Wydział Elektroniki
i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Programowanie obiektowe
Wyjątki

Krzysztof Gracki
kgr@ii.pw.edu.pl

pok. 312

Politechnika
Warszawska

PW

Zagadnienia

- Mechanizm biblioteki ANSI-C **<setjmp.h>**
- Identyfikowanie wyjątków w C++
- Mechanizm obsługi wyjątków w C++
- Przykład w C++
- Przekazywanie wyjątku do bloku obsługi
- Wybór bloku obsługi
- Reemisja wyjątku
- Specyfikacja wyjątków; słowo kluczowe **noexcept**
- Akcesoria standardowe
- Usługi **assert(),exit(),abort()**
- Standardowe klasy błędów w **<stdexcept>**
- Akcesoria w pliku **<exception>**
- Przyczyny aktywacji **terminate()**
- Przykład **unexpected()**
- Przykład użycia **uncaught_exception()**
- Blok **try** funkcyjny

PW

Obsługa błędów

Proces obsługi błędów nie jest łatwym zadaniem!

- ustawianie kodów błędu (globalne)
- funkcje
  - sprawdzanie parametrów
  - zwracanie wartości
- klasy/obiekty
- wyjątki – nielokalne przekazanie sterowania

Podstawy

Wyjątek: stan programu lub jego otoczenia, w którym kontynuacja obliczeń **jest niemożliwa**

- próba dzielenia przez 0
- wywołanie **`sqrt(x)`** z argumentem ujemnym
- próba pobrania elementu z pustego stosu
- niepowodzenie alokacji dynamicznej
- brak miejsca na nośniku przy próbie zapisu
- brak pliku otwieranego do odczytu
- ...

Różne programy - różne traktowanie sytuacji

- obliczenia z klasą **`Fraction`** - dzielenia przez 0 OK
- dziedzina liczb zespolonych - **`sqrt(-2)`** OK

Podstawy (cd1)

Kategorie wyjątków

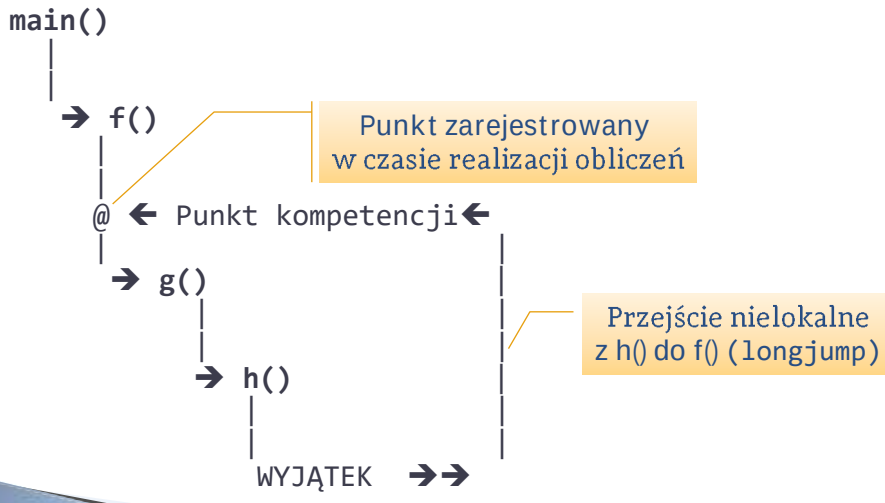
- niekrytyczne - można ich uniknąć badając stan programu
- krytyczne - pojawianie się jest nieprzewidywalne

Mechanizm obsługi wyjątków zwykle **nie rozróżnia** tych kategorii (kategorie są nieostre)

Wyjątki i sterowanie nielokalne

- Funkcja wykrywająca wyjątek zazwyczaj nie potrafi go obsłużyć (np. funkcja biblioteczna)
- Ten sam rodzaj wyjątku może wymagać różnych reakcji, zależnie od kontekstu wystąpienia
- Potrzebny mechanizm ustanawiania „punktu kompetencji" i nielokalnego przekazywania sterowania

Podstawy (cd2)



Mechanizm biblioteki ANSI-C <setjmp.h>

Biblioteka definiuje akcesoria:

- **jmp_buf**
  Struktura danych i typ do zarejestrowania stanu obliczeń
- **int setjmp(jmp_buf B);**
  Funkcja do rejestracji "punktu(ów) kompetencji" w buforze B; zwraca 0
- **void longjmp(jmp_buf B, int r);**
  Funkcja realizujaca przejście nielokalne; wykorzystując przekazany jej bufor B symuluje powrót z funkcji setjmp() ale z przekazaniem wyniku r. Wartość r jest klasyfikatorem przyczyny przejścia nielokalnego – identyfikuje wyjątek.

Mechanizm ANSI-C –przykład


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

class A { /* ... */ };

void f(), g(), h();
jmp_buf jb;
int w=3;
void main()
{ //  A a(1);
  cout<<"Start main(): w="<<w<<'\n';
  f();
  cout<<"Koniec main()\n";
}
```

Udawanie generowania wyjątkiu (3)

```
void f()
{ //A b(2);
  cout<<"Start  f()\n";

  switch(setjmp(jb))// Punkty kompetencji
  { case 0: g();    // Początek obliczeń "iebezpiecznych"
            break;
    case 1: cout<<"Wyjatek 1\n"; break;
    case 2: cout<<"Wyjatek 2\n"; break;
    default:cout<<"Inne wyjatki\n";
            /*???*/
            break;
  }
  // ...
  cout<<"Koniec f()\n";
}
```

Mechanizm ANSI-C –przykład (cd2)


```
void g()
{ //A c(3);

  cout<<"Start  g()\n";
  if(w==1)
  { cout<<"longjmp(jb, 1)\n";
    longjmp(jb, 1);
  } else
  if(w==2)
  { cout<<"longjmp(jb, 2)\n";
    longjmp(jb, 2);
  }
  h();
  // ....
  cout<<"Koniec g()\n";
}
```

```
void h()
{ //A d(4);

cout<<"Start  h()\n";
  if(w==3)
  { cout<<"longjmp(jb, 3)"
        << endl;
    longjmp(jb, 3);
  }
  cout<<"Koniec h()"
      <<endl;
}
```

Wygeneruj wyjątek 3)

Mechanizm ANSI-C –przykład (cd3)

Wynik wykonania (zależy od zmiennej w)

```
Start main(): w=0
Start  f()
Start  g()
Start  h()
Koniec h()
Koniec g()
Koniec f()
Koniec main()
```

```
Start main(): w=1
Start  f()
Start  g()
longjmp(jb, 1)
Wyjatek 1
Koniec f()
Koniec main()
```

```
Start main(): w=3
Start  f()
Start  g()
Start  h()
longjmp(jb, 3)
Inne wyjatki
Koniec f()
Koniec main()
```

Uwagi:

- "Obsługa" wyjątku sprowadzona do komunikatu
- Przypadek `w==0` - wykonanie bez zgłaszania wyjątków
- Funkcja `f()` przygotowana do przechwycenia dowolnego wyjątku (zawiera sekcję `default`); miejsce zaznaczone `/*???*/` powinno zawierać kod obsługi i, być może, przekazanie sterowania do innego (istotnie kompetentnego) punktu obsługi

Wady mechanizmu ANSI-C

- Mechanizm niskiego poziomu - niestrukturalny
- Identyfikacja wyjątków przez numerację - utrudnia wprowadzenie klasyfikacji, np. organizacji hierarchicznej
- Brak bezpośrednio dodatkowych informacją o wyjątku (trzeba korzystać z pośrednictwa zmiennych globalnych).
- **`longjmp()`** w wersji ANSI-C nie wspiera semantyki tworzenia/usuwania obiektów języka C++: podczas przekazywania sterowania nielokalnie do punktu kompetencji **obiekty lokalne na stosie nie są poddawane destrukcji**

Mechanizm ANSI-C z `klasami`

Przykład z obiektami lokalnymi klasy A

(zdjęte komentarze z deklaracji)

```
A(int) 1
Start main(): w=3
A(int) 2
Start  f()
A(int) 3
Start  g()
A(int) 4
Start  h()
longjmp(jb, 3)
Inne wyjatki
Koniec f()
~A() 2
Koniec main()
~A() 1
```

```
// Z akcją naprawczą
Start main(): w=3
Start  f()
Start  g()
Start  h()
longjmp(jb, 3)
Inne wyjatki
Start  g()   <= raz jeszcze
Start  h()
Koniec h()
Koniec g()
Koniec f()
Koniec main()
```

Identyfikowanie wyjątków w C++

- Numeracja wyjątków (jak wg **`setjmp()`, `longjmp()`**)
- Przy pomocy obiektów dowolnego typu
  - Przy pomocy dziedziczenia względem klas wyjątków dostępnych w bibliotece standardowej
  - Przy pomocy hierarchii typów specjalizowanych dla danej aplikacji (klasyfikacja bierna)
  - Przy pomocy obiektów aktywnych (np. wyjątek powoduje uruchomienie agenta komunikacji z serwerem producenta oprogramowania)

Mechanizm obsługi wyjątków w C++

Trzy konstrukcje składniowe współtworzą zewnętrzną manifestację mechanizmu obsługi wyjątków:

- Blok **`try`** - rozpoczyna obliczenie prowadzące (być może) do powstania sytuacji wyjątkowej
- Bloki **`catch`** - odpowiedzialne za obsługę konkretnego typu lub zbioru typów wyjątków; blok **`catch`** jest "punktem kompetencji". Bloki **`catch`** występują zawsze bezpośrednio po bloku **`try`**
- Instrukcja **`throw`** - zgłasza wyjątek i uruchamia mechanizm obsługi.

Przykład w C++ (cd1)


```
void f()  {
  A b(2);
  try {    // Punkt kompetencji
     g();  // Obliczenia "niebezpieczne"
  }
  catch(int exnum)// Wyjątki typu int
  { cout<<"Wyjatek "<<exnum<<endl;
  }
  catch(...) // Inne typy wyjątków
  { cout<<"Inne wyjatki\n";
    w = 0; g();
   // w=0; reprezentuje akcję naprawczą
  }
  // ...
}
```

```
void g()
{ A c(3);
  if(w==1 || w==2) {
    cout<<"throw "<<w<<endl;
    throw w;
  }
  h();
  // ....
}
void h()
{ A d(4);
  if(w==3) {
    cout<<"throw char*\n";
    throw "ERROR";
  }
}
```

```
Wynik wykonania:
A(int) 1
A(int) 2
A(int) 3
A(int) 4
throw char*
~A() 4
~A() 3
Inne wyjatki
A(int) 3
A(int) 4
~A() 4
~A() 3
~A() 2
~A() 1
```

Przykład w C++ (cd3)

Uwagi

- Funkcja h() zgłasza wyjątek:
  `throw "ERROR";`
- Przechwycenie przez blok obsługi
  `catch(...){/* ... */}`
- Brak tego bloku obsługi => wywołanie funkcji
  `std::terminate();`
  (ta domyślnie wywołuje **`abort()`** kończącą program komunikatem „`abnormal program termination`„ lub podobnym)
- Koniec przez **`terminate()`** - destruktory nie muszą być aktywowane (decyzja kompilatora).

Niektóre kompilatory na brak bloku obsługi `catch(...)`, reaguje tak:

```
A(int) 1
A(int) 2
A(int) 3
A(int) 4
throw char*

terminate called after throwing
an instance of 'char const*'

This application has requested
the Runtime to terminate it in
an unusual way. Please contact
the application's support team
for more information.
```

Przekazywanie wyjątku do bloku obsługi

- Wyjątek może być reprezentowany obiektem dowolnego typu z operacją kopiowania (typy wbudowane, klasy z publicznym konstruktorem kopiującym)
- Blok **`catch`** parametryzowany jak funkcja jednoparametrowa
- Wyjątek przekazywany do bloku przez wartość, odniesienie albo jako wskazanie (wg zwykłych reguł)
- Blok parametryzowany przez wartość otrzyma *kopię* obiektu (albo podobiektu) użytego w **`throw`**.
- Blok parametryzowany przez odniesienie nie wymaga kopiowania
- Obiekt w **`throw`** jest tymczasowy (rezyduje gdzieś w pamięci statycznej wg decyzji twórców kompilatora)

Wybór bloku obsługi

- Po zgłoszeniu wyjątku **`throw`** *`wyrażenie`*`;` poszukiwany jest blok obsługi wrażliwy na zgłoszony typ wyjątku
- Bloki obsługi brane są pod uwagę **w kolejności definiowania** w sąsiedztwie bloku **`try`**
- Wybór wg następujących reguł (C i T typy parametru w **`catch`** i wyrażenia w **`throw`** po zignorowaniu zewnętrznych kwalifikatorów **`const`** i/lub **`volatile`**):
  - C = X albo X& i T=X, (X jest pewnym typem)
  - C = X albo X& i T=Y, (X jest jednoznaczną, publiczną klasą bazową Y)
  - C = X* i T = Y* (można wykonać standardową konwersję wskazań od Y* do X*)
  - Typ tablicowy `X[]` jest tożsamy z `X*`
- **Kolejności definiowania bloków obsługi**:
  - bloki wrażliwe na klasy bazowe **po** blokach wrażliwych na klasy pochodne;
  - blok `catch(...){/* */}` zawsze **na końcu**.

Wybór bloku obsługi (cd1)


```
//Przykład
class A{/* ... */ };
class B: public A { /* ... */ };
void fun()
{ try            // Punkt kompetencji
  { go();  }
  catch(A x)  // Przechwyci wyjątki A i B
  { /* x inicjowany przez A::A(A&)) */ }
  catch(B& x){}   // Nigdy nie wybierany
  catch(...)// Inne, bez A i B
  { // ........
    throw;  // Ponowne zgłoszenie tego
            // samego wyjątku
  }
}
```


Słowo kluczowe noexcept [dawniej throw()]

- Funkcja deklarująca **noexcept** zapowiada, że nie będzie zgłaszać wyjątku; gdyby jednak wyjątek wystąpił, to będzie potraktowany "brutalnie" przez wywołanie funkcji **terminate()**.

```
void trim(vector<Fraction>& vf, int div) noexcept //≡ noexcept(true)
// Nie będzie wyjątków dzielenia przez div==0
{ for(int i; i<vf.size(); ++i) vf[i] /= div;}
```

- Funkcja może być zadeklarowana jako **warunkowo** nie zgłaszająca wyjątków; specyfikacja taka zwykle odwołuje się do właściwości parametru(ów) szablonu:

```
template<class T>
void fun(T& tr, int sel) noexcept(T::gun(0))
// Wyjątek możliwy, jeśli T::gun(0) zgłasza
{ // ...
  tr.serwis(T::gun(sel));
}
```

Reemisja wyjątku

- Ponowne zgłoszenie wyjątku - instrukcja **`throw;`** (bez wyrażenia); używa się, jeżeli blok obsługi nie jest w stanie doprowadzić obsługi do końca
- Dopuszczalne tylko **wewnątrz bloku `catch`**
- Będzie zgłoszony **ten sam wyjątek**, który spowodował aktywację bloku obsługi
- Jeżeli wyjątek dostępny w aktualnym bloku obsługi został przekazany przez wartość (przez konstruktor kopiujący), to nie wpływa to na sposób reemisji (np. skopiowany został podobiekt bazowy, ale reemisja dotyczyć będzie **pełnego** obiektu pochodnego)

Specyfikacja wyjątków (removed in C++17)

Oficjalna nazwa: "Dynamic exception specification"

```
void fun() throw (A,B);
```
fun() może zgłosić wyjątek A lub B i żadnego innego

```
void fun(int) throw ();
```
fun(int) nie będzie zgłaszać żadnych wyjątków

```
void fun(char*);
```
fun(char*) może zgłaszać dowolne wyjątki.

- Specyfikacja wyjątków nie jest brana pod uwagę przy rozróżnianiu funkcji w zbiorze przeciążonym
- Kontrola zgodności specyfikacji z faktycznym zachowaniem – w czasie wykonania programu
- Nie jest błędem użycie w funkcji ze specyfikacją braku wyjątków funkcji potencjalnie zgłaszającej dowolny wyjątek (pod warunkiem niezgłaszania przez nią wyjątków)

Wyjątki w MS Visual Studio

Specyfikacja wyjątku	Znaczenie

`noexcept`
`noexcept(true)`
`throw()`
	Visual Studio 2017 w wersji 15.5 lub nowszej: w /std:c++17 `noexcept`, `noexcept(true)` i `throw()` **są równoważne**.
	W /std:c++17 `throw()` jest aliasem dla elementu noexcept(true).
	W /std:c++17 i później, gdy wyjątek jest zgłaszany z funkcji zadeklarowanej przy użyciu dowolnej z tych specyfikacji, `std::terminate` jest wywoływany zgodnie z wymaganiami standardu C++17.

`noexcept(false)`
`throw(...)`
`Brak specyfikacji`
	Funkcja może zgłosić wyjątek dowolnego typu.

`throw(type)`	(C++14 i starsze) Funkcja może zgłosić wyjątek typu `type`. Kompilator akceptuje składnię, ale interpretuje ją jako `noexcept(false)`. W /std:c++17 trybie i nowszych kompilator wyświetla **ostrzeżenie C5040**.

https://docs.microsoft.com/pl-pl/cpp/cpp/exception-specifications-throw-cpp?view=msvc-170

Klasa auto_ptr [C++11: deprecated !]

- Obiekty dynamiczne, tworzone poprzez operator new, nie podlegają gwarantowanej destrukcji podczas zwijania stosu
- Trzeba zastosować klasę pomocniczą – „opakowanie" wskaźnika" **auto_ptr** – wymuszającą automatyzm destrukcji obiektów dowiązanych
- Inne „opakowania": **unique_ptr, shared_ptr**

Kod nieodporny na wyjątki

```
void f()
{
  string * p = new string;
  fun(p); //Może być wyjątek
  // Kod może być zerwany
  delete p;//Ubytek pamięci?
}
```

Kod z auto_ptr

```
void f()
{ auto_ptr<string>
    p(new string);
  fun(p); //Może być wyjątek
  // Zwolnienie pamięci
  // automatyczne
}
```

Mechanizm RAII

(ang. Resource Acquisition Is Initialization)

- Kontrola od utworzenia obiektu (w konstruktorze) i zagwarantowanie automatycznego zwolnienia (w destruktorze)
- Zasób do zwolnienia musi być przekazany w konstruktorze obiektu

```
auto_ptr<std::string> p(new std::string);
```

- Można zwolnic zasób w destruktorze obiektu

Klasa auto_ptr (plik <memory>)

```
template<typename T>
class auto_ptr
{ T* p; // Prywatne
public:
  explicit auto_ptr(T* pp = nullptr) noexcept
     : p(pp) {}
  auto_ptr(auto_ptr& a) noexcept
     : p(a.release()) {}
  auto_ptr& operator=(auto_ptr& a) noexcept
  { reset(a.release());
   return *this;
  }
 ~auto_ptr() { delete p; }
  T& operator* () const noexcept  {
return *p; }
  T* operator->() const noexcept {
return p;  }
  T* get() const noexcept { return p; }
  T* release() noexcept
  { T* tmp = p; p = nullptr;
      return tmp;
  }
  void reset(T* pp = nullptr) noexcept
  { if(pp != p)
   {  delete p;
      p = pp;
   }
  }
  // ....
};
```

Klasa szablonowa **shared_ptr<T>** jest lepszym rozwiązaniem. Wewnętrznie stosowane jest zliczanie referencji do obiektu zarządzanego (podobnie do reprezentacji grupowej String).

Akcesoria standardowe

- Środowisko C++ zawiera predefiniowane klasy wyjątków i dodatkowe funkcje usługowe. Standardowe wyjątki (dziedziczą wg exception):

```
// Wyjątek      Zgłaszany przez
// ------------------------------------
bad_alloc       operator new
bad_cast        dynamic_cast<Typ>(exp)
bad_typeid      operator typeid
bad_exception   spec. wyjątków funkcji
```

Akcesoria standardowe (cd1)


```
 // Wewnątrz namespace std
class exception
{ public:
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
    // - - - - - -
    // instaluje tekst zależny od implementacji
    exception() throw();

    // instaluje tekst wg argumentu
    exception(const char *const&) throw();      //EXT

    // Jak wyżej, ale bez alokacji pamięci
    exception(const char *const&, int) throw(); //EXT
};
```

Akcesoria standardowe (cd2)

- Funkcje do zarządzania zachowaniem **terminate()** i **unexpected()**:

```
typedef void (*PROC)();
// PROC: wskazanie na procedurę bezparametrową
PROC set_terminate(PROC);
PROC set_unexpected(PROC);
```

- Można podmienić standardowe zachowanie **terminate()** i **unexpected()** na własne, np.:

```
set_unexpected(MojaFun)
```

- Nowa reakcja na niespodziewane wyjątki wg funkcji **MojaFun()**; **set_unexpected()** zwraca wskazanie na dotychczas obowiązującą funkcję.
- Funkcja **set_terminate(PROC)** działa analogicznie.

Usługi assert(),exit(),abort()

- Makro **assert(warunek)**:
  pozwala kontrolować warunki poprawności wykonania podczas uruchamiania programu (DEBUG)
- Funkcja **exit(int ec)**:
  - Przeprowadza destrukcję obiektów statycznych (obiekty lokalne są porzucane bez destrukcji)
  - Zamyka (z uwzględnieniem buforów) otwarte pliki / strumienie
  - Kończy wykonanie programu, przekazując argument **ec** do systemu jako kod powrotu
- Funkcja **abort()**:
  kończy program bez finalizacji obiektów statycznych i strumieni; generuje komunikat "abnormal program termination" i zwraca kod (na ogół 3) zakończenia procesu.

Standardowe klasy błędów w <stdexcept>


```
namespace std
{
   class logic_error;      //:public exception
  class domain_error;     //:public logic_error
  class invalid_argument; //:public logic_error
  class length_error;     //:public logic_error
  class out_of_range;     //:public logic_error
  class runtime_error;    //:public exception
  class range_error;      //:public runtime_error
  class overflow_error;   //:public runtime_error
  class underflow_error;  //:public runtime_error
}
```

logic_error: błąd możliwy do wykrycia (niekrytyczny)
runtime_error: zazwyczaj błąd krytyczny

Przykład (cd)


```
int main()
{
  set_unexpected(uhandler);
    try
      {
        B obj, * bp = 0;
        //typeid(*bp);                  // (1): bad_typeid
        //D &d = dynamic_cast<D&>(obj); // (2): bad_cast
        //char *cp=new char[1000000000];// (3): bad_alloc
        //riskfun();                    // (4): bad_exception
        throw logic_error("strange!");  // (5): strange!
      }
      catch (exception& e)
      {
        cerr << "exception: " << e.what() << endl;
      }
      return 0;
}
/* (1): exception: Attempted a typeid of NULL pointer!
       (2): exception: Bad dynamic_cast!
       (3): exception: bad allocation
       (4): exception: bad exception
       (5): exception: strange!  */
```

```
void uhandler() {
  cerr << "uhandler" << endl;
  throw;
}
```

Przyczyny aktywacji `terminate()`

Funkcja **`terminate()`** jest wołana w stanie "paniki" mechanizmu obsługi wyjątków

- pomiędzy **`throw exp;`** a **`catch()`** zgłaszany jest nowy wyjątek
- nie można znaleźć bloku obsługi dla wyjątku; także zgłoszenie wyjątku w funkcji specyfikowanej **`noexcept`**
- podczas "zwijania" stosu **destruktor** obiektu **zgłasza wyjątek** (por. **`uncaught_exception()`**)
- konstruktor / destruktor obiektu statycznego generuje wyjątek
- próba reemisji wyjątku (**`throw;`**) poza blokiem **`catch`**
- zadziałała domyślna funkcja **`unexpected_handler()`**
- **`unexpected()`** zgłasza wyjątek nie przewidziany w specyfikacji wyjątków i specyfikacja nie zawiera **`std::bad_exception`**

Blok `try` funkcyjny


```
class Buf
{
      char* p;
public:
      explicit Buf(size_t);
      ~Buf();
};
Buf::Buf(size_t n) :p(new char[n]) {}
Buf::~Buf() { delete[] p; }
void fun(Buf& b) { /* ... */ }
static Buf big(100000); // Co z ewentualnym wyjątkiem?
int main()
{
      try
      {
            Buf b(1024); //Ewentualny wyjątek będzie obsłużony
            fun(b);
      }
      catch (...) {/*...abort();*/ }
}
```

Może wyzwolić wyjątek

Blok try funkcyjny (cd1)


```
Rozwiązanie 1 – obsługa w konstruktorze
Buf::Buf(size_t n) :p(0)
{
      try { p = new char[n]; }
      catch (...) {/*...abort();*/ }
}
Wada: nie działa dla składowych const

Rozwiązanie 2 (funkcja usługowa)
class Buf
{
      char* const p;
public:
      explicit Buf(size_t);
      ~Buf();
};
template <typename T> T* alloc(size_t const n)
{
      try { return new T[n]; }
      catch (...) {/*...abort();*/ }
}
Buf::Buf(size_t n) :p(alloc<char>(n)) {}
```


Blok try funkcyjny (cd2)

function-try-block:

try *ctor-initializer*$_{opt}$ *function-body handler-seq*

```
Rozwiązanie 3 (blok try funkcyjny)
// ...
Buf::Buf(size_t n)
try :p(new char[n]) // reszta listy init
{ /* ciało konstruktora, tu puste */
}
catch (...)
{/*...abort();*/
}

Wersja ostateczna(ze specyfikacją wyjątków)
class Buf
{
  char* const p; // Inne składowe
public:
  explicit Buf(size_t) throw();
  ~Buf() throw();
};
```

```
// Przykład bloku
// try funkcji globalnej

void f(int n)
try
{ // ...
  if (n == -1) throw
              bad_alloc();
}
catch (exception& e)
{
   cout << e.what();
};
```

Wyjątki - podsumowanie

- Zgłoszenie wyjątku w konstruktorze
  - obsługa przez blok **try funkcyjny**
  - **nie zostanie wywołany destruktor**
  - obiekty muszą byś w pełni skonstruowane
- Aktualnie obsługiwany wyjątek jest ponownie zgłaszany, jeśli osiągnie koniec procedury obsługi bloku **try** konstruktora lub destruktora.

```
struct A {
  A() { throw exception("exc. w A()"); }
  ~A(){ std::cerr << "~A()"; }
};

int main()
{ try   { B b;    }
  catch (exception e) {
   cerr << "catch main:" << e.what();
  }
  return 0;
}
```

```
struct B { A a;
 B() try : a() { cout << "B()"; }
     catch (exception e) {
    /* przechwycenie wyj. A */
    cerr << "catch B:" <<e.what()
         << endl;;
    }
    ~B() { std::cerr << "~B()"; }
};
```

```
catch B:exc. w A()
catch main :exc. w A()
```