosi suma rozmiarów jej składników.
2. Składniki takiej zmiennej na ogół są rozmieszczane w pamięci zgodnie z kolejnością deklaracji i mogą być wyrównywane na granicy słowa o określonej długości (np. 32-bitowego).
3. Dwa składniki zmiennej mogą zajmować jedno słowo wspólnie.
4. `sizeof()` oblicza liczbę bajtów danych zajmowaną przez strukturę.

Bezpośrednie odwołania do pól struktury

Kropka oddziela nazwę zmiennej o złożonym typie danych od nazwy składnika, do którego odnosi się operacja, np.:

```
struct wektor {
    double x;
    double y;
    double z;
} w;

w.x = 1;
cout << w.x << endl;
```

Bezpośrednie odwołania do pól struktury

Kropka oddziela nazwę zmiennej o złożonym typie danych od nazwy składnika, do którego odnosi się operacja, np.:

```
struct wektor {
    double x;
    double y;
    double z;
} w;

w.x = 1;
cout << w.x << endl;
```

`w.x` pełni rolę zmiennej typu `double`.

Odwołania do pól struktury przez wskaźnik

Symbol -> oznacza, że operacja ma odnosić się do pola o danej nazwie, będącego składnikiem zmiennej (o złożonym typie danych) wskazywanej przez wskaźnik, np.:

```
struct wektor {
    double x;
    double y;
    double z;
} w, *wsk;

wsk = &w;
wsk->x = 1;
cout << wsk->x << endl;
```

Odwołania do pól struktury przez wskaźnik

Symbol -> oznacza, że operacja ma odnosić się do pola o danej nazwie, będącego składnikiem zmiennej (o złożonym typie danych) wskazywanej przez wskaźnik, np.:

```
struct wektor {
    double x;
    double y;
    double z;
} w, *wsk;

wsk = &w;
wsk->x = 1;
cout << wsk->x << endl;
```

`wsk->x` pełni rolę zmiennej typu `double`.

Początkowe wartości pól struktury

```
struct wektor {
    double x;
    double y;
    double z;
};

struct wektor w1 = { 1, 2, 3 };
struct wektor w2 = { .y = 0, .x = 1, .z = 2 };
struct wektor w3 = { .x = 1, .y = 2, };
```

Początkowe wartości pól struktury

```
struct wektor {
    double x;
    double y;
    double z;
};

struct wektor w1 = { 1, 2, 3 };
struct wektor w2 = { .y = 0, .x = 1, .z = 2 };
struct wektor w3 = { .x = 1, .y = 2, };
```

Pole z zmiennej w3 nie ma określonej wartości początkowej.

Definiowanie i używanie unii

1. Definicja unii wygląda tak samo, jak definicja struktury o identycznych polach, tylko zamiast słowa kluczowego `struct` używa się słowa kluczowego `union`.
2. Różnica polega na tym, że wszystkie pola unii znajdują się pod tym samym adresem w pamięci.
3. Stąd wynika, że pól unii używa się zamiennie.
4. Odwołania do pól unii zapisuje się tak samo, jak odwołania do pól struktury.

Definiowanie i używanie unii

1. Definicja unii wygląda tak samo, jak definicja struktury o identycznych polach, tylko zamiast słowa kluczowego `struct` używa się słowa kluczowego `union`.
2. Różnica polega na tym, że wszystkie pola unii znajdują się pod tym samym adresem w pamięci.
3. Stąd wynika, że pól unii używa się zamiennie.
4. Odwołania do pól unii zapisuje się tak samo, jak odwołania do pól struktury.

Przykład

```
union bajty {
    int n;
    unsigned char b[sizeof(int)];
};
```

Przykład użycia unii

Drukowanie wartości bajtów składowych dla wartości typu int

```
union bajty {
    int n;
    unsigned char b[sizeof(int)];
};

union bajty x;

cout << "Podaj n: ";
cin >> x.n;

for (int i = 0; i < sizeof(int); i++)
    cout << "b[" << i << "] = 0x" << hex << (int)x.b[i] << endl;
```

Definiowanie klas

1. Słowo kluczowe `class`.
2. Nazwa klasy (litery, znak _ i cyfry – oprócz pierwszego znaku).
3. Specyfikacja klas nadrzędnych (od których pochodzi ta klasa).
4. Lista składników (w nawiasie klamrowym).
   - Pola – typ danych i nazwa (jak dla struktur).
   - Metody – nagłówek (typ wyniku, nazwa, lista argumentów).
   - Dla każdego składnika można określić zasady dostępu.

Definiowanie klas

1. Słowo kluczowe `class`.
2. Nazwa klasy (litery, znak _ i cyfry – oprócz pierwszego znaku).
3. Specyfikacja klas nadrzędnych (od których pochodzi ta klasa).
4. Lista składników (w nawiasie klamrowym).
   - Pola – typ danych i nazwa (jak dla struktur).
   - Metody – nagłówek (typ wyniku, nazwa, lista argumentów).
   - Dla każdego składnika można określić zasady dostępu.

```
class Vector {
  public:  // specyfikacja zasad dostępu
        double x, y;          // pola
        double norm(void);    // metoda
        double length(void); // metoda
};
```

Modyfikatory dostępu

Modyfikator dostępu (*ang. access modifier*)

Określa, z jakich miejsc w programie można odwoływać się do składników klasy („oddziałuje" na składniki zadeklarowane „pod" nim).

`public` – do tych składników klasy można odwoływać się z dowolnego miejsca w programie.

`private` – do tych składników klasy można odwoływać się tylko z metod będących składnikami tej klasy.

`protected` – do tych składników klasy można odwoływać się z metod będących składnikami tej klasy oraz klas pochodnych w stosunku do niej.

Modyfikatory dostępu

Modyfikator dostępu (*ang. access modifier*)

Określa, z jakich miejsc w programie można odwoływać się do składników klasy („oddziałuje" na składniki zadeklarowane „pod" nim).

`public` – do tych składników klasy można odwoływać się z dowolnego miejsca w programie.

`private` – do tych składników klasy można odwoływać się tylko z metod będących składnikami tej klasy.

`protected` – do tych składników klasy można odwoływać się z metod będących składnikami tej klasy oraz klas pochodnych w stosunku do niej.

Domyślnym modyfikatorem dostępu jest `private`.

Metody

Metody definiuje się podobnie jak zwykłe funkcje

Przed nazwą metody trzeba umieścić nazwę klasy, której składnikiem jest ta metoda oraz symbol `::`, np.:

```
double Wektor::length(void)
{
    // Tu znajduje się treść metody.
}
```

Metody

Metody definiuje się podobnie jak zwykłe funkcje

Przed nazwą metody trzeba umieścić nazwę klasy, której składnikiem jest ta metoda oraz symbol `::`, np.:

```
double Wektor::length(void)
{
    // Tu znajduje się treść metody.
}
```

Metodę zawsze wywołuje się dla konkretnego obiektu i jego składniki (pola oraz inne metody) mogą być używane w treści metody jak zwykłe zmienne (w przypadku pól) lub funkcje (w przypadku metod).

Odwoływanie się do składników klasy

Do składników klasy można odwoływać się tylko poprzez obiekty tej klasy, z pomocą symboli . i ->.

```
Klasa obiekt;

obiekt.pole = wart;
obiekt.metoda(wart);
```

```
Klasa *wsk;

wsk = new Klasa;
wsk->pole = wart;
wsk->metoda(wart);
```

Kontekst odwołania do składnika obiektu

Zmienna po lewej stronie symbolu . lub wskaźnik po lewej stronie symbolu -> określa kontekst odwołania do pola obiektu lub wywołania metody.

Kontekst odwołania do składnika obiektu

Zmienna po lewej stronie symbolu . lub wskaźnik po lewej stronie symbolu -> określa kontekst odwołania do pola obiektu lub wywołania metody.

```
class Vector {
 public:
        double x, y;
        double norm(void);
        double length(void);
};

double Vector::norm(void)
{
        return x*x + y*y;
}

double Vector::length(void)
{
        return sqrt(norm());
}
```

```
Vector wk, *ptr;

wk.x = 0;
wk.y = 1;
// Metoda norm() będzie wywołana w
// metodzie length() dla pól z obiektu wk
cout << wk.length() << endl;

ptr = new Vector;
ptr->x = 1;
ptr->y = 1;
// Metoda norm() będzie wywołana w
// metodzie length() dla pól z obiektu
// pod adresem ptr
cout << ptr->length() << endl;
```

Implementacja metod

Definicja klasy określa tylko nagłówki metod. Aby można było z nich korzystać, trzeba podać ich implementację (sposób działania).

Implementacja metody jest bardzo podobna do definicji funkcji.

Implementacja metod

Definicja klasy określa tylko nagłówki metod. Aby można było z nich korzystać, trzeba podać ich implementację (sposób działania).

Implementacja metody jest bardzo podobna do definicji funkcji.

```
double Vector::norm(void)
{
        return x*x + y*y;
}

double Vector::length(void)
{
        return sqrt(norm());
}
```

Metody „inline”

Jeżeli implementacja metody jest szczególnie prosta, można umieścić ją wewnątrz definicji klasy.

Metody „inline”

Jeżeli implementacja metody jest szczególnie prosta, można umieścić ją wewnątrz definicji klasy.

```
class Vector {
  public:
        double x, y;
        double norm(void)
        {
                return x*x + y*y;
        }
        double length(void)
        {
                return sqrt(norm());
        }
};
```

Metody „const”

Jeżeli pola, z których korzysta metoda, nie są przez nią zmieniane, można to zadeklarować umieszczając `const` w nagłówku.

Metody „const”

Jeżeli pola, z których korzysta metoda, nie są przez nią zmieniane, można to zadeklarować umieszczając `const` w nagłówku.

```
class Vector {
  public:
        double x, y;
        double norm(void) const
        {
                return x*x + y*y;
        }
        double length(void) const
        {
                return sqrt(norm());
        }
};
```

Statyczne pola

Pola będące składnikami klas można zadeklarować z atrybutem `static`.

Wówczas są one wspólne dla wszystkich obiektów danej klasy i są inicjowane przed rozpoczęciem wykonywania funkcji `main()`.

Statyczne pola

Pola będące składnikami klas można zadeklarować z atrybutem `static`.

Wówczas są one wspólne dla wszystkich obiektów danej klasy i są inicjowane przed rozpoczęciem wykonywania funkcji `main()`.

```
class Klasa {
  public:
        static int statyczne_pole;  // pole wspólne dla wszystkich obiektów tej klasy
        ...
};

...
Klasa a, b;

...
a.statyczne_pole = 123;
cout << b.statyczne_pole << endl; // wydrukuje 123
cout << Klasa::statyczne_pole << endl; // wydrukuje 123
```

Stałe skojarzone z klasami

W definicji klasy można umieścić definicję stałej.

```
class Klasa {
  public:
        static const double jeden = 1;  // stała
        ...
};
```

Stałe skojarzone z klasami

W definicji klasy można umieścić definicję stałej.

```
class Klasa {
  public:
        static const double jeden = 1;  // stała
        ...
};
```

Dalej można odwoływać się do takiej stałej łącząc nazwę klasy z jej nazwą.

```
cout << Klasa::jeden << endl;
```

Konstruktor

Jeśli w czasie tworzenia obiektu danej klasy należy wykonać jakąś akcję (np. zarezerwować pamięć), to jednym z jej składników powinien być konstruktor (*ang. constructor*).

Konstruktor

Jeśli w czasie tworzenia obiektu danej klasy należy wykonać jakąś akcję (np. zarezerwować pamięć), to jednym z jej składników powinien być konstruktor (*ang. constructor*).

Konstruktor

Metoda wywoływana automatycznie (z odpowiednimi argumentami) podczas tworzenia obiektu. Zawsze ma taką nazwę, jak klasa, której jest składnikiem.

Konstruktor

Jeśli w czasie tworzenia obiektu danej klasy należy wykonać jakąś akcję (np. zarezerwować pamięć), to jednym z jej składników powinien być konstruktor (*ang. constructor*).

Konstruktor

Metoda wywoływana automatycznie (z odpowiednimi argumentami) podczas tworzenia obiektu. Zawsze ma taką nazwę, jak klasa, której jest składnikiem.

```
class Tablica {
  public:
        double *elem;
        int n;
        double& operator [](int n);
        Tablica(int n_el);
};
```

```
Tablica::Tablica(int n_el)
{
        elem = new double[n_el];
        if (elem)
                n = n_el;
}

Tablica tab(10);  // wywołanie konstruktora
```

Definiowanie wielu konstruktorów

W jednej klasie może być wiele konstruktorów, ale muszą one różnić się jednoznacznie liczbą lub typami danych argumentów.

Definiowanie wielu konstruktorów

W jednej klasie może być wiele konstruktorów, ale muszą one różnić się jednoznacznie liczbą lub typami danych argumentów.

```
class Tablica {
  public:
        double *elem;
        int n;
        ...
        void init(int n_el);
        Tablica(int n_el);
        Tablica(int n_el, double r);
};

void Tablica::init(int n_el)
{
        elem = new double[n_el];
        if (elem)
                n = n_el;
}
```

```
Tablica::Tablica(int n_el)
{
        init(n_el);
}

Tablica::Tablica(int n_el, double r)
{
        init(n_el);
        for (int i = 0; i < n; i++)
                elem[i] = r;
}

...
Tablica tab(10);  // Tablica(int)

...
Tablica tmp(10, -1);  // Tablica(int, double)
```

Kiedy wywoływane są konstruktory

Dla zmiennych lokalnych

Bezpośrednio po utworzeniu zmiennej w czasie wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu definicji zmiennej w obrębie bloku).

Kiedy wywoływane są konstruktory

Dla zmiennych lokalnych

Bezpośrednio po utworzeniu zmiennej w czasie wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu definicji zmiennej w obrębie bloku).

Dla zmiennych dynamicznych

Bezpośrednio po utworzeniu zmiennej z użyciem `new`, np.:

```
wsk = new Tablica(10);
```

Kiedy wywoływane są konstruktory

Dla zmiennych lokalnych

Bezpośrednio po utworzeniu zmiennej w czasie wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu definicji zmiennej w obrębie bloku).

Dla zmiennych dynamicznych

Bezpośrednio po utworzeniu zmiennej z użyciem `new`, np.:

```
wsk = new Tablica(10);
```

Dla zmiennych globalnych lub statycznych

Przed rozpoczęciem wykonywania funkcji `main()`, bezpośrednio po zarezerwowaniu pamięci na te zmienne.

Kopiowanie argumentów konstruktora

Kopiowanie argumentów konstruktora do pól obiektu można zapisywać w skrótowej formie `pole1(argument1), pole2(argument2), ...` po nagłówku konstruktora i znaku `:` (przed nawiasem klamrowym rozpoczynającym właściwą treść konstruktora).

Kopiowanie argumentów konstruktora

Kopiowanie argumentów konstruktora do pól obiektu można zapisywać w skrótowej formie `pole1(argument1), pole2(argument2), ...` po nagłówku konstruktora i znaku `:` (przed nawiasem klamrowym rozpoczynającym właściwą treść konstruktora).

```
class Wektor {
  public:
        double x, y;
        Wektor(void): x(0), y(0) {}
        Wektor(double a, double b): x(a), y(b) {}
        double norma(void) const
        {
                return x*x + y*y;
        }
        ...
};
```

Destruktor

Jeżeli przed usunięciem obiektu z pamięci trzeba przeprowadzić jakąś dodatkową czynność (np. zwolnić pamięć zarezerwowaną przez konstruktor), to jednym ze składników klasy powinien być destruktor (*ang. destructor*).

Destruktor

Jeżeli przed usunięciem obiektu z pamięci trzeba przeprowadzić jakąś dodatkową czynność (np. zwolnić pamięć zarezerwowaną przez konstruktor), to jednym ze składników klasy powinien być destruktor (*ang. destructor*).

Destruktor

Metoda wykonywana automatycznie bezpośrednio przed usunięciem obiektu z pamięci. Jej nazwa musi składać się ze znaku ~ i nazwy klasy, której jest składnikiem.

Destruktor

Jeżeli przed usunięciem obiektu z pamięci trzeba przeprowadzić jakąś dodatkową czynność (np. zwolnić pamięć zarezerwowaną przez konstruktor), to jednym ze składników klasy powinien być destruktor (*ang. destructor*).

Destruktor

Metoda wykonywana automatycznie bezpośrednio przed usunięciem obiektu z pamięci. Jej nazwa musi składać się ze znaku ~ i nazwy klasy, której jest składnikiem.

Do destruktora nie można przekazywać argumentów, więc w każdej klasie może być co najwyżej jeden destruktor.

Definiowanie destruktora

```
class Tablica {
  public:
        double *elem;
        int n;
        ...
        void init(int n_el);
        Tablica(int n_el)
        {
                init(n_el);
        }
        Tablica(int n_el, double r);
        ~Tablica(void);
};

void Tablica::init(int n_el)
{
        elem = new double[n_el];
        if (elem)
                n = n_el;
}
```

```
Tablica::Tablica(int n_el, double r)
{
        init(n_el);
        for (int i = 0; i < n; i++)
                elem[i] = r;
}

Tablica::~Tablica(void)
{
        n = 0;
        if (elem)
                delete [] elem;
}

...
Tablica *wsk;

...
wsk = new Tablica(10, 0);
...
delete wsk;  // ~Tablica()
```

Kiedy wywoływany jest destruktor

Dla zmiennych lokalnych

Bezpośrednio przed usunięciem zmiennej z pamięci po zakończeniu wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu klamry } kończącej blok).

Kiedy wywoływany jest destruktor

Dla zmiennych lokalnych

Bezpośrednio przed usunięciem zmiennej z pamięci po zakończeniu wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu klamry } kończącej blok).

Dla zmiennych dynamicznych

Bezpośrednio przed usunięciem zmiennej z użyciem `delete`.

Kiedy wywoływany jest destruktor

Dla zmiennych lokalnych

Bezpośrednio przed usunięciem zmiennej z pamięci po zakończeniu wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu klamry } kończącej blok).

Dla zmiennych dynamicznych

Bezpośrednio przed usunięciem zmiennej z użyciem `delete`.

Dla zmiennych globalnych lub statycznych

Po zakończeniu wykonywania funkcji `main()`, bezpośrednio przed zwolnieniem pamięci zajmowanej przez te zmienne.

Na czym polega przeciążanie operatorów

Zamiast podawać nazwę metody lub funkcji, można ją oznaczyć z pomocą symbolu operatora. Wtedy liczba jej argumentów zależy od liczby argumentów tego operatora.

Na czym polega przeciążanie operatorów

Zamiast podawać nazwę metody lub funkcji, można ją oznaczyć z pomocą symbolu operatora. Wtedy liczba jej argumentów zależy od liczby argumentów tego operatora.

```
class Wektor {
  public:
        double x, y;
        void add(Wektor w);
};

void Wektor::add(Wektor w)
{
        x += w.x;
        y += w.y;
}
...

Wektor w, v;

...
w.add(v);
```

Na czym polega przeciążanie operatorów

Zamiast podawać nazwę metody lub funkcji, można ją oznaczyć z pomocą symbolu operatora. Wtedy liczba jej argumentów zależy od liczby argumentów tego operatora.

```
class Wektor {
  public:
        double x, y;
        void add(Wektor w);
};

void Wektor::add(Wektor w)
{
        x += w.x;
        y += w.y;
}
...

Wektor w, v;

...
w.add(v);
```

```
class Wektor {
  public:
        double x, y;
        void operator +=(Wektor w);
};

void Wektor::operator +=(Wektor w)
{
        x += w.x;
        y += w.y;
}
...

Wektor w, v;

...
w += v;  // wywołanie metody operator +=
```

Przeciążanie operatorów – argumenty

Argumenty funkcji wywoływanej jako operator nie muszą być tego samego typu jak obiekt, w kontekście którego wywoływana jest metoda.

Przeciążanie operatorów – argumenty

Argumenty funkcji wywoływanej jako operator nie muszą być tego samego typu jak obiekt, w kontekście którego wywoływana jest metoda.

```
class Wektor {
  public:
        double x, y;
        void operator *=(double r);
};

void Wektor::operator *=(double r)
{
        x *= r;
        y *= r;
}
...

Wektor w;

...
w *= 2;  // wywołanie metody operator *=
```

Przeciążanie operatorów – argumenty

Argumenty funkcji wywoływanej jako operator nie muszą być tego samego typu jak obiekt, w kontekście którego wywoływana jest metoda.

```
class Wektor {
  public:
        double x, y;
        void operator *=(double r);
};

void Wektor::operator *=(double r)
{
        x *= r;
        y *= r;
}
...

Wektor w;

...
w *= 2;  // wywołanie metody operator *=
```

```
class Wektor {
  public:
        double x, y;
        void operator +=(double r);
};

void Wektor::operator +=(double r)
{
        x += r;
}
...

Wektor w;

...
w += 1;  // wywołanie metody operator +=
```

Przeciążanie symbolu []

Można także przeciążać operator wskazania elementu tablicy [].
Reprezentująca go funkcja najczęściej zwraca referencję, aby można było użyć jej po lewej stronie operator przypisania =.

Przeciążanie symbolu []

Można także przeciążać operator wskazania elementu tablicy []. Reprezentująca go funkcja najczęściej zwraca referencję, aby można było użyć jej po lewej stronie operator przypisania =.

```
class Wektor {
  public:
        double x, y;
        double& wsp(int n)
        {
                return n == 1 ? x : y;
        }
};

...

Wektor w;

...
w.wsp(1) = 2;  // wywołanie metody wsp()
```

Przeciążanie symbolu []

Można także przeciążać operator wskazania elementu tablicy []. Reprezentująca go funkcja najczęściej zwraca referencję, aby można było użyć jej po lewej stronie operator przypisania =.

```
class Wektor {
  public:
        double x, y;
        double& wsp(int n)
        {
                return n == 1 ? x : y;
        }
};

...

Wektor w;

...
w.wsp(1) = 2;  // wywołanie metody wsp()
```

```
class Wektor {
  public:
        double x, y;
        double& operator [](int n)
        {
                return n == 1 ? x : y;
        }
};

...

Wektor w;

...
w[1] = 2;  // wywołanie metody operator []
```

Wskaźnik `this`

Każdy obiekt dowolnej klasy zawiera pole o nazwie `this`, które jest wskaźnikiem, a jego wartością jest adres tego obiektu.

Wskaźnik `this`

Każdy obiekt dowolnej klasy zawiera pole o nazwie `this`, które jest wskaźnikiem, a jego wartością jest adres tego obiektu.

Wskaźnik `this` jest typu `(Klasa *)`, gdzie `Klasa` jest typem danych (klasą) odpowiadającą obiektowi, w skład którego on wchodzi.

Wskaźnik `this`

Każdy obiekt dowolnej klasy zawiera pole o nazwie `this`, które jest wskaźnikiem, a jego wartością jest adres tego obiektu.

Wskaźnik `this` jest typu `(Klasa *)`, gdzie `Klasa` jest typem danych (klasą) odpowiadającą obiektowi, w skład którego on wchodzi.

Wskaźnik `this` jest polem prywatnym, więc może być wykorzystywany tylko przez metody będące składnikami klasy obiektu.

Wskaźnik `this`

Każdy obiekt dowolnej klasy zawiera pole o nazwie `this`, które jest wskaźnikiem, a jego wartością jest adres tego obiektu.

Wskaźnik `this` jest typu `(Klasa *)`, gdzie `Klasa` jest typem danych (klasą) odpowiadającą obiektowi, w skład którego on wchodzi.

Wskaźnik `this` jest polem prywatnym, więc może być wykorzystywany tylko przez metody będące składnikami klasy obiektu.

Obiekt, w którego skład wchodzi pole `this`, jest wartością wyrażenia `*this`.

Do czego przydaje się `this`?

Wskaźnik `this` przydaje się (między innymi) przy przeciążaniu operatora preinkrementacji.

Do czego przydaje się this?

Wskaźnik this przydaje się (między innymi) przy przeciążaniu operatora preinkrementacji.

```
class Wektor {
        double x, y;
  public:
        Wektor(void): x(0), y(0) {}
        Wektor(double a, double b): x(a), y(b) {}
        double norma(void) const
        {
                return x*x + y*y;
        }
        Wektor operator ++(void)
        {
                x++;
                y++;
                return *this;
        }
};
```

```
Wektor w(1, 1), v;

v = ++w;

cout << w.norma() << " " << v.norma() << endl;
```

Przeciążanie postinkrementacji

W celu przeciążenia postinkrementacji trzeba zdefiniować metodę z jednym argumentem typu `int`.

Przeciążanie postinkrementacji

W celu przeciążenia postinkrementacji trzeba zdefiniować metodę z jednym argumentem typu `int`.

```
class Wektor {
        double x, y;
  public:
        Wektor(void): x(0), y(0) {}
        Wektor(double a, double b): x(a), y(b) {}
        double norma(void) const
        {
                return x*x + y*y;
        }
        Wektor operator ++(int zero)
        {
                Wektor w(x, y);

                x++;
                y++;
                return w;
        }
};
```

```
Wektor w(1, 1), v;

v = w++; // v = w.operator++(0)

cout << w.norma() << " " << v.norma() << endl;
```

Ostrożnie z przypisaniem!

Domyślna operacja przypisania (kopiowanie wartości pól z jednego obiektu do drugiego) nie musi działać zgodnie z oczekiwaniami.

Ostrożnie z przypisaniem!

Domyślna operacja przypisania (kopiowanie wartości pól z jednego obiektu do drugiego) nie musi działać zgodnie z oczekiwaniami.

```
class Tablica {
        double *elem;  // prywatne!
        int n;         // prywatne!
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        double& operator [](int i)
        {
                return elem[i];
        }
};

Tablica::Tablica(int nr_el)
{
        elem = new double[nr_el];
        if (elem)
                n = nr_el;
}
```

```
Tablica::~Tablica(void)
{
        n = 0;
        if (elem)
                delete [] elem;
}

...

Tablica a(2), b(3);

for (int i = 0; i < 3; i++)
        b[i] = 0;
a = b;

// Dlaczego wykonanie tego nie spowoduje błędu?
cout << a[0] << " " << a[1] << " " << a[2] << endl;
```

Przeciążanie przypisania

Rozwiązaniem problemu jest przeciążenie operatora przypisania.

Przeciążanie przypisania

Rozwiązaniem problemu jest przeciążenie operatora przypisania.

```
class Tablica {
        double *elem;
        int n;
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        double& operator [](int i)
        {
                return elem[i];
        }
        void operator =(Tablica& t);
};

Tablica::Tablica(int nr_el)
{
        elem = new double[nr_el];
        if (elem)
                n = nr_el;
}
```

```
Tablica::~Tablica(void)
{
        n = 0;
        if (elem)
                delete [] elem;
}

void Tablica::operator =(Tablica& t)
{
        if (n == t.n)
                for (int i = 0; i < n; i++)
                        elem[i] = t.elem[i];
}

...
Tablica a(2), b(2);

...
a = b;  // a.operator=(b)
```

Przeciążanie przypisania (inny typ argumentu)

Argument metody wywoływanej jako operator = może być dowolny.

Przeciążanie przypisania (inny typ argumentu)

Argument metody wywoływanej jako operator = może być dowolny.

```
class Tablica {
        double *elem;
        int n;
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        double& operator [](int i)
        {
                return elem[i];
        }
        void operator =(double r);
};

Tablica::Tablica(int nr_el)
{
        elem = new double[nr_el];
        if (elem)
                n = nr_el;
}
```

```
Tablica::~Tablica(void)
{
        n = 0;
        if (elem)
                delete [] elem;
}

void Tablica::operator =(double r)
{
          for (int i = 0; i < n; i++)
                elem[i] = r;
}

...
Tablica a(2);

...
a = 0;  // a.operator=(r)
```

Przeciążone przypisanie zwracające wynik

„Normalne” przypisanie zwraca wynik, który można wykorzystać w wyrażeniu. Podonie może być z przeciążonym przypisaniem.

Przeciążone przypisanie zwracające wynik

„Normalne” przypisanie zwraca wynik, który można wykorzystać w wyrażeniu. Podonie może być z przeciążonym przypisaniem.

```
class Tablica {
        double *elem;
        int n;
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        double& operator [](int i)
        {
                return elem[i];
        }
        Tablica& operator =(Tablica& t);
};

Tablica::Tablica(int nr_el)
{
        elem = new double[nr_el];
        if (elem)
                n = nr_el;
}
```

```
Tablica::~Tablica(void)
{
        n = 0;
        if (elem)
                delete [] elem;
}

Tablica& Tablica::operator =(Tablica& t)
{
        if (n == t.n)
                for (int i = 0; i < n; i++)
                        elem[i] = t.elem[i];
        return *this;
}

...
Tablica a(2), b(2), c(2);

...
a = b = c;  // a.operator=(b.operator=(c))
```

Różne operatory o tym samym symbolu

W jednej klasie może być zdefiniowanych wiele metod wywoływanych jako operator o tym samym symbolu (np. +=).

Różne operatory o tym samym symbolu

W jednej klasie może być zdefiniowanych wiele metod wywoływanych jako operator o tym samym symbolu (np. +=).

Muszą one różnić się typem argumentów tak, aby kompilator mógł jednoznacznie stwierdzić która z nich ma być wywołana w danym miejscu programu.

Różne operatory o tym samym symbolu

W jednej klasie może być zdefiniowanych wiele metod wywoływanych jako operator o tym samym symbolu (np. +=).

Muszą one różnić się typem argumentów tak, aby kompilator mógł jednoznacznie stwierdzić która z nich ma być wywołana w danym miejscu programu.

Operatory mogą być również przeciążane z pomocą funkcji, które nie są metodami.

Różne operatory o tym samym symbolu

W jednej klasie może być zdefiniowanych wiele metod wywoływanych jako operator o tym samym symbolu (np. +=).

Muszą one różnić się typem argumentów tak, aby kompilator mógł jednoznacznie stwierdzić która z nich ma być wywołana w danym miejscu programu.

Operatory mogą być również przeciążane z pomocą funkcji, które nie są metodami.

Funkcje te mogą reprezentować operatory o takich symbolach, jakie zostały już przeciążone z pomocą metod.

Przeciążanie dodawania

W celu przeciążenia dodawania wygodnie jest użyć funkcji nie będącej metodą.

Przeciążanie dodawania

W celu przeciążenia dodawania wygodnie jest użyć funkcji nie będącej metodą.

```
class Wektor {
  public:
        double x, y;
        Wektor(void): x(0), y(0) {}
        Wektor(double a, double b): x(a), y(b) {}
        double norma(void) const
        {
                return x*x + y*y;
        }
};
```

```
Wektor operator +(Wektor w, Wektor v)
{
        Wektor wynik;

        wynik.x = w.x + v.x;
        wynik.y = w.y + v.y;
        return wynik;
}

...
Wektor a(1, 1), b(1, 0), c;

...
c = a + b;  // operator +(a, b)
```

Przeciążanie operacji konwersji typów danych

W C++ konwersja typów danych, jak np. `(double)zmienna`, może być przeciążona podobnie do operatorów.

Przeciążanie operacji konwersji typów danych

W C++ konwersja typów danych, jak np. `(double)zmienna`, może być przeciążona podobnie do operatorów.

W celu przeciążenia operacji konwersji typów danych w klasie, dla której chcemy zmienić działanie tej operacji, definiuje się metodę o nazwie `operator typ()`, gdzie `typ` jest typem danych, na który będą „konwertowane" obiekty danej klasy, np. `operator double()`.

Przeciążanie operacji konwersji typów danych

W C++ konwersja typów danych, jak np. `(double)zmienna`, może być przeciążona podobnie do operatorów.

W celu przeciążenia operacji konwersji typów danych w klasie, dla której chcemy zmienić działanie tej operacji, definiuje się metodę o nazwie `operator typ()`, gdzie `typ` jest typem danych, na który będą „konwertowane" obiekty danej klasy, np. `operator double()`.

W definicji metody `operator double()` oraz analogicznych metod dla innych typów danych nie deklaruje się typu zwracanego wyniku i argumentów (są one określone domyślnie).

Przeciążanie konwersji typów danych – przykład

Metoda `operator double()` określa znaczenie zapisu `(double)w` oraz jest wykorzystywana przy niejawnych konwersjach (np. nadaje sens wyrażeniu `w == 25`).

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

class Wektor {
  public:
    double x, y;
    Wektor(void): x(0), y(0) {}
    Wektor(double a, double b): x(a), y(b) {}
    operator double() const { return x*x + y*y; }
};

int main()
{
    Wektor w(3, 4);

    cout << (double)w << endl;  // w.operator double()
    if (w == 25)  // w.operator double() == 25
        cout << "OK" << endl;

    return 0;
}
```

Funkcje zaprzyjaźnione z klasami

Jeżeli pola danej klasy są prywatne, ale funkcja (np. przeciążająca operator) ma mieć do nich dostęp, to można zadeklarować ją jako „przyjaciela” (*ang. friend*) tej klasy.

Funkcje zaprzyjaźnione z klasami

Jeżeli pola danej klasy są prywatne, ale funkcja (np. przeciążająca operator) ma mieć do nich dostęp, to można zadeklarować ją jako „przyjaciela” (*ang. friend*) tej klasy.

```
class Wektor {
        double x, y;
  public:
        Wektor(void): x(0), y(0) {}
        Wektor(double a, double b): x(a), y(b) {}
        double norma(void) const
        {
                return x*x + y*y;
        }
        friend Wektor operator +(Wektor w, Wektor v);
};
```

Klasy zaprzyjaźnione z innymi klasami

Jeżeli pola danej klasy (np. A) są prywatne, ale metody z innej klasy (np. B) mają mieć do nich dostęp, to można zadeklarować klasę B jako „przyjaciela" (*ang. friend*) klasy A.

Klasy zaprzyjaźnione z innymi klasami

Jeżeli pola danej klasy (np. A) są prywatne, ale metody z innej klasy (np. B) mają mieć do nich dostęp, to można zadeklarować klasę B jako „przyjaciela" (*ang. friend*) klasy A.

```
class A {
        double a;
  public:
        A(void): x(0) {}
        A(double x): a(x) {}
        ...
        friend class B;  // Metody z B mają dostęp do pola a z A
};
```

Klasy nadrzędne i podrzędne

Jedna klasa jest podrzędna (*ang. secondary*) w stosunku do drugiej lub jest jej podklasą (*ang. subclass*), lub jest klasą pochodną (*ang. derived class*) w stosunku do niej, jeżeli dziedziczy (*ang. inherit*) po niej część składników (tzn. część jej składników pochodzi z tamtej klasy).

Klasy nadrzędne i podrzędne

Jedna klasa jest podrzędna (*ang. secondary*) w stosunku do drugiej lub jest jej podklasą (*ang. subclass*), lub jest klasą pochodną (*ang. derived class*) w stosunku do niej, jeżeli dziedziczy (*ang. inherit*) po niej część składników (tzn. część jej składników pochodzi z tamtej klasy).

Klasa, po której składniki są dziedziczone, nazywana jest klasą nadrzędną (*ang. primary*) w stosunku do klasy dziedziczącej lub jej nadklasą (*ang. superclass*), lub klasą macierzystą (*ang. parent class*) albo klasą bazową (*ang. base class*) w stosunku do niej.

Klasy nadrzędne i podrzędne

Jedna klasa jest podrzędna (*ang. secondary*) w stosunku do drugiej lub jest jej podklasą (*ang. subclass*), lub jest klasą pochodną (*ang. derived class*) w stosunku do niej, jeżeli dziedziczy (*ang. inherit*) po niej część składników (tzn. część jej składników pochodzi z tamtej klasy).

Klasa, po której składniki są dziedziczone, nazywana jest klasą nadrzędną (*ang. primary*) w stosunku do klasy dziedziczącej lub jej nadklasą (*ang. superclass*), lub klasą macierzystą (*ang. parent class*) albo klasą bazową (*ang. base class*) w stosunku do niej.

Podklasy można traktować jako bardziej precyzyjne specyfikacje rzeczy mających wspólne własności (w takim sensie „jamnik" jest bardziej precyzyjnym okerśleniem własności zwierzęcia, niż „pies").

Definiowanie klas pochodnych

W C++ w definicji klasy można umieścić listę klas nadrzędnych w stosunku do niej (z modyfikatorami dostępu).

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void);
        double abs(void);
};
```

```
class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void);
        double abs(void);
};
```

W zasięgu tej definicji klasa `Wektor_3D` będzie zawierała pola `x`, `y` i `z`. Metody zdefiniowane w klasie `Wektor_3D` zastępują analogiczne metody zdefiniowane w klasie macierzystej.

Definiowanie klas pochodnych

W C++ w definicji klasy można umieścić listę klas nadrzędnych w stosunku do niej (z modyfikatorami dostępu).

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void);
        double abs(void);
};
```

```
class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void);
        double abs(void);
};
```

W zasięgu tej definicji klasa `Wektor_3D` będzie zawierała pola `x`, `y` i `z`. Metody zdefiniowane w klasie `Wektor_3D` zastępują analogiczne metody zdefiniowane w klasie macierzystej.

Modyfikator dostępu `public` oznacza, że dostęp do pól `x`, `y` w klasie pochodnej jest taki, jak w klasie macierzystej.

Dziedziczenie i operacje przypisania

Zasada przypisania dla obiektów

Obiektowi klasy bazowej zawsze można przypisać obiekt klasy pochodnej. Wówczas *domyślnie* (tzn. jeśli przypisanie nie jest przeciążone *w klasie bazowej*) wartości pól będących składnikami obydwu klas są kopiowane z obiektu klasy pochodnej do obiektu klasy bazowej (zgodnie z kierunkiem przypisania).

Dziedziczenie i operacje przypisania

Zasada przypisania dla obiektów

Obiektowi klasy bazowej zawsze można przypisać obiekt klasy pochodnej. Wówczas *domyślnie* (tzn. jeśli przypisanie nie jest przeciążone *w klasie bazowej*) wartości pól będących składnikami obydwu klas są kopiowane z obiektu klasy pochodnej do obiektu klasy bazowej (zgodnie z kierunkiem przypisania).

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void);
        double abs(void);
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void);
        double abs(void);
};
```

```
Wektor_3D w_3d;
Wektor_2D w_2d;

...
w_2d = w_3d;  // OK

// Wartości pól x, y z obiektu w_3d są kopiowane
// do pól x, y w obiekcie w_2d (odpowiednio).

// Przypisanie w odwrotnym kierunku
// byłoby błędne!
```

Dziedziczenie, referencje i operacje przypisania

Zasada przypisania dla referencji

Obiekt klasy pochodnej zawsze może być przedmiotem (celem) referencji do obiektu klasy bazowej.

Dziedziczenie, referencje i operacje przypisania

Zasada przypisania dla referencji

Obiekt klasy pochodnej zawsze może być przedmiotem (celem) referencji do obiektu klasy bazowej.

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void);
        double abs(void);
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void);
        double abs(void);
};
```

```
void drukuj(Wektor_2D& wektor)
{
        cout << '(' << wektor.x << ','
                << wektor.y << ')' << endl;
}

...
Wektor_3D w_3d;

...
drukuj(w_3d);  // OK

// Wewnątrz funkcji drukuj() obiekt w_3d
// występuje pod nazwą wektor i
// jest traktowany jako obiekty klasy
// Wektor_2D.
```

Dziedziczenie, wskaźniki i operacje przypisania

Zasada przypisania dla wskaźników

Wskaźnikowi do obiektów klasy bazowej zawsze można przypisać adres obiektu klasy pochodnej. Wówczas wskaźnik ten może być używany przy odwołaniach do pól i wywoływaniu metod z tego obiektu, traktowanego jako obiekt klasy bazowej.

Dziedziczenie, wskaźniki i operacje przypisania

Zasada przypisania dla wskaźników

Wskaźnikowi do obiektów klasy bazowej zawsze można przypisać adres obiektu klasy pochodnej. Wówczas wskaźnik ten może być używany przy odwołaniach do pól i wywoływaniu metod z tego obiektu, traktowanego jako obiekt klasy bazowej.

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void);
        double abs(void);
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void);
        double abs(void);
};
```

```
Wektor_3D w_3d;
Wektor_2D *wsk;

...
wsk = &w_3d;  // OK

cout << wsk->abs() << endl;  // Wektor_2D::abs()

// Metoda abs() z klasy Wektor_2D zostanie
// wywołana w kontekście obiektu w_3d i będzie
// go traktować jako obiekt klasy Wektor_2D.
```

Polimorfizm

Obiekty klas pochodnych mogą zastępować obiekty klas bazowych we wszystkich sytuacjach. W szczególności dowolna funkcja operująca obiektami klasy bazowej może także operować obiektami dowolnej klasy pochodnej w stosunku do niej (wtedy te obiekty są traktowane jako obiekty klasy bazowej).

Polimorfizm

Obiekty klas pochodnych mogą zastępować obiekty klas bazowych we wszystkich sytuacjach. W szczególności dowolna funkcja operująca obiektami klasy bazowej może także operować obiektami dowolnej klasy pochodnej w stosunku do niej (wtedy te obiekty są traktowane jako obiekty klasy bazowej).

Polimorfizm (*ang. polymorphism*)

Własność języka programowania pozwalająca posługiwać się zmiennymi o różnych typach danych w jednakowy sposób.

Polimorfizm

Obiekty klas pochodnych mogą zastępować obiekty klas bazowych we wszystkich sytuacjach. W szczególności dowolna funkcja operująca obiektami klasy bazowej może także operować obiektami dowolnej klasy pochodnej w stosunku do niej (wtedy te obiekty są traktowane jako obiekty klasy bazowej).

Polimorfizm (*ang. polymorphism*)

Własność języka programowania pozwalająca posługiwać się zmiennymi o różnych typach danych w jednakowy sposób.

Polimorficzne funkcje

Mogą operować argumentami o różnych typach danych (np. funkcje w C++, których argumenty są referencjami lub wskaźnikami do zmiennych obiektowych).

Dziedziczenie i metody o jednakowych nagłówkach

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void)
        {
                return x*x + y*y;
        }
        double abs(void)
        {
                return sqrt(norma());
        }
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void)
        {
                return x*x + y*y + z*z;
        }
        double abs(void);
        {
                return sqrt(norma());
        }
};
```

Dziedziczenie i metody o jednakowych nagłówkach

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void)
        {
                return x*x + y*y;
        }
        double abs(void)
        {
                return sqrt(norma());
        }
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void)
        {
                return x*x + y*y + z*z;
        }
        double abs(void);
        {
                return sqrt(norma());
        }
};
```

```
void drukuj_abs(Wektor_2D& wektor)
{
        cout << wektor.abs() << endl;
}

int main()
{
        Wektor_3D w;

        w.x = 0;
        w.y = 3;
        w.z = 4;

        drukuj_abs(w);  // Wydrukuje 3

        // Zmienna w jest traktowana przez
        // funkcję drukuj_abs() tak, jakby była
        // klasy Wektor_2D, więc zostaną użyte
        // metody abs() i norma() zdefiniowane
        // dla klasy Wektor_2D.

        return 0;
}
```

Wirtualne metody

Wirtualna metoda (*ang. virtual method*)

Metoda zdefiniowana w pewnej klasie w taki sposób, że w klasach pochodnych w stosunku do tej klasy można wstawić na jej miejsce (*ang. override*) nowe metody z takim samym nagłówkiem.

Wirtualne metody

Wirtualna metoda (*ang. virtual method*)

Metoda zdefiniowana w pewnej klasie w taki sposób, że w klasach pochodnych w stosunku do tej klasy można wstawić na jej miejsce (*ang. override*) nowe metody z takim samym nagłówkiem.

Jeżeli metoda zostanie zadeklarowana jako wirtualna (w klasie bazowej) i w klasie pochodnej w stosunku do niej jest zdefiniowana metoda o takim samym nagłówku, to dla obiektów klasy pochodnej w każdej sytuacji będzie wywoływana metoda zdefiniowana w klasie pochodnej.

Wirtualne metody

Wirtualna metoda (*ang. virtual method*)

Metoda zdefiniowana w pewnej klasie w taki sposób, że w klasach pochodnych w stosunku do tej klasy można wstawić na jej miejsce (*ang. override*) nowe metody z takim samym nagłówkiem.

Jeżeli metoda zostanie zadeklarowana jako wirtualna (w klasie bazowej) i w klasie pochodnej w stosunku do niej jest zdefiniowana metoda o takim samym nagłówku, to dla obiektów klasy pochodnej w każdej sytuacji będzie wywoływana metoda zdefiniowana w klasie pochodnej.

W szczególności będzie tak w przypadku, gdy obiekt klasy pochodnej zastępuje obiekt klasy bazowej zawierającej wirtualną metodę.

Deklarowanie wirtualnych metod

```
class Wektor_2D {
  public:
        double x, y;
        virtual double norma(void)
        {
                return x*x + y*y;
        }
        double abs(void)
        {
                return sqrt(norma());
        }
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void)
        {
                return x*x + y*y + z*z;
        }
};
```

Deklarowanie wirtualnych metod

```
class Wektor_2D {
  public:
        double x, y;
        virtual double norma(void)
        {
                return x*x + y*y;
        }
        double abs(void)
        {
                return sqrt(norma());
        }
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void)
        {
                return x*x + y*y + z*z;
        }
};
```

```
void drukuj_abs(Wektor_2D& wektor)
{
        cout << wektor.abs() << endl;
}

int main()
{
        Wektor_3D w_3d;

        w_3d.x = 0;
        w_3d.y = 3;
        w_3d.z = 4;

        drukuj_abs(w_3d);  // Wydrukuje 5

        // Zmienna w jest traktowana przez
        // funkcję drukuj_abs() tak, jakby była
        // klasy Wektor_2D, ale metoda norma()
        // jest wirtualna, więc dla zmiennej w_3d
        // będzie wywołana metoda norma()
        // zdefiniowana dla Wektor_3D.

        return 0;
}
```

Wirtualne destruktory

W C++ destruktor może być zadeklarowany jako wirtualna metoda.

Wirtualne destruktory

W C++ destruktor może być zadeklarowany jako wirtualna metoda.

W takim przypadku destruktor zdefiniowany w klasie pochodnej jest zawsze wykonywany dla obiektów tej klasy przed destruktorem zdefiniowanym dla klasy bazowej.

Wirtualne destruktory

W C++ destruktor może być zadeklarowany jako wirtualna metoda.

W takim przypadku destruktor zdefiniowany w klasie pochodnej jest zawsze wykonywany dla obiektów tej klasy przed destruktorem zdefiniowanym dla klasy bazowej.

Jeżeli destruktor nie jest wirtualny, to destruktor zdefiniowany dla klasy pochodnej może nie być wykonany, w zależności od sposobu operowania obiektem.

Wywoływanie konstruktora klasy bazowej

Konstruktor klasy pochodnej może (i na ogół powinien) jawnie określić który konstruktor klasy bazowej należy wykonać.

```
class Wektor_2D {
  public:
        double x, y;
        Wektor_2D(double a, double b): x(a), y(b) {}
        ...
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        Wektor_3D(double a, double b, double c): Wektor_2D(a, b), z(c) {}
        ...
};
```

Modyfikator dostępu `protected`

Składniki klasy zadeklarowane z modyfikatorem dostępu `protected` są dostępne dla metod będących składnikami tej klasy oraz klas pochodnych w stosunku do niej.

Modyfikator dostępu protected

Składniki klasy zadeklarowane z modyfikatorem dostępu `protected` są dostępne dla metod będących składnikami tej klasy oraz klas pochodnych w stosunku do niej.

```
class Wektor_2D {
  protected:
        double x, y;
  public:
        virtual double norma(void);
        ...
};

class Wektor_3D : public Wektor_2D {
        double z;
  public:
        double norma(void)
        {
                return x*x + y*y + z*z;   // OK
        }
        ...
};
```

Modyfikator dostępu protected

Składniki klasy zadeklarowane z modyfikatorem dostępu protected są dostępne dla metod będących składnikami tej klasy oraz klas pochodnych w stosunku do niej.

```
class Wektor_2D {
  protected:
        double x, y;
  public:
        virtual double norma(void);
        ...
};

class Wektor_3D : public Wektor_2D {
        double z;
  public:
        double norma(void)
        {
                return x*x + y*y + z*z;   // OK
        }
        ...
};
```

```
...

int main()
{
        ...

        cout << w.x << endl; // Błąd!
        ...
}
```

Dostęp do składników klas bazowych

Domyślnie składniki klasy bazowej są traktowane tak, jakby były zadeklarowane w klasie pochodnej z modyfikatorem dostępu `private`.

Dostęp do składników klas bazowych

Domyślnie składniki klasy bazowej są traktowane tak, jakby były zadeklarowane w klasie pochodnej z modyfikatorem dostępu `private`.

Dla składników klasy bazowej można jawnie określić dostęp do nich w obiektach klas pochodnych, z pomocą modyfikatora dostępu przed nazwą klasy bazowej w definicji klasy pochodnej.

```
class Wektor_2D {
  public:
        double x, y;
        ...
};

class Wektor_3D : public Wektor_2D {
        ...
};
```

Dostęp do składników klas bazowych

Domyślnie składniki klasy bazowej są traktowane tak, jakby były zadeklarowane w klasie pochodnej z modyfikatorem dostępu `private`.

Dla składników klasy bazowej można jawnie określić dostęp do nich w obiektach klas pochodnych, z pomocą modyfikatora dostępu przed nazwą klasy bazowej w definicji klasy pochodnej.

```
class Wektor_2D {
  public:
        double x, y;
        ...
};

class Wektor_3D : public Wektor_2D {
        ...
};
```

`public` oznacza, że dostęp do składników klasy bazowej w obiekcie klasy pochodnej ma być taki, jaki byłby w obiekcie klasy bazowej.

Dostęp do składników klas bazowych i protected

protected oznacza, że składniki klasy bazowej zadeklarowane (w definicji klasy bazowej) z modyfikatorem dostępu public mają być traktowane tak, jakby były zadeklarowane w klasie pochodnej z modyfikatorem dostępu protected.

```
class Wektor_2D {
  public:
        double x, y;
        ...
};

class Wektor_3D : protected Wektor_2D {
        ...
};

int main()
{
        Wektor_3D w3d;

        w3d.x = 0;  // Błąd!
        ...
}
```

Klasy abstrakcyjne

Klasa abstrakcyjna (*ang. abstract class*)

Klasa, dla której nie można zdefiniować obiektu. Można tylko tworzyć klasy pochodne w stosunku do niej i tworzyć obiekty tych klas.

Klasy abstrakcyjne

Klasa abstrakcyjna (*ang. abstract class*)

Klasa, dla której nie można zdefiniować obiektu. Można tylko tworzyć klasy pochodne w stosunku do niej i tworzyć obiekty tych klas.

W C++ klasę abstrakcyją definiuje się deklarując przynajmniej jedną wirtualną metodę bez implementacji.

Klasy abstrakcyjne

Klasa abstrakcyjna (*ang. abstract class*)

Klasa, dla której nie można zdefiniować obiektu. Można tylko tworzyć klasy pochodne w stosunku do niej i tworzyć obiekty tych klas.

W C++ klasę abstrakcyją definiuje się deklarując przynajmniej jedną wirtualną metodę bez implementacji.

```
class Wektor {
  public:
        double x, y;
        virtual double norma(void) = 0;
        ...
};
```

```
class Wektor_2D : public Wektor {
  public:
        double norma(void)
        {
                return x*x + y*y;
        }
        ...
};
```

Wejście-wyjście w C++

`ios_base`

Klasa bazowa dla wszystkich klas wejścia-wyjścia reprezentujących strumienie danych (*ang. data stream*).

Wejście-wyjście w C++

`ios_base`

Klasa bazowa dla wszystkich klas wejścia-wyjścia reprezentujących strumienie danych (*ang. data stream*).

`ios`

Klasa pochodna od `ios_base`. Zawiera składniki wspólne dla strumieni wejściowych (*ang. input*) i wyjściowych (*ang. output*).

Klasy wejścia-wyjścia w C++

`istream`

Klasa pochodna od `ios`. Zawiera składniki potrzebne do obsługi strumieni wejściowych, niezależnie od źródła (*ang. source*) danych, m. in. rodzinę metod `operator >>()`.

`ostream`

Klasa pochodna od `ios`. Zawiera składniki potrzebne do obsługi strumieni wyjściowych, niezależnie od miejsca przeznaczenia (*ang. destination*) danych, m. in. rodzinę metod `operator <<()`.

Klasy wejścia-wyjścia w C++

`istream`

Klasa pochodna od `ios`. Zawiera składniki potrzebne do obsługi strumieni wejściowych, niezależnie od źródła (*ang. source*) danych, m. in. rodzinę metod `operator >>()`.

`ostream`

Klasa pochodna od `ios`. Zawiera składniki potrzebne do obsługi strumieni wyjściowych, niezależnie od miejsca przeznaczenia (*ang. destination*) danych, m. in. rodzinę metod `operator <<()`.

`iostream`

Klasa pochodna od `istream` i `ostream`. Zawiera składniki potrzebne do obsługi strumieni wejściowych i wyjściowych, niezależnie od źródła lub miejsca przeznaczenia danych.

Klasy wejścia-wyjścia w C++ – c. d.

`ifstream`

Klasa pochodna od `istream`. Zawiera składniki potrzebne do odczytywania danych z pliku (tekstowego).

`ofstream`

Klasa pochodna od `ostream`. Zawiera składniki potrzebne do zapisywania danych w pliku (tekstowym).

Klasy wejścia-wyjścia w C++ – c. d.

`ifstream`

Klasa pochodna od `istream`. Zawiera składniki potrzebne do odczytywania danych z pliku (tekstowego).

`ofstream`

Klasa pochodna od `ostream`. Zawiera składniki potrzebne do zapisywania danych w pliku (tekstowym).

`fstream`

Klasa pochodna od `iostream`. Zawiera składniki potrzebne do zapisywania danych w pliku (tekstowym) i odczytywania danych z pliku (tekstowego).

Klasy wejścia-wyjścia w C++ – c. d.

`istringstream`

Klasa pochodna od `istream`. Zawiera składniki potrzebne do odczytywania danych z ciągu znaków (reprezentowanego przez obiekt klasy `string`).

`ostringstream`

Klasa pochodna od `ostream`. Zawiera składniki potrzebne do zapisywania danych do ciągu znaków (reprezentowanego przez obiekt klasy `string`).

Klasy wejścia-wyjścia w C++ – c. d.

`istringstream`

Klasa pochodna od `istream`. Zawiera składniki potrzebne do odczytywania danych z ciągu znaków (reprezentowanego przez obiekt klasy `string`).

`ostringstream`

Klasa pochodna od `ostream`. Zawiera składniki potrzebne do zapisywania danych do ciągu znaków (reprezentowanego przez obiekt klasy `string`).

`stringstream`

Klasa pochodna od `iostream`. Zawiera składniki potrzebne do odczytywania danych z ciągu znaków i zapisywania danych do ciągu znaków (reprezentowanego przez obiekt klasy `string`).

Błędy wejścia-wyjścia i konwersja typów danych

W klasach `istream` i `ostream` przeciążenie konwersji typów danych pozwala na używanie obiektów tych klas w wyrażeniach typów całkowitych oraz typu `bool`.

Błędy wejścia-wyjścia i konwersja typów danych

W klasach `istream` i `ostream` przeciążenie konwersji typów danych pozwala na używanie obiektów tych klas w wyrażeniach typów całkowitych oraz typu `bool`.

Obowiązuje przy tym zasada, że jeśli obiekt reprezentuje wartość liczbową 0 lub wartość `false` typu `bool`, to nie jest skojarzony z żadnym źródłem lub miejscem przeznaczenia danych albo reprezentuje strumień, dla którego ostatnia operacja wejścia-wyjścia zakończyła się niepowodzeniem.

Błędy wejścia-wyjścia i konwersja typów danych

W klasach `istream` i `ostream` przeciążenie konwersji typów danych pozwala na używanie obiektów tych klas w wyrażeniach typów całkowitych oraz typu `bool`.

Obowiązuje przy tym zasada, że jeśli obiekt reprezentuje wartość liczbową 0 lub wartość `false` typu `bool`, to nie jest skojarzony z żadnym źródłem lub miejscem przeznaczenia danych albo reprezentuje strumień, dla którego ostatnia operacja wejścia-wyjścia zakończyła się niepowodzeniem.

Operacje wejścia-wyjścia reprezentowane przez symbole >> oraz << zwracają wyniki będące referencjami do obiektów klasy `istream` i `ostream`, odpowiednio.

Wyniki operacji wejścia-wyjścia

Operator wyjścia << zwraca wynik typu `ostream&`, który można wykorzystać do sprawdzenia, czy ostatnia operacja zakończyła się sukcesem, tzn. czy „konwersja" tego obiektu na wartość liczbową (lub typu `bool`) daje liczbę różną od zera (lub `true`).

Wyniki operacji wejścia-wyjścia

Operator wyjścia << zwraca wynik typu `ostream&`, który można wykorzystać do sprawdzenia, czy ostatnia operacja zakończyła się sukcesem, tzn. czy „konwersja" tego obiektu na wartość liczbową (lub typu `bool`) daje liczbę różną od zera (lub `true`).

Operator wejścia >> zwraca wynik typu `istream&`, który można wykorzystać do sprawdzenia, czy ostatnia operacja zakończyła się sukcesem – jak dla operatora wyjścia.

Wyniki operacji wejścia-wyjścia

Operator wyjścia << zwraca wynik typu `ostream&`, który można wykorzystać do sprawdzenia, czy ostatnia operacja zakończyła się sukcesem, tzn. czy „konwersja" tego obiektu na wartość liczbową (lub typu `bool`) daje liczbę różną od zera (lub `true`).

Operator wejścia >> zwraca wynik typu `istream&`, który można wykorzystać do sprawdzenia, czy ostatnia operacja zakończyła się sukcesem – jak dla operatora wyjścia.

Przykład

```
if (cin >> r)  // Odczytaj liczbę z cin i sprawdź, czy udało się.
    cout << r*r << endl;
```

Łączenie operacji wejścia-wyjścia

Wyniki zwracane przez >> i << pozwalają na łączenie operacji wejścia-wyjścia w ciągi.

Przykład

1. Zapisz `a` do `cout` i zwróć referencję do `cout` jako wynik.
2. Zapisz `b` w strumieniu, do którego referencja została zwrócona jako wynik poprzedniej operacji i zwróć referencję do tego strumienia jako wynik.
3. Zapisz `endl` w strumieniu, do którego referencja została zwrócona jako wynik poprzedniej operacji i zwróć referencję do tego strumienia jako wynik.

```
cout << a << b << endl;  // lub ((cout << a) << b) << endl;
```

Przeciążanie operacji wejścia-wyjścia

Operatory wejścia-wyjścia >> i << mogą być zdefiniowane dla każdej klasy z wykorzystaniem standardowego mechanizmu przeciążania operatorów z użyciem funkcji o dwóch argumentach.

Przeciążanie operacji wejścia-wyjścia

Operatory wejścia-wyjścia >> i << mogą być zdefiniowane dla każdej klasy z wykorzystaniem standardowego mechanizmu przeciążania operatorów z użyciem funkcji o dwóch argumentach.

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

class Wektor {
  public:
    double x, y;
    Wektor(void): x(0), y(0) {}
    Wektor(double a, double b): x(a), y(b) {}
    ...  // Inne metody i pola.
};

ostream& operator <<(ostream& out, Wektor& w)
{
    return out << '(' << w.x << ", " << w.y << ')';
}
```

```
int main()
{
    Wektor w(3, 4);

    ...  // Inne instrukcje.
    // Wykonaj (operator <<(cout, w)) << endl
    cout << w << endl;

    return 0;
}
```

Błędny argument metody-operatora

W następującym kodzie jest problem polegający na tym, że przekazanie błędnego argumentu metodzie `operator =()` zostanie zignorowane, ale wynik wykonania programu prawdopodobnie będzie błędny.

```
class Tablica {
        double *elem;
        int n;
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        ...
        Tablica& operator =(Tablica& t);
};

...
```

```
Tablica& Tablica::operator =(Tablica& t)
{
        if (n == t.n)  // Warunkowo
                for (int i = 0; i < n; i++)
                        elem[i] = t.elem[i];
        return *this;  // Bezwarunkowo!
}
```

Błędny argument metody-operatora

W następującym kodzie jest problem polegający na tym, że przekazanie błędnego argumentu metodzie `operator =()` zostanie zignorowane, ale wynik wykonania programu prawdopodobnie będzie błędny.

```
class Tablica {
        double *elem;
        int n;
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        ...
        Tablica& operator =(Tablica& t);
};

...
```

```
Tablica& Tablica::operator =(Tablica& t)
{
        if (n == t.n)  // Warunkowo
                for (int i = 0; i < n; i++)
                        elem[i] = t.elem[i];
        return *this;  // Bezwarunkowo!
}
```

Rozwiązaniem może być zapisanie metody `operator =()` tak, aby zgłaszała wyjątek, gdy warunek `n == t.n` nie jest spełniony.

Co to jest wyjątek?

Wyjątek (*ang. exception*)

Błąd ujawniający się podczas wykonywania programu, który powoduje, że wykonywanie go w normalny sposób nie może być kontynuowane.

Co to jest wyjątek?

Wyjątek (*ang. exception*)

Błąd ujawniający się podczas wykonywania programu, który powoduje, że wykonywanie go w normalny sposób nie może być kontynuowane.

Zgłoszenie wyjątku (*ang. throwing an exception*)

Czynność polegająca na przerwaniu aktualnie wykonywanego kodu (najczęściej funkcji) i utworzeniu obiektu (dowolnego typu) zawierającego informacje o błędzie.

Co to jest wyjątek?

Wyjątek (*ang. exception*)

Błąd ujawniający się podczas wykonywania programu, który powoduje, że wykonywanie go w normalny sposób nie może być kontynuowane.

Zgłoszenie wyjątku (*ang. throwing an exception*)

Czynność polegająca na przerwaniu aktualnie wykonywanego kodu (najczęściej funkcji) i utworzeniu obiektu (dowolnego typu) zawierającego informacje o błędzie.

Instrukcja `throw`

W C++ wyjątki zgłasza się z pomocą specjalnej instrukcji, złożonej ze słowa kluczowego `throw` i wyrażenia (dowolnego typu), które stanowi jej argument.

Zgłoszenie wyjątku z użyciem `throw`

Argument instrukcji `throw` służy do utworzenia obiektu zawierającego informacje o błędzie, który spowodował zgłoszenie wyjątku (reprezentującego wyjątek).

Zgłoszenie wyjątku z użyciem `throw`

Argument instrukcji `throw` służy do utworzenia obiektu zawierającego informacje o błędzie, który spowodował zgłoszenie wyjątku (reprezentującego wyjątek).

```
class Tablica {
        double *elem;
        int n;
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        ...
        Tablica& operator =(Tablica& t);
};

...
```

```
Tablica& Tablica::operator =(Tablica& t)
{
        if (n != t.n)
                throw -1;

        for (int i = 0; i < n; i++)
                elem[i] = t.elem[i];
        return *this;
}
```

Zgłoszenie wyjątku z użyciem `throw`

Argument instrukcji `throw` służy do utworzenia obiektu zawierającego informacje o błędzie, który spowodował zgłoszenie wyjątku (reprezentującego wyjątek).

```
class Tablica {
        double *elem;
        int n;
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        ...
        Tablica& operator =(Tablica& t);
};

...
```

```
Tablica& Tablica::operator =(Tablica& t)
{
        if (n != t.n)
                throw -1;

        for (int i = 0; i < n; i++)
                elem[i] = t.elem[i];
        return *this;
}
```

Metoda `operator =()` zgłasza wyjątek, jeżeli obiekty po obu stronach symbolu przypisania nie są ze sobą zgodne.

Skutki użycia `throw`

Funkcja zgłaszająca wyjątek

W wyniku wykonania `throw` funkcja (lub metoda) zgłaszająca wyjątek jest przerywana (podobnie, jak w przypadku wykonania `return`), a obiekt reprezentujący wyjątek jest przekazywany do funkcji (lub metody), która ją wywołała.

Skutki użycia `throw`

Funkcja zgłaszająca wyjątek

W wyniku wykonania `throw` funkcja (lub metoda) zgłaszająca wyjątek jest przerywana (podobnie, jak w przypadku wykonania `return`), a obiekt reprezentujący wyjątek jest przekazywany do funkcji (lub metody), która ją wywołała.

Funkcja, która wywołała funkcję zgłaszającą wyjątek

Odebranie wyjątku (obiektu reprezentującego wyjątek) od wywołanej przez nią funkcji (metody) ma taki skutek, jakby ona sama zgłosiła wyjątek (tzn. jakby w trakcie jej wykonywania nastąpiło wykonanie `throw` z takim argumentem, jak obiekt reprezentujący wyjątek).

Przekazywanie obiektów reprezentujących wyjątki

Jeżeli `a()` wywołuje `b()`, która wywołuje `c()`, to zgłoszenie wyjątku przez `c()` spowoduje przekazanie obiektu reprezentującego wyjątek do `b()`, a następnie do `a()` (i tak dalej).

Przekazywanie obiektów reprezentujących wyjątki

Jeżeli `a()` wywołuje `b()`, która wywołuje `c()`, to zgłoszenie wyjątku przez `c()` spowoduje przekazanie obiektu reprezentującego wyjątek do `b()`, a następnie do `a()` (i tak dalej).

Obiekty reprezentujące wyjątki są przekazywane „w dół” łańcucha wywołań funkcji (*ang. function call chain*) i ostatecznie docierają do funkcji `main()`, chyba że „po drodze” zostaną przechwycone (*ang. catch*).

Przekazywanie obiektów reprezentujących wyjątki

Jeżeli `a()` wywołuje `b()`, która wywołuje `c()`, to zgłoszenie wyjątku przez `c()` spowoduje przekazanie obiektu reprezentującego wyjątek do `b()`, a następnie do `a()` (i tak dalej).

Obiekty reprezentujące wyjątki są przekazywane „w dół" łańcucha wywołań funkcji (*ang. function call chain*) i ostatecznie docierają do funkcji `main()`, chyba że „po drodze" zostaną przechwycone (*ang. catch*).

Dotarcie obiektu reprezentującego wyjątek do funkcji `main()` powoduje natychmiastowe przerwanie wykonywania programu (z odpowiednim komunikatem o błędzie), chyba że (obiekt reprezentujący) wyjątek zostanie przechwycony wewnątrz funkcji `main()`.

Przechwytywanie wyjątków

Bloki `try` i `catch`

Wyjątek może być przechwycony, jeżeli:

1. Wywołanie zgłaszającej go funkcji miało miejsce w bloku poprzedzanym przez słowo kluczowe `try` oraz
2. z tym blokiem związany jest blok poprzedzany słowem kluczowym `catch` i definicją zmiennej (w nawiasie okrągłym), której typ danych odpowiada typowi danych obiektu reprezentującego wyjątek.

Przechwytywanie wyjątków

Bloki `try` i `catch`

Wyjątek może być przechwycony, jeżeli:

1. Wywołanie zgłaszającej go funkcji miało miejsce w bloku poprzedzanym przez słowo kluczowe `try` oraz
2. z tym blokiem związany jest blok poprzedzany słowem kluczowym `catch` i definicją zmiennej (w nawiasie okrągłym), której typ danych odpowiada typowi danych obiektu reprezentującego wyjątek.

```
Tablica& Tablica::operator =(Tablica& t)
{
        if (n != t.n)
                throw -1;
        for (int i = 0; i < n; i++)
                elem[i] = t.elem[i];
        return *this;
}
```

```
void kopiuj(Tablica& a, Tablica& b)
{
        try {
                a = b;
        } catch (int kod) {
                cerr << "BŁĄD: " << kod << endl;
        }
}
```

Zasady przechwytywania wyjątków

Z jednym blokiem `try` można związać dowolną liczbę bloków `catch`. Wówczas muszą one następować jeden po drugim (bez żadnych instrukcji między nimi).

Zasady przechwytywania wyjątków

Z jednym blokiem `try` można związać dowolną liczbę bloków `catch`. Wówczas muszą one następować jeden po drugim (bez żadnych instrukcji między nimi).

Jeżeli wiele bloków `catch` „pasuje” do obiektu reprezentującego wyjątek, wybrany zostanie ten, który jest zdefiniowany jako pierwszy.

Zasady przechwytywania wyjątków

Z jednym blokiem `try` można związać dowolną liczbę bloków `catch`. Wówczas muszą one następować jeden po drugim (bez żadnych instrukcji między nimi).

Jeżeli wiele bloków `catch` „pasuje" do obiektu reprezentującego wyjątek, wybrany zostanie ten, który jest zdefiniowany jako pierwszy.

Jeżeli blok `catch` zostanie „dopasowany" do obiektu reprezentującego wyjątek (tzn. jego typ danych odpowiada typowi danych zmiennej zdefiniowanej w nawiasie okrągłym po słowie kluczowym `catch` dla tego bloku), to zostaną wykonane instrukcje znajdujące się wewnątrz tego bloku.

Zasady przechwytywania wyjątków c. d.

Jeżeli blok `catch` zostanie „dopasowany” do wyjątku (tzn. do obiektu reprezentującego wyjątek), to:

1. zmienna zdefiniowana w nawiasie okrągłym po słowie kluczowym `catch` dla tego bloku otrzyma wartość równą obiektowi reprezentującemu wyjątek oraz
2. zmienna ta może być wykorzystywana w instrukcjach wewnątrz tego bloku tak, jak gdyby była argumentem funkcji (można ją nazwać zmienną reprezentującą wyjątek).

Zasady przechwytywania wyjątków c. d.

Jeżeli blok `catch` zostanie „dopasowany” do wyjątku (tzn. do obiektu reprezentującego wyjątek), to:

1. zmienna zdefiniowana w nawiasie okrągłym po słowie kluczowym `catch` dla tego bloku otrzyma wartość równą obiektowi reprezentującemu wyjątek oraz
2. zmienna ta może być wykorzystywana w instrukcjach wewnątrz tego bloku tak, jak gdyby była argumentem funkcji (można ją nazwać zmienną reprezentującą wyjątek).

Domyślna (*ang. default*) obsługa wyjątku

Blok `catch`, dla którego w nawiasie okrągłym po słowie kluczowym `catch` (zamiast definicji zmiennej) znajduje się wielokropek (`...`), zostanie dopasowany do każdego wyjątku. W tym bloku nie można wykorzystywać zmiennej reprezentującej wyjątek.

Natychmiastowe przechwytywanie wyjątków

Jeżeli wykonanie `throw` nastąpi bezpośrednio w bloku `try`, z którym związany jest blok `catch` „pasujący” do obiektu reprezentującego wyjątek, to przerwane zostanie tylko wykonywanie bloku `try` zawierającego instrukcję `throw`.

Natychmiastowe przechwytywanie wyjątków

Jeżeli wykonanie `throw` nastąpi bezpośrednio w bloku `try`, z którym związany jest blok `catch` „pasujący" do obiektu reprezentującego wyjątek, to przerwane zostanie tylko wykonywanie bloku `try` zawierającego instrukcję `throw`.

W takim przypadku obiekt reprezentujący wyjątek jest przekazywany bezpośrednio do odpowiedniego bloku `catch` bez przerywania wykonywania funkcji, w której zgłaszany jest wyjątek.

Natychmiastowe przechwytywanie wyjątków

Jeżeli wykonanie `throw` nastąpi bezpośrednio w bloku `try`, z którym związany jest blok `catch` „pasujący” do obiektu reprezentującego wyjątek, to przerwane zostanie tylko wykonywanie bloku `try` zawierającego instrukcję `throw`.

W takim przypadku obiekt reprezentujący wyjątek jest przekazywany bezpośrednio do odpowiedniego bloku `catch` bez przerywania wykonywania funkcji, w której zgłaszany jest wyjątek.

```
try {
        ...
        throw MyException;
        ...
} catch (MyException e) {
        ...
}
```

Deklarowanie typów zgłaszanych wyjątków dla funkcji

Można zadeklarować, że funkcja będzie zgłaszać tylko wyjątki określonego typu (np. `int`).

Deklarowanie typów zgłaszanych wyjątków dla funkcji

Można zadeklarować, że funkcja będzie zgłaszać tylko wyjątki określonego typu (np. `int`).

```
int moja_funkcja(MojaKlasa arg) throw(int)
{
        // Ta funkcja może (ale nie musi) zgłaszać tylko wyjątki typu int
        ...
}
```

Deklarowanie typów zgłaszanych wyjątków dla funkcji

Można zadeklarować, że funkcja będzie zgłaszać tylko wyjątki określonego typu (np. `int`).

```
int moja_funkcja(MojaKlasa arg) throw(int)
{
        // Ta funkcja może (ale nie musi) zgłaszać tylko wyjątki typu int
        ...
}
```

Użycie pustej listy typów danych przy `throw()` oznacza, że funkcja w ogóle nie będzie zgłaszać wyjątków.

Deklarowanie typów zgłaszanych wyjątków dla funkcji

Można zadeklarować, że funkcja będzie zgłaszać tylko wyjątki określonego typu (np. `int`).

```
int moja_funkcja(MojaKlasa arg) throw(int)
{
        // Ta funkcja może (ale nie musi) zgłaszać tylko wyjątki typu int
        ...
}
```

Użycie pustej listy typów danych przy `throw()` oznacza, że funkcja w ogóle nie będzie zgłaszać wyjątków.

```
int moja_funkcja(MojaKlasa arg) throw()
{
        // Ta funkcja nie może zgłaszać wyjątków
        ...
}
```

Standardowe klasy reprezentujące wyjątki

`exception`

Klasa bazowa dla standardowych klas używanych do reprezentowania wyjątków.

Standardowe klasy reprezentujące wyjątki

`exception`

Klasa bazowa dla standardowych klas używanych do reprezentowania wyjątków.

Klasami pochodnymi w stosunku do `exception` są m. in. `bad_alloc`, `bad_cast`, `bad_exception`, `bad_typeid`, `logic_error`, `runtime_error`.

Standardowe klasy reprezentujące wyjątki

`exception`

Klasa bazowa dla standardowych klas używanych do reprezentowania wyjątków.

Klasami pochodnymi w stosunku do `exception` są m. in. `bad_alloc`, `bad_cast`, `bad_exception`, `bad_typeid`, `logic_error`, `runtime_error`.

Definiując nową klasę pochodną w stosunku do `exception` zwykle przesłania się metodę `what()`, której wynikiem jest wskaźnik do tablicy znakowej (o elementach typu `char`) zawierającej komunikat o błędzie.

Przykład – wyjątek bad_alloc

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

int main () {
    try {
        double *myarray = new double[1000000000];
    } catch (exception& e) {
        cout << "Standard exception: " << e.what() << endl;
    }

    return 0;
}
```

Przykład – rozszerzanie klasy exception

```
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
 public:
    virtual const char* what() const throw()
    {
        return "My exception happened";
    }

    int x;
    myexception(void): x(0) {}
};
```

```
int main()
{
    myexception myex;

    try {
        throw myex;
    } catch (myexception& e) {
        cout << e.what() << endl;
        e.x = 1;
    }

    cout << myex.x << endl;

    return 0;
}
```

Co to jest szablon

Szablon (*ang. template*)

Konstrukcja pozwalająca uniknąć duplikowania kodu źródłowego poprzez zapisanie go w sposób umożliwiający wielokrotną kompilację dla różnych „podstawionych” klas.

Co to jest szablon

Szablon (*ang. template*)

Konstrukcja pozwalająca uniknąć duplikowania kodu źródłowego poprzez zapisanie go w sposób umożliwiający wielokrotną kompilację dla różnych „podstawionych" klas.

Następujący zapis oznacza, że definicja klasy `ElementListy` jest szablonem i należy ją dopełnić (w dalszej części programu) określając klasę `T`, dla której ma ona być skompilowana (może ona być kompilowana dla różnych klas `T`):

```
template<class T> class ElementListy {
 public:
    T dane;
    ElementListy *nast;
};
```

Przykład – klasa Stos zdefiniowana jako szablon

Na tym etapie klasa T, od której zależy definicja klasy Stos, nie została jeszcze określona.

```
template<class T> class Stos {
    ElementListy<T> *pierwszy;
 public:
    Stos(void): pierwszy(NULL) {}
    void wstaw(T s);
    T zdejmij(void);
    bool pusty(void);
};

template<class T> void Stos<T>::wstaw(T obj)
{
    ElementListy<T> *el;

    el = new ElementListy<T>;
    if (!el)
        return;
    el->dane = obj;
    el->nast = pierwszy;
    pierwszy = el;
}
```

```
template<class T> T Stos<T>::zdejmij(void)
{
    ElementListy<T> *el;
    T obj;

    // Zakładamy, że stos nie jest pusty.
    el = pierwszy;
    obj = el->dane;
    pierwszy = el->nast;
    delete el;
    return obj;
}

template<class T> bool Stos<T>::pusty(void)
{
    return pierwszy == NULL;
}
```

Przykład – program wykorzystujący klasę Stos

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
        cerr << "Wymagany 1 argument" << endl;
        return EXIT_FAILURE;
    }

    ifstream input(argv[1]);

    if (!input) {
        cerr << "Problem z otwieraniem pliku " << argv[1] << endl;
        return EXIT_FAILURE;
    }

    // Informujemy kompilator, że klasą T, od której zależy definicja klasy Stos, jest string.
    Stos<string> st;
    string wiersz;

    // Odczytujemy wiersze z pliku i wkładamy na stos.
    while (getline(input, wiersz))
        st.wstaw(wiersz);

    // Drukujemy zawartość stosu.
    while (!st.pusty())
        cout << st.zdejmij() << endl;

    return EXIT_SUCCESS;
}
```

Literatura

 B. Stroustrup, *Język C++* (Wydawnictwo Naukowo-Techniczne, Warszawa 2002).

 B. Eckel, *Thinking in C++. Edycja polska* (Wydawnictwo Helion, Gliwice 2002).