Konwersje

- Konwersja typu ma miejsce wtedy, kiedy wspólnie korzysta się ze zmiennych lub obiektów różnych typów, na przykład w instrukcjach przypisania, w wyrażeniach, podczas przekazywania argumentów aktualnych do wywoływanej funkcji.
- Konwersja typu może być:
  - niejawna (ang.*implicite*) – wykonywana automatycznie przez kompilator.
  - jawna (ang. *explicite*)– wykonywana na życzenie użytkownika,
- Konwersja niejawna wykonywana jest przez kompilator wtedy, kiedy wynika to z kontekstu:
  - w wyrażeniach arytmetycznych (konwersja jednego argumentu na typ drugiego argumentu, zgodnie z regułami zdefiniowanymi w języku).

```
int i=2;
double d=2.5;
cout<< (i+d) <<endl; // i zostanie przekształcone do double: 2.
```

  - w instrukcji przypisania (na typ występujący po lewej stronie operatora =)

```
int i;
double d=2.5;
i=d;  /* d zostanie obcięte do typu int: 2 */
```

  - w instrukcji wywołania funkcji (na typ występujący w prototypie funkcji)

```
extern double sqrt(double);
// 2 jest przekształcone do double: 2.
cout << "Pierwiastek z 2 to: " << sqrt(2) << endl;
```

  - podczas zwracania wartości wyznaczonej w funkcji, której typ jest różny od typu zadeklarowanego w nagłówku funkcji

```
double roznica(int x, int y) {
  // wynik zostanie przekształcony do double
  return x-y;
}
```

- Konwersja jawna wykonywana jest w wyniku użycia operatora konwersji.
  - Jawna konwersja w stylu języka C wymaga podania typu w nawiasach:

```
double z=6.0
int k;
k=(int)z; // lub k=int(z);
z=(double)k;
```

  - Stara składnia języka C++ :

```
double z=6.0
int k;
k=int(z);
z=double(k);
```

  - W standardzie języka C++ wprowadzono nazwane operatory rzutowania:
    - `static_cast` - konwersja typów pokrewnych, np. `double` do `int`
    - `dynamic_cast` - konwersja kontrolowana w czasie wykonywania programu
    - `const_cast` - konwersja usuwająca kwalifikator `const`
    - `reinterpret_cast` - konwersja typów nie spokrewnionych, np. wskaźnik do nie spokrewnionego typu wskaźnikowego

- Składnia instrukcji z rzutowaniem ma postać:

```
operator_rzutowania < typ konwersji> (wyrażenie);
```

- Przykład:

```
double d=2.58;
int i = static_cast<int>d;

int zaokr(float f) {
  return static_cast<int> (f+0.5);
}
```

```
double d=2.58;
int i = (int)d;

int zaokr(float f) {
  return int(f+0.5);
}
```

Konwersje związane z obiektami

- Możliwe są następujące przypadki
  - typ wbudowany na obiekt – należy zdefiniować odpowiedni konstruktor
  - obiekt na typ wbudowany - należy zdefiniować funkcję rzutowania
  - obiekt jednej klasy na obiekt inne klasy - odpowiedni konstruktor lub funkcja rzutowania

Konwersja typu wbudowanego na typ klasy

- Wszystkie konstruktory z jednym argumentem typu wbudowanego realizują konwersję tego typu na typ klasy.

```
class Punkt
{
  int x, y ;
public :
  Punkt (int xx=0, int yy=0)
  { x = xx ; y = yy ; }

  Punkt (const Punkt & p)
  { x = p.x ; y = p.y ; }
} ;

void fun (Punkt p)
{  cout << "** funkcja fun  " << "\n" ;
}

W main()

   void fun (Punkt) ;
   Punkt a(3,4) ;

   a = Punkt (12) ;  // konwersja: jawne wywołanie konstruktora,
                     // utworzony zostanie obiekty tymczasowy typu Punkt,
                     // następnie zostanie przypisany obiektowi a

   a = 12 ;          // konwersja: wywołanie niejawne konstruktora:
                        //    a = Punkt(12)

   fun(4) ;          // konwersja: wywołanie niejawne konstruktora,
                     //    konstruktor kopiujący nie będzie używany
```

- Załóżmy, że oprócz konstruktora jednoargumentowego zdefiniowano również operator przypisania liczby typu `int` obiektowi klasy `Punkt`. Który z nich będzie użyty w przypadku instrukcji:

```
 a = 12;
```

- Konwersja z użyciem konstruktora jest wykonywana niejawnie. Można zabronić używania konstruktora do konwersji niejawnych. Służy do tego słowo kluczowe `explict`:

```
class Punkt:
{
  ...
public:
  explicit Punkt(int);
  ...
};

W main():

 a=12;       // zostanie odrzucone
 a=Punkt(3); // OK. - trzeba podać jawnie

 Punkt x(1,1), y(2,5);
 a=b+5; // błąd
 a=b+Punkt(5); // OK. - podano jawnie
```

Konwersja typu klasy na typ wbudowany

- Tego typu konwersja wymaga zdefiniowania funkcji operatora rzutowania.
- Przykład:

```
class Punkt {
  int x,y;
  ...
}
```

Aby w `main()` można było dokonywać konwersji:

```
Punkt a(2,3);
int n;
n = int(a);  // zapis jawny funkcji operatora rzutowania
n = b;       // zapis niejawny
```

trzeba zdefiniować funkcję rzutowania, która jest składową klasy:

```
operator int(); // UWAGA: w funkcji operatora konwersji
                // nie podaje się typu
```

i ma postać następującą:

```
Punkt::operator int() {
  return x;
}
```

- Przebieg konwersji: wywołanie funkcji operatora rzutowania, która przekształca obiekt w zmienną typu podstawowego, następnie wykonanie zwykłego podstawienia.

- Przykład: niejawne wywołanie operatora rzutowania gdy przekazywany jest obiekt do funkcji

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

class Punkt {
   int x, y ;
 public :
   Punkt (int xx=0, int yy=0)   // konstruktor
     { x = xx ; y = yy ; }
   Punkt (const Punkt & p)      // konstruktor kopiujący
     { x = p.x ; y = p.y ; }
   operator int()               // operator rzutowania
     { return x ; }
} ;

void fun (int n)
{  cout << "** wywolanie funkcji z argumentem : " << n << endl ;
}

int main()
{
   Punkt a(3,4) ;
   fun (a) ;
}
```

Która z funkcji (konstruktor kopiujący, operator rzutowania) zostanie użyta przy przekazywaniu obiektu `a` do funkcji? A może obydwie?

- Przykład: niejawne wywołanie operatora rzutowania podczas obliczania wartości wyrażenia

```
class Punkt
{
  int x, y ;
public :
  Punkt (int xx=0, int yy=0)
    { x = xx ; y = yy ; }
  operator int()
    { return x ;  }
} ;
```

Co wydrukuj poniższy program?

```
#include <iostream>
using namespace std;
int main()
{  Punkt a(3,4), b(5,7) ;
   int n1, n2 ;
   n1 = a + 3 ;    cout << "n1 = " << n1 << "\n" ;
   n2 = a + b ;    cout << "n2 = " << n2 << "\n" ;

   double z1, z2 ;
   z1 = a + 3 ;    cout << "z1 = " << z1 << "\n" ;
   z2 = a + b ;    cout << "z2 = " << z2 << "\n" ;
   return 0;
}
```

Jak to działa:

– Kompilator po napotkaniu operatora dodawania, w którym jednym z argumentów jest obiekt klasy sprawdza, czy nie ma przeciążonego operatora dodawania.
– Jeśli go nie znajdzie, szuka operatora rzutowania, który pozwoli mu wykonać działanie ze standardowym operatorem dodawania.

- Przykład: konwersja ułamka do typu `double` (stara i nowa składnia)

```
Ulamek::operator double() {
  {return double(licznik)/double(mianownik); }
```

lub

```
Ulamek::operator double() {
  {return static_cast<double>(licznik)/static_cast<double>(mianownik);
}
```

Użycie:

```
Ulamek x(22,7);
if (z>3.14 && z < double(x)) ...

Ulamek x (22,7);
if (z>3.14 && z < static_cast<double>(x)) ...

Ulamek x (22,7);
if (z>3.14 && z < x) ...
```

Konwersja typu jednej klasy na typ innej klasy

- Przykład: chcemy dokonywać konwersji obiektu typu Punkt na obiekt typu liczby zespolonej.
- Wariant 1: można w klasie A zdefiniować operator rzutowania wykonujący konwersję A na B:

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

class zespolona ;

class Punkt
{ int x, y ;
 public :
  Punkt (int xx=0, int yy=0) {x=xx ; y=yy ; }
  operator zespolona () ;       // konwerja Punkt --> zespolona
} ;

class zespolona
{ double rzeczywista, urojona ;
 public :
  zespolona (double r=0, double i=0) { rzeczywista=r ; urojona=i ; }
  friend Punkt::operator zespolona () ;
  void drukuj () { cout << rzeczywista << " + " << urojona <<"i\n" ; }
} ;

Punkt::operator zespolona ()
{ zespolona r ;
  r.rzeczywista=x ; r.urojona=y ;
  return r ;
}

main()
{ Punkt a(2,5) ;
  zespolona c ;
  c = (zespolona) a ; c.drukuj () ;     // konwersja jawna
  Punkt b (9,12) ;
  c =  b ;           c.drukuj () ;      // konwersja niejwana
}
```

- Wariant 2: za pomocą konstruktora; konstruktor klasy A, który otrzymuje argument klasy B wykonuje konwersję B na A.:

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

class Punkt ;
class Zespolona
{
  double rzeczywista, urojona ;
public :
  Zespolona (double r=0, double i=0)
   { rzeczywista=r ; urojona=i ; }
  Zespolona (Punkt) ;
  void drukuj () { cout << rzeczywista << " + " << urojona << "i\n" ; }
} ;

class Punkt
{
  int x, y ;
public :
  Punkt (int xx=0, int yy=0)
   { x=xx ; y=yy ; }
  friend Zespolona::Zespolona (Punkt) ;
} ;

Zespolona::Zespolona (Punkt p)
{ rzeczywista = p.x ; urojona = p.y ; }

main()
{  Punkt a(3,5) ;
   Zespolona c (a) ; c.drukuj () ;
}
```

Obsługa błędów i sytuacji wyjątkowych

- Wyjątek (ang. *exception*) to zdarzenie spowodowane przez anormalną sytuację, która nie może wystąpić podczas zwykłego wykonywania programu.

Tradycyjna obsługa błędów

- Tradycyjne podejście do zaprogramowania reakcji na błędy:
  - Zaniechanie wykonania programu i wysłanie komunikatu o błędzie.
  - Przekazanie do programu wartości reprezentującej błąd.
  - Zignorowanie błędu.
  - Przekazanie do programu poprawnej wartości i przesłanie informacji o błędzie przez specjalną zmienną.
  - Wywołanie specjalnie napisanej procedury obsługi błędu.

Co można wykorzystywać?

- Standardowe wyjście diagnostyczne `cerr` (ang. *standard error*). Wyjście to nie jest buforowane.

- Funkcja `exit` (Należy dołączyć plik nagłówkowy `<cstdlib>` lub `<stdlib.h>`). Spowoduje zakończenie programu, wywołane zostaną tylko destruktory obiektów globalnych i statycznych.

- Funkcja `abort` : (Należy dołączyć plik nagłówkowy `<cstdlib>` lub `<stdlib.h>`). Spowoduje zakończenie programu, nie wywołane zostaną żadne destruktory.

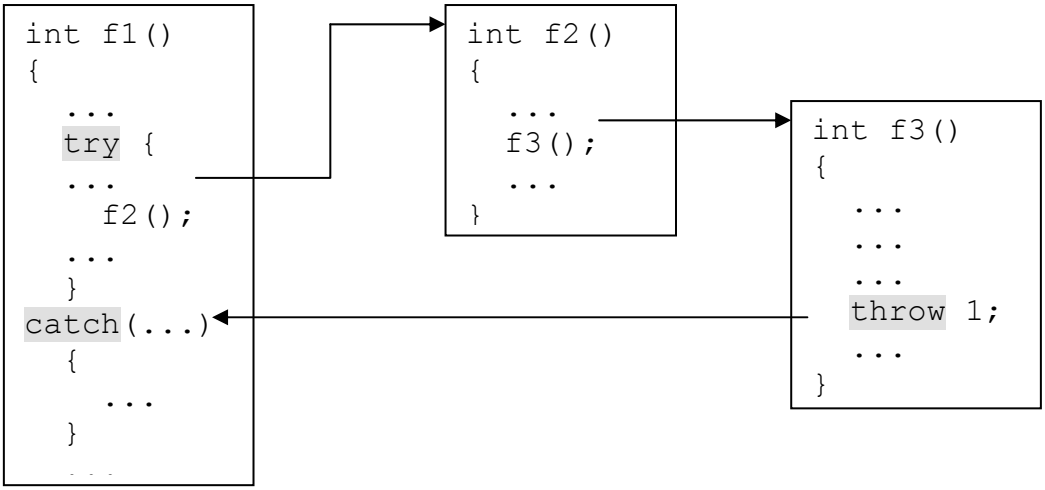
- Makroinstrukcja `assert` (Należy dołączyć plik nagłówkowy `<cstdlib>` lub `<assert.h>`) Spowoduje zakończenie wykonywania programu, jeśli nie jest spełniony podany waunek.

  Przykład:

```
void f(int *wsk)
{
  assert(wsk != 0); // wywołaj abort(), jeśli wsk==0
...
}
```

Model obsługi wyjątków (błędów) w języku C++

- Wyjątek to obiekt, który jest przekazywany z obszaru programu, w którym pojawi się problem do tej części programu, w której problem zostanie obsłużony.
- W skład mechanizmu obsługi wyjątków wchodzą:
  - miejsca zgłoszenia wyjątku – do zgłoszenia wyjątku służy słowo `throw`
  - procedury przechwytujące i obsługujące wyjątki, procedura obsługująca wyjątek rozpoczyna się od słowa `catch`
  - bloki `try`, w których mogą być uaktywnione wyjątki



Przykłady

- Przykład: wychwytywanie dzielenia przez 0, wersja A - wyjątek jest napisem

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

double iloraz(double dzielna, double dzielnik)
{
  if (dzielnik == 0)

// zgłoszenie wyjątku
    throw "proba dzielenia przez zero";
  return dzielna/dzielnik;
}

int main()
{
  double a,b,wynik;
  cout << "Wpisz dwie liczby (EOF - koniec) ";
  while ( cin >> a >> b) {

// w tym obszarze może pojawić się dzielenie przez zero
    try {
      wynik=iloraz(a,b);
      cout << "Iloraz= " << wynik << endl;
    }

// obsluga wyjatku typu napis
    catch (const char * s) {
      cout << "Wystapil wyjatek: " << s << endl;
    }

    cout << "Wpisz wpisz nastepny zestaw liczb (EOF - koniec) ";
  }

  cout << "Koniec programu" << endl;
  return 0;
}
```

- Przykład: wychwytywanie dzielenia pr zez 0, wersja B – wyjątek jest obiektem własnej klasy

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

/* klasa do obsługi wyjątków */
class DzieleniePrzezZero{
private:
  const char *komunikat;
public:
 DzieleniePrzezZero(const char *tekst)
 { komunikat = tekst;}
 const char *jaki() const
 { return komunikat; }
};

double iloraz(double dzielna, double dzielnik)
{
  if (dzielnik == 0)
    throw DzieleniePrzezZero("proba dzielenia przez zero");
  return dzielna/dzielnik;
}

int main()
{
  double a,b,wynik;
  cout << "Wpisz dwie liczby (EOF - koniec) ";
  while ( cin >> a >> b) {
    try {
      wynik=iloraz(a,b);
      cout << "Iloraz= " << wynik << endl;
    }
    catch (DzieleniePrzezZero wyj) {
      cout << "Wystapil wyjatek: " << wyj.jaki() << endl;
    }

    cout << "Wpisz dwie liczby (EOF - koniec) ";
  }
  cout << "Koniec programu" << endl;
  return 0;
}
```

Punkt zgłoszenia wyjątku, sterowanie wraca do funkcji wywołującej

W tym fragmencie można oczekiwać zgłaszania wyjątków

Procedura obsługi wyjątku typu DzieleniePrzezZero

- Przykład: klasa do działań na wektorach, wychwytywanie przekroczenia indeksu za pomocą wyjątku

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

// klasa do obsługi działań na wektorach
class wektor
{ int l_elem ;
  int * wsk ;
 public :
  wektor (int) ;
  ~wektor () ;
  int & operator [] (int) ;
} ;

// klasa do obsługi wyjatkow zwiazanych
// z przekroczeniem zakresu
class wektor_zakres
{ } ;

wektor::wektor (int n)
{ wsk = new int [l_elem = n] ; }

wektor::~wektor ()
{ delete wsk ; }

int & wektor::operator [] (int i)
{ if (i<0 || i>l_elem)
   { wektor_zakres wekt ;
     throw (wekt) ;
   }
  return wsk[i] ;
}

main ()
{
  try {
    wektor v(10) ;
    v[11] = 5 ;    // indeks poza zakresem
  }
  catch (wektor_zakres wekt)
  { cout << "wyjatek: przekroczenie zakresu  \n" ;
    exit (1) ;
  }
}
```

Punkt zgłoszenia wyjątku, sterowanie wraca do funkcji wywołującej → `throw (wekt) ;`

W tym fragmencie można oczekiwać zgłaszania wyjątków → `try {`

Procedura obsługi wyjątku → `catch (wektor_zakres wekt)`

- Przykład: Przykład: klasa do działań na wektorach, wychwytywanie błędnej inicjalizacji wektora oraz przekroczenia indeksu za pomocą wyjątków.

```
#include <iostream>
#include <stdlib.h>
using namespace std;

class wektor {
  int l_elem ;
  int * wsk ;
 public :
  wektor (int) ;
  ~wektor () ;
  int & operator [] (int) ;
} ;
// Obsługa wyjątków związanych z podaniem złej liczby składowych
class wektor_utworz
{ public :
   int ile ;              // liczba żądanych składowych
   wektor_utworz (int i) { ile = i ; }
} ;
// Obsługa wyjątków związanych z przekroczeniem zakresu
class wektor_zakres
{ public :
   int poza ;
   wektor_zakres (int i) { poza = i ; }
} ;

wektor::wektor (int n)
{ if (n <= 0)
    { wektor_utworz w(n) ;     // błąd: liczba elementów > 0
      throw w ;
    }
  wsk = new int [l_elem = n] ;
}

wektor::~wektor ()
{ delete wsk ; }

int & wektor::operator [] (int i)
{ if (i<0 || i>l_elem)
     { wektor_zakres w(i) ;   // błąd: przekroczony zakres indeksów
       throw w ;
     }
  return wsk[i] ;
}

int main () {
  try
  { wektor v(-3) ;    // wyjatek: wektor_utworz
    v[11] = 5 ;       // wyjatek: wektor_zakres
  }
  catch (wektor_zakres w)
  { cout << "wyjatek: indeks " << w.poza << " poza zakresem \n" ;
    exit (1) ;
  }
  catch (wektor_utworz w)
  { cout << "wyjatek: proba utworzenia wektora o liczbie elementow = "
         << w.ile << endl ;
    exit (2) ;
  }
}
```

Obsługa dowolnego wyjątku

- Dowolny wyjątek reprezentuje trójkropek:

```
catch (...) { // instrukcje }
```

- Jeśli jest wykorzystywany, powinien znajdować się na końcu zestawu bloków `catch`.

Ponowne zgłaszanie wyjątku

- Czasem jeden blok `catch` nie jest w stanie w pełni obsłużyć wyjątku. Wtedy w bloku `catch` można ponownie zgłosić wyjątek. Służy do tego wyrażenie:
  `throw;`

Określanie typów wyjątków zgłaszanych w funkcji

- W prototypie funkcji można określić typy wyjątków, które mogą być zgłaszane w tej funkcji:

```
void f() throw (A,B);
```

funkcja może zgłaszać wyjątki typu A i

- Klauzulę `throw` należy powtórzyć w nagłówku funkcji podczas jej definicji.
  - Powyższy zapis jest równoważny następującemu:

```
void f()
{
  try {
    ...
  }
  catch (A a) { throw ; }
  catch (B b) { throw ; }
  catch (...) { unexpected() ; } // wyjątek nieoczekiwany
}
```

- Przykład: ponowne zgłaszanie wyjątku

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
char NazwaPliku[]="test.txt";

/* zapisz tablicę do pliku */
void Zapis(fstream &f,double *Tab, int Rozmiar) {
  for(int i=0;i<Rozmiar;++i) {
     f.write((char*)(Tab+i),sizeof(double));
     if(!f.good()) throw "Blad zapisu";
  }
}
/* zapisz do tablicy */
void Zapis(double *Tab,int Rozmiar) {
  fstream f;
  f.open(NazwaPliku,ios::out);
  if(!f.good()) throw "Nie mozna otworzyc do zapisu";
  try   {
    Zapis(f,Tab,Rozmiar);
  }
  catch(...) {
    f.close(); // zamykamy plik
    throw;     // zgłaszamy ponownie
  }
}

/* czytaj do tablicy z pliku */
void Odczyt(fstream &f,double *Tab,int Rozmiar) {
  for(int i=0;i<Rozmiar;++i) {
    f.read((char*)(Tab+i),sizeof(double));
    if(!f.good()) throw "Blad odczytu"; // zgłoszony wyjątek
  }
}

/* czytaj do tablicy */
void Odczyt(double *Tab,int Rozmiar) {
  fstream f;
  f.open(NazwaPliku,ios::in);
  if(!f.good()) throw "Nie mozna otworzyc do odczytu"; // zgłoszony wyjątek
  try  {
    Odczyt(f,Tab,Rozmiar);
  }
  catch(...) {
    f.close(); // zamykamy plik
    throw;     // zgłaszamy ponownie
  }
}

int main() {
  const int R=100;
  double Tab[R];
  try   {
     Zapis(Tab,R); cout<<"\rZapisano poprawnie"<<endl;
     cout<<"Nacisnij <Enter> ...";   cin.get();
     Odczyt(Tab,R);  cout<<"\rOdczytano poprawnie"<<endl;
  }
  catch(const char *Blad) { cout<<"Wyjatek: "<< Blad<<endl;   }
  return 0;
}
```

Niepowodzenia przydziału pamięci

- Niepowodzenia przydziału pamięci można obsługiwać za pomocą kilku metod:
  - Standard języka C++ przewiduje zgłoszenie wyjątku `bad_alloc`, gdy nie powiedzie się `new`.
  - Wiele kompilatorów nie jest jeszcze zgodnych z tym standardem, `new` w przypadku niepowodzenia zwraca 0.
  - Standard języka C++ pozwala używać wersję `new`, która zwraca 0 w przypadku niepowodzenia, ale trzeba to odpowiednio zgłosić kompilatorowi.
  - Można również wykorzystać do obsługi niepowodzeń `new` funkcję `set_new_handler`.

- **Przykład 1**: Program automatycznie zgłasza wyjątek klasy `bad_alloc`. W klasie tej zdefiniowana jest funkcja `what()`, która pozwala wyprowadzić tekst opisujący wyjątek.

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

int main() {
 long rozmiar ;
 int * wsk ;
 int nr_bloku ;

 try {
   cout << "Rozmiar bloku pamieci ? " ;
   cin >> rozmiar ;
   for (nr_bloku=1 ; ; nr_bloku++)
     { wsk = new int [rozmiar] ;
       cout << "Przydzielono blok nr : " << nr_bloku << "\n" ;
     }
   }
   catch (bad_alloc & wyjatek) {
     cout << "Wystapil wyjatek: " << wyjatek.what() << endl;
     exit (1);
   }
   // instrukcje ...
   return 0;
}
```

- **Przykład 2**: Program zgłasza wyjątek typu `bad_alloc`, ale nie chcemy z tego korzystać. Wyłączamy zgłaszanie wyjątku.

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
int main() {
  long rozmiar ;
  int * wsk ;
  int nr_bloku ;
  cout << "Rozmiar bloku pamieci ? " ;
  cin >> rozmiar ;
  for (nr_bloku=1 ; ; nr_bloku++)
    { wsk = new (nothrow) int [rozmiar] ;
      if (wsk == 0) {
        cout << "Brak pamieci\n" ;
        cout << "Zakoncz wykonywanie\n" ;
        exit (1) ;
      }
    cout << "Przydzielono blok nr : " << nr_bloku << "\n" ;
  }
}
```

- **Przykład 3**: Korzystanie z `set_new_handler`

    - Funkcja `set_new_handler()` pozwala określić funkcję, która będzie wywołana w przypadku nieuzyskania przydziału pamięci za pomocą `new`.

```
#include <iostream>
#include <cstdlib>
#include <new>      // dla set_new_handler

int main()
{
// prototyp funkcji wywoływanej, gdy brak pamieci
   void brak_pamieci () ;
// set_new_handler pozwala zarejestrować procedurę obsługi
   set_new_handler (brak_pamieci) ;

   long rozmiar ;
   int * wsk ;
   int nr_bloku ;
   cout << "Rozmiar bloku pamieci ? " ;
   cin >> rozmiar ;
   for (nr_bloku=1 ; ; nr_bloku++)
      { wsk = new int [rozmiar] ;
        cout << "Przydzielono blok nr : " << nr_bloku << "\n" ;
      }
}

void brak_pamieci ()
{  cout << "Brak pamieci\n" ;
   cout << "Zakoncz wykonywanie\n" ;
   exit (1) ;
}
```

Wyjątki a konstruktory i destruktory

- Konstruktor nie zwraca wartości. Tradycyjne zgłosznie błędu w konstrukorze to:
  - ustawienie nielokalnej flagi i oczekiwanie, że użytkownik sprawdzi jej wartość
  - zwrócenie częściowo poprawnie utworzonego obiektu i oczekiwanie, że użytkownik sprawdzi poprawność tego obiektu

Obsługa wyjątków w konstruktorze

- C++ gwarantuje, że przy wychodzeniu z obszaru zasięgu dla wszystkich obiektów, których *konstruktory zostały wykonane do końca*, wykonane zostaną destruktory.
- Jeśli konstruktor nie zostanie wykonany do końca i zostanie wyrzucony wyjątek, to dla danego obiektu nie zostanie wywołany odpowiadający mu destruktor.
- Jeśli w konstruktorze przydzielana jest pamięć, destruktor nie będzie mógł jej zwolnić.
- Rozwiązanie najprotsze:
  - przechwytywanie wyjątków w konstruktorze i tam zwalnianie zasobów

Obsługa wyjątków w destruktorze

- Należy unikać wyrzucania wyjątków w destruktorach. Jeśli jest niezbędny, musi być obsłużony w destruktorze.

Błędy we-wy

Stan strumienia

- Z każdym strumieniem wejścia lub wyjścia jest związany jego *stan*. Właściwe ustawienie i testowanie stanu strumienia umożliwia obsługę błędów i warunków niestandardowych.
- Stan strumienia jest reprezentowany za pomocą zbioru flag (znaczników):

Nazwa bitu	Znaczenie
`goodbit`	`goodbit` = 0 oznacza, że wszystko jest w porządku (nie jest ustawiony żaden z pozostałych bitów)
`eofbit`	Bit ustawiany po napotkania końca pliku
`failbit`	Bit ustawiany wskutek odczytania danych innego typu niż oczekiwany, również podczas problemów przy zapisywaniu danych, można nadal używać strumienia – błąd dotyczy ostatniej operacji
`badbit`	Bit ustawiany w przypadku podejrzenia uszkodzenia strumienie (na przykład problemów z odczytem pliku), nie można już używać strumienia, błąd krytyczny

- Wartości znaczników moża odczytywać i ustawiać za pomocą funkcji składowych klasy `ios`:

Nazwa funkcji	Działanie
`good()`	Zwraca `true`, gdy wszystko jest w porządku (wszystkie bity stanu strumienia są wyzerowane)
`eof()`	Zwraca `true`, jeśli napotlany zostanie oniec pliku (ustawiony `eofbit`)
`fail()`	Zwraca `true`, jeśli wystąpi błąd (ustawiony `failbit`)
`bad()`	Zwraca `true`, jeśli wystąpi błąd krytyczny (ustawiony `badbit`)
`rdstate()`	Zwraca stan znaczników strumienia
`clear(int=0)`	Czyści wszystkie bity stanu strumienia i ustawia je zgodnie z podanym argumentem
`setstate(state)`	Ustawia bity określone w argumencie

Przykłady:

`f1` oznacza strumień

```
f1.clear(); // wyczyści wszystkie bity stanu

f1.clear(badbit); // ustawi bit badbit, zaś wszystkim innym przypisze 0
f1.clear(badbit | f1.rdstate()); // ustawi bit badbit,
                        // wszystkie inne pozostawi nie zmienione
```

- Znaczniki są zdefiniowane w klasie ios_base (dawniej ios). Pełna nazwa znacznika to:

  `std::ios_base::eofbit`
  `std::ios::eofbit` (ze względów zgodności wstecz)

Przeciążone operatory ( ) oraz !

- Do badania stanu strumienia zdefiniowano przeciążone operatory `()` oraz `!`.
- Jeśli `f1` oznacza strumień, to `(f1)` :
  - przyjmuje wartość *prawda* , jeśli żaden z bitów błędu nie jest ustawiony (`good()` zwraca 1)
  - w przeciwnym wypadku przyjmuje wartość *nieprawda*
- Jeśli `f1` oznacza strumień, to `!f1` :
  - przyjmuje wartość *nieprawda* , jeśli jeden z bitów błędu jest ustawiony (`good()` zwraca 0)
  - w przeciwnym wypadku (żaden z bitów błędu nie jest ustawiony) przyjmuje wartość *nieprawda*

Badanie stanu strumienia

- Sprawdzenie, czy można utworzyć plik

```
ofstream wyjscie ("wynik", ios::out|ios::binary);
if (!wyjscie)
  { cout << "Nie mozna utworzyc pliku \n" ; exit (1) ; }
```

- Potwierdzenie, że zakończono wczytywanie danych z powodu napotkania końca pliku

```
while (cin >> liczba)
  {
    suma += liczba;
  }
if (cin.eof())
  cout << "Zakonczono wczytywanie danych : koniec pliku"<< endl;
```

- Aby można było wczytywać nowe dane ze strumienia `cin`, trzeba wyczyścić bit `eof`:

```
while (cin >> liczba)
{
  suma += liczba;
}
if (cin.eof())
  cout << "Zakonczono wczytywanie danych : koniec pliku"<< endl;
// ...
// instrukcje
// ...
cout << "Nowe dane:" << endl;
cin.clear();
while (cin >> liczba)
{
  suma += liczba;
}
```

Operacje we/wy a wyjątki

- Strumienie we/wy domyślnie nie generują wyjątków. Istnieje jednak możliwość określenia dla każdego bitu stanu czy jego ustawienie ma spowodować automatycznie wygenerowanie wyjątku. Służy do tego funkcja składowa `exceptions()`. Wygenerowany wyjątek jest obiektem klasy `ios_base::failure`.

- Przykłady:

```
cout.exceptions(ios::eofbit);     // generuj wyjątek dla bitu końca pliku
cout.exceptions(ios::eofbit | ios::failbit | ios::badbit);
                                  // generuj wyjątek dla wszystkich bitów stanu
```

- Przykład

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

int main()
{
  cin.exceptions(ios::failbit);
  cout.precision(2);
  cout << showpoint << fixed;
  cout << "Wpisz liczby: ";
  double suma = 0.0;
  double liczba;
  try {
    while (cin >> liczba)
    {
      suma += liczba;
    }
  }
  catch(ios::failure & b)
  {
     cout << b.what() << endl;
     cout << "Wyjatek we-wy - Koniec pliku? \n";
  }

  cout << "Ostatnia wpisana liczba= " << liczba << "\n";
  cout << "Suma = " << suma << "\n";
  return 0;
}
```

- Wyjątek zostanie zgłoszony również wtedy, kiedy w danych wejściowych wystąpi błąd. Chcąc rozróżnić powód zgłoszenia wyjątku należałoby zbadać stan strumienia. Na przykład:

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

int main()
{
  cout.precision(2);
  cout << showpoint << fixed;
  cout << "Wpisz liczby: ";
  double suma = 0.0;
  double liczba;
  try {
    while (cin >> liczba)
    {
      suma += liczba;
    }
    if (!cin.eof()) {
      throw ios::failure ("blad podczas czytania");
    }
  }
  catch(ios::failure & b)
  {
    cout << "Wyjatek we-wy: "<< b.what() << endl;
    return 1;
  }
  cout << "Ostatnia wpisana liczba= " << liczba << "\n";
  cout << "Suma = " << suma << "\n";
  return 0;
}
```