Wstęp do programowania, część IV

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

30 listopada 2011

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 komponentów (tzn. część jej komponentów 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 komponentów (tzn. część jej komponentów pochodzi z tamtej klasy).

Klasa, po której komponenty 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 komponentów (tzn. część jej komponentów pochodzi z tamtej klasy).

Klasa, po której komponenty 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, w jakim „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.

Klasy macierzyste. klasy pochodne i przypisanie

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 *obu klas* są kopiowane z obiektu klasy pochodnej do obiektu klasy bazowej (zgodnie z kierunkiem przypisania).

Klasy macierzyste. klasy pochodne i przypisanie

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 *obu 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
// jest traktowany tak, jakby był 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 tak, jak gdyby był to 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 tak, jak gdyby był to 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

Dzięki wymienionym zasadom przypisania 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.

Polimorfizm

Dzięki wymienionym zasadom przypisania 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.

Polimorfizm (*ang. polymorphism*)

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

Polimorfizm

Dzięki wymienionym zasadom przypisania 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.

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 tej klasy.

        return 0;
}
```

Wirtualne metody

Wirtualna metoda (*ang. virtual method*)

Metoda, która jest składnikiem klasy bazowej i w klasach pochodnych może być przesłonięta (*ang. override*) przez nowe metody z takim samym nagłówkiem.

Wirtualne metody

Wirtualna metoda (*ang. virtual method*)

Metoda, która jest składnikiem klasy bazowej i w klasach pochodnych może być przesłonięta (*ang. override*) przez nowe metody z takim samym nagłówkiem.

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

Wirtualne metody

Wirtualna metoda (*ang. virtual method*)

Metoda, która jest składnikiem klasy bazowej i w klasach pochodnych może być przesłonięta (*ang. override*) przez nowe metody z takim samym nagłówkiem.

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

W szczególności będzie tak w przypadku, gdy wywołanie metody odbywa się z użyciem referencji bądź wskaźnika do obiektów klasy bazowej.

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;

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

        drukuj_abs(w);  // 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
        // 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).

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).

Użycie `public` oznacza, że dostęp do składników klasy bazowej w obiektach klasy pochodnej ma być taki, jak dostęp do tych składników w obiektach klasy bazowej.

Dostęp do składników klas bazowych c. d.

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

- jeśli obiekt jest klasy bazowej, to dostęp do składnika jest nieograniczony,
- jeśli obiekt jest klasy pochodnej, to dostęp do składnika mają metody z klasy bazowej, metody z klasy pochodnej i metody z klas pochodnych w stosunku do niej.

Dostęp do składników klas bazowych c. d.

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

- jeśli obiekt jest klasy bazowej, to dostęp do składnika jest nieograniczony,
- jeśli obiekt jest klasy pochodnej, to dostęp do składnika mają metody z klasy bazowej, metody z klasy pochodnej i metody z klas pochodnych w stosunku do niej.

Użycie `private` jest równoważne określeniu domyślnych ograniczeń dostępu (tzn. dla obiektów klasy bazowej – jak w jej definicji, dla obiektów klasy pochodnej – jak dla `private`).

Definiowanie klas abstrakcyjnych

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.

Definiowanie klas abstrakcyjnych

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.

Definiowanie klas abstrakcyjnych

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*).

Wejście-wyjście w C++ c. d.

`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 <<()`.

Wejście-wyjście w C++ c. d.

`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.

Wejście-wyjście 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).

Wejście-wyjście 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).

Wejście-wyjście 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`).

Wejście-wyjście 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`).

Standardowy strumień wyjściowy

Standardowy strumień wyjściowy (*ang. standard output stream*)

Ustalone miejsce przeznaczenia danych, do którego program może zapisywać dane bez konieczności jawnego kojarzenia go z plikiem.

Standardowy strumień wyjściowy

Standardowy strumień wyjściowy (*ang. standard output stream*)

Ustalone miejsce przeznaczenia danych, do którego program może zapisywać dane bez konieczności jawnego kojarzenia go z plikiem.

Operacje zapisywania danych przeprowadzane z użyciem tego źródła przebiegają tak, jak gdyby było ono plikiem tekstowym (tzn. dane są zapisywane w postaci ciągów znaków).

Standardowy strumień wyjściowy

Standardowy strumień wyjściowy (*ang. standard output stream*)

Ustalone miejsce przeznaczenia danych, do którego program może zapisywać dane bez konieczności jawnego kojarzenia go z plikiem.

Operacje zapisywania danych przeprowadzane z użyciem tego źródła przebiegają tak, jak gdyby było ono plikiem tekstowym (tzn. dane są zapisywane w postaci ciągów znaków).

`cout`

Obiekt klasy `ostream` reprezentujący standardowy strumień wyjściowy, nazywany też standardowym wyjściem (*ang. standard output*).

Standardowy strumień wejściowy

Standardowy strumień wejściowy (*ang. standard input stream*)

Ustalone źródło danych, z którego program może odczytywać dane bez konieczności jawnego kojarzenia go z plikiem.

Standardowy strumień wejściowy

Standardowy strumień wejściowy (*ang. standard input stream*)

Ustalone źródło danych, z którego program może odczytywać dane bez konieczności jawnego kojarzenia go z plikiem.

Operacje odczytywania danych przeprowadzane z użyciem tego źródła przebiegają tak, jak gdyby było ono plikiem tekstowym (tzn. odczytywane są ciągi znaków, na podstawie których generowane są wartości liczbowe).

Standardowy strumień wejściowy

Standardowy strumień wejściowy (*ang. standard input stream*)

Ustalone źródło danych, z którego program może odczytywać dane bez konieczności jawnego kojarzenia go z plikiem.

Operacje odczytywania danych przeprowadzane z użyciem tego źródła przebiegają tak, jak gdyby było ono plikiem tekstowym (tzn. odczytywane są ciągi znaków, na podstawie których generowane są wartości liczbowe).

`cin`

Obiekt klasy `istream` reprezentujący standardowy strumień wejściowy, nazywany też standardowym wejściem (*ang. standard input*).

Standardowy strumień diagnostyczny

Standardowy strumień diagnostyczny (*ang. standard error stream*)

Miejsce przeznaczenia danych o własnościach podobnych do `cout`, ale przeznaczone do zapisywania komunikatów o błędach.

Standardowy strumień diagnostyczny

Standardowy strumień diagnostyczny (*ang. standard error stream*)

Miejsce przeznaczenia danych o własnościach podobnych do `cout`, ale przeznaczone do zapisywania komunikatów o błędach.

`cerr`

Obiekt klasy `ostream` reprezentujący standardowy strumień diagnostyczny.

Standardowy strumień diagnostyczny

Standardowy strumień diagnostyczny (*ang. standard error stream*)

Miejsce przeznaczenia danych o własnościach podobnych do `cout`, ale przeznaczone do zapisywania komunikatów o błędach.

`cerr`

Obiekt klasy `ostream` reprezentujący standardowy strumień diagnostyczny.

Standardowy strumień diagnostyczny

Standardowy strumień diagnostyczny (*ang. standard error stream*)

Miejsce przeznaczenia danych o własnościach podobnych do `cout`, ale przeznaczone do zapisywania komunikatów o błędach.

`cerr`

Obiekt klasy `ostream` reprezentujący standardowy strumień diagnostyczny.

Używając `cerr` można „odfiltrowywać" komunikaty o błędach przy przekierowaniu standardowego strumienia wyjściowego do pliku.

Przekierowywanie standardowych strumieni

W systemie Linux standardowe strumienie są domyślnie kojarzone z terminalem, przy pomocy którego uruchamiany jest program. Każdy z nich można jednak skojarzyć z określonym plikiem.

Przekierowywanie standardowych strumieni

W systemie Linux standardowe strumienie są domyślnie kojarzone z terminalem, przy pomocy którego uruchamiany jest program. Każdy z nich można jednak skojarzyć z określonym plikiem.

Skierowanie standardowego wyjścia programu na plik

```
$ ./program > wynik.txt
```

W pliku `wynik.txt` nie będą zapisywane komunikaty ze strumienia `cerr`.

Przekierowywanie standardowych strumieni

W systemie Linux standardowe strumienie są domyślnie kojarzone z terminalem, przy pomocy którego uruchamiany jest program. Każdy z nich można jednak skojarzyć z określonym plikiem.

Skierowanie standardowego wyjścia programu na plik

```
$ ./program > wynik.txt
```

W pliku `wynik.txt` nie będą zapisywane komunikaty ze strumienia `cerr`.

Podłączanie standardowego wejścia programu do pliku

```
$ ./program < dane.txt
```

Zastępowanie plików standardowymi strumieniami

Często zdarza się, że chcemy przeprowadzić operację zapisu lub odczytu używając pliku o podanej nazwie albo, jeśli nazwa pliku nie zostanie podana, jednego ze standardowych strumieni.

Zastępowanie plików standardowymi strumieniami

Często zdarza się, że chcemy przeprowadzić operację zapisu lub odczytu używając pliku o podanej nazwie albo, jeśli nazwa pliku nie zostanie podana, jednego ze standardowych strumieni.

Można to zrealizować korzystając z obserwacji, że klasa `ofstream` jest klasą pochodną w stosunku do klasy `ostream`, a klasa `ifstream` jest klasą pochodną w stosunku do klasy `istream`.

Zastępowanie plików standardowymi strumieniami

Często zdarza się, że chcemy przeprowadzić operację zapisu lub odczytu używając pliku o podanej nazwie albo, jeśli nazwa pliku nie zostanie podana, jednego ze standardowych strumieni.

Można to zrealizować korzystając z obserwacji, że klasa `ofstream` jest klasą pochodną w stosunku do klasy `ostream`, a klasa `ifstream` jest klasą pochodną w stosunku do klasy `istream`.

W tym celu należy utworzyć referencję, która będzie zastępowała obiekt klasy np. `ofstream`, reprezentujący plik lub obiekt `cout`, w zależności od sytuacji (obiektów klasy `ostream` lub `istream` nie można kopiować, czyli np. przekazywać do funkcji przez wartość).

Zastępowanie pliku przez cout

Program drukuje losową liczbę do pliku lub na standardowe wyjście.

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

void zapisz(ostream& out)
{
    srandom(time(NULL));
    out << random() << endl;
}
```

```
int main(int argc, char *argv[])
{
    if (argc <= 1) {
        zapisz(cout);
        return 0;
    }

    ofstream plik;

    plik.open(argv[1]);
    if (plik == 0) {
        zapisz(cout);
        return 0;
    }

    zapisz(plik);
    plik.close();

    return 0;
}
```

Metoda `.clear()`

Operacje wejścia-wyjścia często kończą się niepowodzeniem (z różnych powodów).

Metoda `.clear()`

Operacje wejścia-wyjścia często kończą się niepowodzeniem (z różnych powodów).

Jeżeli operacja wejścia-wyjścia zakończyła się niepowodzeniem, to obiekt reprezentujący strumień, na którym była ona przeprowadzana, „pamięta” o błędzie (w tym stanie nie powinien być używany do przeprowadzania dalszych operacji).

Metoda `.clear()`

Operacje wejścia-wyjścia często kończą się niepowodzeniem (z różnych powodów).

Jeżeli operacja wejścia-wyjścia zakończyła się niepowodzeniem, to obiekt reprezentujący strumień, na którym była ona przeprowadzana, „pamięta" o błędzie (w tym stanie nie powinien być używany do przeprowadzania dalszych operacji).

Metoda `.clear()`, dostępna w obiektach klasy `istream` i `ostream`, służy do sprowadzenia obiektu, w kontekście którego jest wywoływana, do stanu poprzedzającego operację zakończoną niepowodzeniem (tak, aby można było ją powtórzyć), np.:

```
plik.clear();
```

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;
    if (w == 25)
        cout << "OK" << endl;

    return 0;
}
```

Przeciążona konwersja typów danych i błędy wejścia-wyjścia

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

Przeciążona konwersja typów danych i błędy wejścia-wyjścia

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

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 lub reprezentuje strumień, dla którego ostatnia operacja wejścia-wyjścia zakończyła się niepowodzeniem.

Przeciążona konwersja typów danych i błędy wejścia-wyjścia

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

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 lub 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 (sprawdzając, czy „konwersja" tego obiektu na wartość liczbową lub `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 (sprawdzając, czy „konwersja" tego obiektu na wartość liczbową lub `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 (sprawdzając, czy „konwersja" tego obiektu na wartość liczbową lub `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 (sprawdzając, czy „konwersja" tego obiektu na wartość liczbową lub `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 (sprawdzając, czy „konwersja" tego obiektu na wartość liczbową lub `bool` daje liczbę różną od zera lub `true`).

Przykład

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

Pomijanie ciągów znaków nie reprezentujących liczb

Jeżeli program ma odczytać liczbę ze strumienia wejściowego, a w strumieniu znajduje się ciąg znaków nie reprezentujący liczby, to można go pominąć.

Pomijanie ciągów znaków nie reprezentujących liczb

Jeżeli program ma odczytać liczbę ze strumienia wejściowego, a w strumieniu znajduje się ciąg znaków nie reprezentujący liczby, to można go pominąć.

Niech obiekt `plik` reprezentuje strumień, a `r` będzie zmienną typu `double`.

```
if (plik >> r) {
    ... // Odczyt udany, robimy coś z r.
} else {
    plik.clear();
    while (!plik.eof()) {  // Sprawdź, czy w pliku są jeszcze nie odczytane dane.
        c = plik.peek();  // Jaki będzie następny znak?
        plik.ignore();  // Pomiń następny znak w strumieniu.
        if (c == ' ' || c == '\t' || c == '\n')
            break;  // Zakończ pętlę, jeśli ostatni pominięty znak był znakiem przerwy.
        if (c <= '9' && c >= '0') {
            plik.putback(c);
            break;
        }
    }
}
```

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;
}
```