Лекція 10

Виняткові ситуації

У цій лекції

10.1. Механізм обробки виняткових ситуацій
10.2. Тонкості обробки виняткових ситуацій
10.3. Стандартні виняткові ситуації

Одним з найбільш яскравих утілень принципу об'єктно-орієнтованого програмування є *механізм обробки виняткових ситуацій* у мові С++. У ході виконання програми можуть виявитися різні помилки. Вони можуть бути позв'язані з неправильним програмуванням (наприклад, вихід індексу масиву за межі припустимого чи переповнення пам'яті), а іноді їхня причина не залежить від програміста (скажемо, розрив зв'язку при мережевому з'єднанні). У кожній з цих ситуацій реакція програми непередбачена. Іноді вона завершує виконання, і лише після закінчення деякого інтервалу часу починають позначатися наслідку помилки, а частіше програма негайно припиняє роботу, піддаючи ризику дані, що знаходяться в пам'яті чи у файлі. Якщо не передбачити акуратне завершення роботи, використовуючи *обробку виняткових ситуацій*, результати можуть виявитися неприємними.

В подальшому ми будемо називати *винятковою ситуацією* будь-яку подію, що вимагає особливої обробки. При цьому зовсім неважливо, чи є ця подія фатальною чи простою помилкою. Перевірка умов, що описують виняткову ситуацію, і реакція на її виникнення називається *обробкою виняткової ситуації*. Ця задача покладається на *оброблювача виняткової ситуації*.

10.1. Механізм обробки виняткових ситуацій

Обробка виняткових ситуацій у мові С++ є об'єктно-орієнтованою. Це значить, що виняткова ситуація є *об'єктом*, що генерується при виникненні незвичайних умов, передбачених програмістом, і передається *оброблювачу*, що неї *перехоплює*. Об'єктом, що описує природу виняткової ситуації, може бути будь-як сутність — літерал, рядок, об'єкт класу, число і т.д. Не слід думати, що виняткова ситуація обов'язково повинна бути об'єктом якого-небудь класу.

10.1.1. Обробка виняткових ситуацій

В основі обробки виняткових ситуацій у мові С++ лежать три ключових слова: `try`, `catch` і `throw`. Якщо програміст підозрює, що визначений фрагмент програми може спровокувати помилку, він повинний занурити цю частину коду в блок `try`. Необхідно мати на увазі, що зміст помилки (за винятком стандартних ситуацій) визначає сам програміст. Це значить, що програміст може задати будь-яку умову, що приведе до створення виняткової ситуації. Після цього необхідно вказати, у яких умовах варто генерувати виняткову ситуацію. Для цієї мети призначене ключове слово `throw`. І нарешті, виняткову ситуацію потрібно перехопити й обробити в блоці `catch`. Ось як виглядає ця конструкція.

```
try
{
    // Тіло блоку try
    if(умова)throw виняткова_ситуація
 }
 catch(тип1 аргумент)
 {
    // Тіло блоку catch
 }
 catch(тип2 аргумент)
 {
    // Тіло блоку catch
 }
 .
 .
 .
 catch(типN аргумент)
 {
    // Тіло блоку catch
 }
```

Розмір блоку `try` не обмежений. У нього можна занурити як один оператор, так і цілу програму. Один блок `try` можна зв'язати з довільною кількістю блоків `catch`. Оскільки кожен блок `catch` відповідає окремому типу виняткової ситуації, програма сама визначить, який з них виконати. У цьому випадку інші блоки `catch` не виконуються. Кожен блок `catch` має аргумент, що приймає визначене значення. Цей аргумент може бути об'єктом будь-якого типу. Якщо програма виконана правильно й у блоці `try` не виникло жодної виняткової ситуації, усі блоки `catch` будуть зігноровані. Якщо в програмі виникла подія, що програміст вважає небажаним, оператор `throw` генерує виняткову ситуацію. Для цього оператор `throw` повинний знаходитися усередині блоку `try` або усередині функції, викликуваної усередині блоку `try`.

Якщо в програмі виникла виняткова ситуація, для якої не передбачені перехоплення й обробка, викликається стандартна функція `terminate()`, що, у свою чергу, викликає функцію `abort()`. Утім, іноді виняткова ситуація не є небезпечної. У цьому випадку можна виправити помилку (наприклад, привласнити нульовому знаменнику ненульове значення) і продовжити виконання програми.

Розглянемо найпростіший приклад.

Обробка виняткової ситуації

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

int main()
{
    int n = 10, m = 0;
    printf("Початок\n");

    try
    {
      printf("У блоці try\n");
      if(m==0) throw "Divide by zero"; else n=n/m;
      printf("Подальша частина блоку не виконується!");
    }
    catch (const char* s)
    {
      printf("%s\n",s);
    }
    printf("Кінець\n");
    return 0;
 }
```

Ця програма виводить на екран наступні рядки.

```
Начало
Усередині блоку try
Розподіл на нуль
Кінець
```

Простежимо за потоком керування при виконанні цієї програми. Спочатку з'являються і ініціалізуються дві целочисельні перемінні (одна з них дорівнює нулю). Потім виводиться повідомлення про початок виконання програми, і потік керування входить у блок `try`. Після виводу рядка повідомлення про вхід у блок `try`, потік керування переходить до перевірки рівності `m==0`. Оскільки ця рівність є істиною, генерується виняткова ситуація (у даному випадку — константний рядок). Керування негайно передається блоку `catch`, аргументом якого є константний символьний вказівник, ігноруючи всі інші оператори в блоці `try`. У цій програмі блок `catch` не робить жодних спроб виправити помилку. Замість цього він просто видає повідомлення — рядок, отриманий як аргумент — і передає керування оператору, що слідує за блоком. На закінчення функція `printf()` виводить на екран рядок `Кінець`, і програма завершує свою роботу.

Тип виняткової ситуації повинний збігатися з типом аргументу розділу `catch`. Поглянемо, що відбудеться, якщо цією умовою зневажити.

Порушення угоди про тип виняткової ситуації

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

 int main()
 {
    int n = 10, m = 0;
```

```
    printf("Начало\n");

    try
    {
      printf("У блоці try\n");
      if(m==0) throw "Розподіл на нуль"; else n=n/m;
      printf("Подальша частина блоку не виконується!");
    }
    catch (const char s)  // Помилка! Необходимо const char* s!
    {
      printf("%s\n",s);
    }
    printf("Кінець\n");
    return 0;
 }
```

У цій програмі ми зробили цілком "природну" помилку — забули поставити зірочку в оголошенні аргументу. Тепер блок `catch` очікує виняткову ситуацію, що представляє собою константний символ, а не вказівник. Ця помилка приводить до аварійного завершення роботи програми.

Покажемо, що відбудеться, якщо виняткова ситуація генерується усередині функції, яка викликається в блоці `try`.

Виняткова ситуація, згенерована усередині функції

```
 #include <iostream>
 using namespace std;
 int Denominator(int);

 int main()
 {
    int n = 10, m;
    printf("Початок\n");

    try
    {
      printf("У блоці try\n");
      m=Denominator(0);
      printf("Подальша частина блоку не виконується!");
    }
    catch (const char* s)
    {
      printf("%s\n",s);
    }
    printf("Кінець\n");
    return 0;
 }

 int Denominator(int i)
 {
    if(i==0)throw "Розподіл на нуль";
    return i;
 }
```

У цій програмі виняткова ситуація генерується у функції `Denominator()`, яка викликається в блоці `try`. Завдяки цьому результати роботи програми цілком збігаються з попередніми.

Якщо блок `try` знаходиться усередині функції, обробка виняткової ситуації виконується при кожнім виклику.

Розміщення блоку try усередині функції

```
 #include <iostream>

 using namespace std;
 int Denominator(int);

 int main()
```

```
{
   int n = 10, m;
   printf("Початок\n");
   m=Denominator(0);
   n = Denominator(11);
   printf("Кінець\n");
   return 0;
}

int Denominator(int i)
{
   printf("У функції Denominator\n");
   try
   {
     printf("У блоці try\n");
     if(i==0) throw("Розподіл на нуль!");
     if(i>10) throw 10;
     printf("Подальша частина блоку не виконується!");
   }
     catch (const char* s)
   {
     printf("%s\n",s);
   }
     catch (int n)
   {
     printf("Чисельник більше %d\n",n);
   }
     return i;
}
```

На екрані з'являться наступні рядки.

```
Усередині функції Denominator
У блоці try
Розподіл на нуль!
Усередині функції Denominator
У блоці try
Чисельник більше 10
Кінець
```

У цій програмі передбачене перехоплення двох виняткових ситуацій. Перша з них має тип `const char*` і генерується, коли знаменник дорівнює нулю, а друга — тип `int` і генерується, коли чисельник перевищує 10. Як бачимо, ці виняткові ситуації перевіряються і перехоплюються при кожнім виклику функції `Denominator()`.

Розглянемо тепер приклад, у якому функція `Denominator()` лише генерує виняткові ситуації, а їх обробка здійснюється у функції `main()`.

Окрема обробка виняткових ситуацій

```
#include <iostream>
 using namespace std;
 int Denominator(int);

 int main()
 {
    int n = 10, m;
    printf("Початок\n");

    try
    {
      printf("У блоці try\n");
      m=Denominator(0);
      n = Denominator(11);
      printf("Подальша частина блоку не виконується!");
    }
    catch (const char* s)
    {
```

```
      printf("%s\n",s);
    }
    catch (int n)
    {
      printf("Чисельник більше 10 %d\n",n);
    }
    printf("Кінець\n");
    return 0;
 }

 int Denominator(int i)
 {
    printf("Усередині функції Denominator\n");
    if(i==0) throw("Розподіл на нуль!");
    if(i>10) throw 10;
    printf("Кінець функції Denominator\n");
    return i;
 }
```

Результат демонструє декілька важливих особливостей, властивим функціям, що збуджують, але не обробляють виняткову ситуацію.

```
Початок
Усередині блоку try
Усередині функції Denominator
Розподіл на нуль!
Кінець
```

По-перше, блоки `try` і `catch` нерозривні. Не можна помістити блок `try` у функцію, залишивши блок `catch` у функції `main()`. Необхідно або обробити виняткову ситуацію усередині функції, як це зроблено в більш ранньому прикладі, або перенести обробку в модуль виклику. У першому випадку функція, завершивши обробку, повертає визначене її специфікацією значення, а в другому — *виняткову ситуацію*. Таким чином, можна обійти обмеження мови C++, відповідно до якого функція може повертати лише одне значення, тип якого визначений заздалегідь. По-друге, механізм обробки виняткових ситуацій дозволяє створювати *альтернативні* значення, що повертаються. По-третє, функції можуть генерувати декілька виняткових ситуацій. Збудивши одну з них, вони негайно припиняють своє виконання і повертають виняткову ситуацію в модуль виклику. Необхідно враховувати, що присвоювання `m=Denominator(0)` чи `n=Denominator(11)` у цьому випадку не виконуються.

Представимо тепер ланцюжок викликів функцій.

Ланцюгове генерування виняткових ситуацій: перший варіант

```
#include <stdio.h>

 using namespace std;
 int Check(int);
 int Divide(int, int);

 int main()
 {
  int n = 10, m=0, l;

  printf("Початок\n");
  l = Divide(n,m);
  printf("Кінець\n");

  return 0;
 }

 int Check(int i)
 {
    printf("Усередині функції Check\n");
    if(i==0) throw("Розподіл на нуль усередині функції Check!");
       printf("Кінець функції Check\n");
    return i;
 }

 int Divide(int n, int m)
```

```
{
   printf("Усередині функції Divide\n");

   try{m=Check(m);}
   catch (const char* s)
   {
     printf("%s\n",s);
     return 1;
   }
   printf("Кінець функції Divide\n");
   return n/m;
}
```

Простежимо за передачею виняткової ситуації.

```
Початок
Усередині функції Divide
Усередині блоку try
Усередині функції Check
Розподіл на нуль усередині функції Check!
Кінець
```

При виклику функції `Divide()` перевіряється знаменник `m`. Для цього викликається функція `Check()`. Якщо знаменник дорівнює нулю, усередині цієї функції генерується виняткова ситуація, що має тип `const char*`. Обробка цієї виняткової ситуації усередині функції `Check()` не передбачена, тому вона передається нагору по ланцюжку викликів — функції `Divide()`. Потім керування передається функції `main()`, і виконання програми завершується.

Перенесемо обробку виняткової ситуації у функцію `main()`.

Ланцюгове генерування виняткових ситуацій: другий варіант

```
#include <iostream>

using namespace std;

int Check(int);
int Divide(int, int);

int main()
{
   int n = 10, m=0, l;

   printf("Початок\n");
   try
   {
       printf("Усередині блоку try\n");
          l = Divide(n,m);
   }
   catch (const char* s)
   {
   printf(" %s\n",s);
   }

   printf("Кінець\n");
   return 0;
}

int Check(int i)
{
   printf("Усередині функції Check\n");
   if(i==0) throw("Розподіл на нуль усередині функції main!");
      printf("Кінець функції Check\n");
   return i;
}

int Divide(int n, int m)
{
```

```
    printf("Усередині функції Divide\n");
    m=Check(m);
    printf("Кінець функції Divide\n");
    return n/m;
}
```

Результат роботи цієї функції такий.

```
Початок
Усередині блоку try
Усередині функції Divide
Усередині функції Check
Розподіл на нуль усередині функції main!
Кінець
```

Оскільки усередині функцій `Check()` і `Divide()` обробка виняткової ситуації не передбачена, вона передається в головний модуль, про що свідчить представлена нижче рядок.

```
Розподіл на нуль усередині функції main!
```

10.1.2. Перехоплення класів виняткових ситуацій

Як відзначено вище, виняткова ситуація — це об'єкт, що може мати будь-який тип, як убудований, так і користувальницький. Створення класів користувальницьких ситуацій дозволяє програмісту точніше визначати реакцію програми на небажану ситуацію. Розглянемо приклад, що ілюструє ця теза.

Клас виняткових ситуацій

```
#include <iostream>

using namespace std;

class Error
{
public:
   int m;
   Error(){ printf("Помилка\n");}
   Error(int x):m(x){}
   void Message(){ printf("Розподіл на нуль!");}
};

class Rational
{
   int n;
   int m;
public:
   Rational(int x, int y){ n = x;    if(y==0)throw Error(y); else n = y; }
    ~Rational(){printf("Dtor Rational");}
};

int main()
{
   try
   {
      Rational q(1,0);
   }
   catch(Error& zero)
   {
      zero.Message();
      return -1;
   }

   return 0;
}
```

Клас `Rational` реалізує концепцію раціонального числа. Зрозуміло, необхідно заборонити створення раціональних чисел, у яких знаменник дорівнює нулю. Для цього передбачений клас `Error`, що представляє

собою виняткову ситуацію. Його об'єкт створюється тільки в тому випадку, якщо поле `m` в об'єкті класу `Rational` дорівнює нулю.

В цьому прикладі почата спроба створити об'єкт виду 1/0. Конструктор класу `Rational` згенерував виняткову ситуацію класу `Error`, що була по посиланню передане в розділ `catch`. У цьому розділі відбувається звертання до функції `Message()` — члену класу `Error`. У підсумку, на екрані з'явиться таке повідомлення.

```
Розподіл на нуль!
```

Зверніть увагу на те, що виняткова ситуація передається по посиланню. Це зовсім не обов’’язково, але дуже бажано, оскільки передача по посиланню ефективніше, ніж передача за значенням.

10.1.3. Ієрархія виняткових ситуацій

Оскільки виняткова ситуація може бути об'єктом класу, у мові C++ існує можливість створювати ієрархію виняткових ситуацій. У цьому випадку блок `catch` перехоплює об'єкти не тільки базового, але і похідних класів. При генерації похідних виняткових ситуацій це приводить до непорозумінь — їх перехоплює блок `catch`, призначений для обробки базових виняткових ситуацій.

Створення ієрархії виняткових ситуацій

```
#include <iostream>

using namespace std;

class ErrorBase
{
public:
   long m;
   ErrorBase(){ printf("ErrorBase\n");}
   ErrorBase(long x):m(x){printf("ErrorBase\n");}
   void Message(){ printf("Розподіл на нуль!");}
};

class ErrorDerived:public ErrorBase
{
public:
   long m;
   ErrorDerived(){ printf("ErrorDerived\n");}
   ErrorDerived(long x):m(x){printf("ErrorDerived\n");}
   void Message(){ printf("Розподіл на нескінченність!");}
};

class Rational
{
   long n;
   long m;
public:
   Rational(long x, long y)
   {
       n = x;
       if(y==0)throw ErrorBase(y);
       else n = y;
       if(y>=1000000)throw ErrorDerived(y);
       else n = y;
   }
   ~Rational(){printf("Dtor Rational");}
};

int main()
{
   try
   {
      Rational q(1,1000000000);
   }
   catch(ErrorBase& zero)
```

```
    {
       zero.Message();
       return -1;
    }
    catch(ErrorDerived& infinity)
    {
       infinity.Message();
       return -1;
    }

    return 0;
 }
```

У цій програмі оголошена ієрархія виняткових ситуацій — базовий клас `ErrorBase` і похідний від нього клас `ErrorDerived`. Виняткові ситуації базового класу генеруються, якщо конструктор класу `Rational` намагається створити об'єкт із нульовим знаменником, а похідний клас `ErrorDerived` описує реакцію програми, коли знаменник занадто великий (більше мільйона). Програма виводить на екран наступні рядки.

```
ErrorBase
ErrorDerived
Розподіл на нуль!
```

Як бачимо, при спробі створити об'єкт, знаменник якого більше мільйона, була згенерована ситуація `ErrorDivide`, однак її перехопив блок `catch`, набудований на базовий клас `ErrorDivide`.

Для розв'язку цієї проблеми необхідно розмістити блок `catch`, що відповідає похідному класу, вище блоку, призначеного для перехоплення об'єктів класу `ErrorBase`.

Перехоплення виняткових ситуацій визначеного класу

```
#include <iostream>

 using namespace std;

 class ErrorBase
 {
 public:
    long m;
    ErrorBase(){ printf("ErrorBase\n");}
    ErrorBase(long x):m(x){printf("ErrorBase\n");}
    virtual void Message(){ printf("Розподіл на нуль!");}
 };

 class ErrorDerived:public ErrorBase
 {
 public:
    long m;
    ErrorDerived(){ printf("ErrorDerived\n");}
    ErrorDerived(long x):m(x){printf("ErrorDerived\n");}
    void Message(){ printf("Розподіл на нескінченність!");}
 };

 class Rational
 {
    long n;
    long m;
 public:
    Rational(long x, long y)
    {
        n = x;
        if(y==0)throw ErrorBase(y);
        else n = y;
        if(y>=1000000)throw ErrorDerived(y);
        else n = y;
    }
    ~Rational(){printf("Dtor Rational");}
 };
```

```
int main()
{
   try
   {
      Rational q(1,1000000000);
   }
   catch(ErrorDerived& infinity)
   {
      infinity.Message();
      return -1;
   }
   catch(ErrorBase& zero)
   {
      zero.Message();
      return -1;
   }
   return 0;
}
```

На екрані ми побачимо наступні рядки.

```
ErrorBase
ErrorDerived
Розподіл на нуль!
```

Зауважимо, що ця задача має ще одне розв'язок — можна оголосити функцію-член `Message()` віртуальної. У цьому випадку оператор `catch`, призначений для обробки виняткових ситуацій базового типу, буде як і раніше перехоплювати об'єкти похідного типу, вважаючи їх базовими. Однак механізм заміщення віртуальних функцій-членів базового класу дозволяє правильно обробити виняткову ситуацію. Утім, цей спосіб вимагає визначеної обережності — вся обробка повинна бути передбачена у функціях-членах похідного класу. Не слід забувати, що керування передається в блок `catch`, призначений для перехоплення *базових* виняткових ситуацій!

Перехоплення виняткових ситуацій похідного класу

```
#include <iostream>

using namespace std;

class ErrorBase
{
public:
   long m;
   ErrorBase(){ printf("ErrorBase\n");}
   ErrorBase(long x):m(x){printf("ErrorBase\n");}
   virtual void Message(){ printf("Розподіл на нуль!\n");}
};

class ErrorDerived:public ErrorBase
{
public:
   long m;
   ErrorDerived(){ printf("ErrorDerived\n");}
   ErrorDerived(long x):m(x){printf("ErrorDerived\n");}
   void Message(){ printf("Розподіл на нескінченність!\n");}
};

class Rational
{
   long n;
   long m;
public:
   Rational(long x, long y)
   {
       n = x;
       if(y==0)throw ErrorBase(y);
```

```
        else n = y;
        if(y>=1000000)throw ErrorDerived(y);
        else n = y;
    }
    ~Rational(){printf("Dtor Rational");}
};

int main()
{
    try
    {
       Rational q(1,1000000000);
    }
    catch(ErrorBase& zero)
    {
       zero.Message();
       printf("Усередині блоку catch для класу ErrorBase!\n");
       return -1;
    }
    catch(ErrorDerived& infinity)
    {
       infinity.Message();
              printf("Усередині блоку catch для класу ErrorDerived!\n");
       return -1;
    }
      return 0;
}
```

Тепер програма виводить на екран очікуваний результат, хоча керування передається блоку catch, призначеному для перехоплення базових ситуацій, оскільки він розташований вище блоку, що відповідає похідної виняткової ситуації.

```
ErrorBase
ErrorDerived
Розподіл на нуль!
Усередині блоку catch для класу ErrorBase!
```

Зверніть увагу на те, що перехоплення виняткових ситуацій, що виникли в конструкторі, припиняє створення об'єкта. З цієї причини деструктор наприкінці програми не викликається.

10.2. Тонкості обробки виняткових ситуацій

Розглянемо декілька корисних прийомів, що дозволяють ефективно використовувати механізм виняткових ситуацій.

10.2.1. Тотальне перехоплення виняткових ситуацій

Іноді ретельна деталізація виняткових ситуацій не потрібна. Наприклад, у попередніх прикладах їх обробка проводилася майже однаково, за винятком супутніх повідомлень про помилки. Отже, було б зручно, якби блок catch можна було настроїти на будь-яку виняткову ситуацію. Зробити це дозволяє наступна конструкція.

```
catch(...)
 {
   // Перехоплення усіх виняткових ситуацій
}
```

Трикрапка в дужках означає, що блок catch здатний перехопити й обробити будь-яку виняткову ситуацію. Повернемося до попереднього прикладу і замінимо блоки catch новою конструкцією.

Тотальне перехоплення виняткових ситуацій: перший варіант

```
#include <iostream>

using namespace std;

void Message();
```

```
 class ErrorBase
 {
 public:
    long m;
    ErrorBase(){ printf("ErrorBase\n");}
    ErrorBase(long x):m(x){printf("ErrorBase\n");}
    virtual void Message(){ printf("Розподіл на нуль!");}
 };

 class ErrorDerived:public ErrorBase
 {
 public:
    long m;
    ErrorDerived(){ printf("ErrorDerived\n");}
    ErrorDerived(long x):m(x){printf("ErrorDerived\n");}
    void Message(){ printf("Розподіл на нескінченність!!");}
};

class Rational
{
    long n;
    long m;
public:
    Rational(long x, long y)
    {
        n = x;
      if(y==0)throw ErrorBase(y);
        else n = y;
      if(y>=1000000)throw ErrorDerived(y);
        else n = y;
    }
    ~Rational(){printf("Dtor Rational");}
};


int main()
{
    int n = 5;
    for(int i = 1; i<=n; i++)
    {
       try
       {
         if(i%2) Rational q(1,0); else Rational q(1,100000000);
       }
       catch(...)
       {
         Message();
       }
    }
    return 0;
 }

    void Message(){ printf("Виняткова ситуація!\n");}
```

Ця програма генерує різні виняткові ситуації: при непарних індексах циклу `i` — базову виняткову ситуацію класу `ErrorBase`, а при парних — похідну виняткову ситуацію класу `ErrorDivide`. Усі вони перехоплюються тим самим блоком `catch`.

```
ErrorBase
Виняткова ситуація!
ErrorBase
ErrorDerived
Виняткова ситуація!
ErrorBase
Виняткова ситуація!
ErrorBase
ErrorDerived
Виняткова ситуація!
```

```
ErrorBase
Виняткова ситуація!
```

Крім того, якщо програміст сумнівається в тім, що він передбачив усі можливі виняткові ситуації, після всіх розділів `catch` можна поставити розділ `catch(...)`. У цьому випадку оператор `catch(...)` використовується для підстрахування, перехоплюючи виняткові ситуації, не передбачені програмістом. (Це нагадує застосування метки `default` в операторі `switch`.)

Для ілюстрації цього прийому повернемося до попередньої програми. Нагадаємо, що виняткові ситуації базового класу `ErrorBase` генеруються при спробі створити об'єкт, що імітує раціональне число з нульовим знаменником, а похідний клас `ErrorDerived` описує ситуацію, коли знаменник більше мільйона. Ясно, що ці не усі виняткові ситуації, що можуть виникнути. Наприклад, обмежений обсяг комп'ютерної пам'яті не дозволяє працювати з числами типу `long`, довжина яких перевищує максимальну. Звичайно, можна було б передбачити і це, написавши клас `ErrorLong`. (І це було б найкращим розв'язком!)

```
catch(const char* s)
 {
    printf("%s",s);
 }
```

І все, щоб перестрахуватися, можна поставити на останнє місце в ланцюжку блоків `catch` універсальний перехоплювач.

Тотальне перехоплення виняткових ситуацій: другий варіант

```
#include <iostream>
#include <limits.h>

 using namespace std;

 class ErrorBase
 {
 public:
    long m;
    ErrorBase(){ printf("ErrorBase\n");}
    ErrorBase(long x):m(x){printf("ErrorBase\n");}
    virtual void Message(){ printf("Розподіл на нуль!\n");}
 };

 class ErrorDerived:public ErrorBase
 {
 public:
    long m;
    ErrorDerived(){ printf("ErrorDerived\n");}
    ErrorDerived(long x):m(x){printf("ErrorDerived\n");}
    void Message(){ printf("Розподіл на нескінченність!\n");}
 };


 class Rational
 {
    long n;
    long m;
 public:
    Rational(long x, long y)
    {
        if(x >= LONG_MAX) throw "Занадто великий чисельник";
        if(y >= LONG_MAX) throw "Занадто великий знаменник";
        n = x;
        if(y==0)throw ErrorBase(y);
        else n = y;
        if(y>=1000000)throw ErrorDerived(y);
        else n = y;
    }
    ~Rational(){printf("Dtor Rational");}
 };


 int main()
```

```
{
   try
   {
      Rational q(1,LONG_MAX);
   }
   catch(ErrorBase& zero)
   {
       zero.Message();
       printf("Усередині блоку catch для класу ErrorBase!\n");
       return -1;
   }
   catch(ErrorDerived& infinity)
   {
       infinity.Message();
       printf("Усередині блоку catch для класу ErrorDerived!\n");
       return -1;
   }
   catch(...)
   {
      printf("Непередбачена виняткова ситуація!");
      return -1;
   }
   return 0;
}
```

Оскільки блок `catch(...)` не має аргументів, у ньому важко передбачити точну реакцію на виниклу виняткову ситуацію, наприклад вивести діагностичний рядок. Мабуть, єдине, що залишається — припинити роботу програми і видати на екран відповідне повідомлення .

10.2.2. Генерація виняткових ситуацій

Оголошуючи функцію, можна перелічити типи виняткових ситуацій, що їй дозволено генерувати. Типи виняткових ситуацій, не включені в список, функція генерувати не зможе. Спроба порушити це обмеження приведе до негайного припинення роботи програми за допомогою ланцюжка викликів стандартних функцій: `unexpected()` — `abort()`. Якщо список порожній, функція взагалі не повинна генерувати ніяких виняткових ситуацій.

Для оголошення списку дозволених виняткових ситуацій використовується наступна синтаксична конструкція.

```
тип_ щоповертається_значення ім'я_функції(аргументи) throw(виняткові ситуації)
 {
   // ...
}
```

Повернемося до демонстраційної програми, що використовує функцію `Check()`.

Список дозволених виняткових ситуацій: перший варіант

```
#include <iostream>
#define MAX 1000000
using namespace std;

int CheckZero(int) throw (const char*);
int CheckMax(int) throw (int);
int Divide(int, int);

int main()
{
   int n = MAX, m=0, l;

   printf("Початок\n");
   try
   {
     printf("Усередині блоку try\n");
     l = Divide(n+1,m);
   }
   catch (const char* s)
   {
```

```
      printf("%s\n",s);
    }
    catch (int k)
    {
      printf("Число чи більше дорівнює %d\n",k);
    }

    printf("Кінець\n");

    return 0;
 }

 int CheckZero(int i) throw (const char*)
 {
    printf("Усередині функції Check\n");
    if(i==0) throw("Розподіл на нуль усередині функції main!");
    printf("Кінець функції Check\n");
    return i;
 }

 int CheckMax(int i) throw (int)
 {
    printf("Усередині функції CheckMax\n");
    if(i>=MAX) throw MAX;
    printf("Кінець функції CheckMax\n");
    return i;
 }

 int Divide(int n, int m)
 {
    printf("Усередині функції Divide\n");
    n=CheckMax(n);
    m=CheckMax(m);
    m=CheckZero(m);
    printf("Кінець функції Divide\n");
    return n/m;
 }
```

У цій програмі, що виконує розподіл двох цілих чисел, можливі наступні виняткові ситуації: 1) якщо чисельник або знаменник перевищує максимально припустиме число і 2) знаменник дорівнює нулю. Для порівняння числа з максимальним використовується функція `CheckMax()`, а для порівняння з нулем — функція `CheckZero()`. Функції `CheckMax()` дозволено генерувати виняткову ситуацію типу `int`, а функція `CheckZero()` може генерувати виняткову ситуацію типу `const char*`. Якщо ці функції спробують згенерувати виняткову ситуацію, не зазначену в списку `throw`, виконання програми завершиться аварійно.

```
Початок
Усередині блоку try
Усередині функції Divide
Усередині функції CheckMax
Число більше 100000
Кінець
```

Припустимо тепер, що в програмі ані чисельник, ані знаменник не повинні дорівнювати нулю. Отже, функція, що виконує перевірку, повинна мати можливість генерувати обидві виняткові ситуації: число дорівнює нулю і число більше максимального. Модифікований варіант програми виглядає так.

Список дозволених виняткових ситуацій: другий варіант

```
#include <iostream>
 #define MAX 1000000
 using namespace std;

 int Check(int) throw (const char*, int);
 int Divide(int, int);

 int main()
 {
```

```
   int n = 0, m=0, l;

   printf("Початок\n");
   try
   {
      printf("Усередині блоку try\n");
      l = Divide(n,m);
   }
   catch (const char* s)
   {
      printf("%s\n",s);
   }
   catch (int k)
   {
   printf("Число чи більше дорівнює %d\n",k);
   }
   printf("Кінець\n");

   return 0;
}

int Check(int i) throw (const char*, int)
{
   printf("Усередині функції Check\n");
   if(i==0) throw("Число дорівнює нулю!");
   if(i>=MAX) throw MAX;
      printf("Кінець функції Check\n");
   return i;
}

int Divide(int n, int m)
{
   printf("Усередині функції Divide\n");
   n=Check(n);
   m=Check(m);
   printf("Кінець функції Divide\n");
   return n/m;
}
```

Результат роботи цієї програми такий.

```
Початок
Усередині блоку try
Усередині функції Divide
Усередині функції Check
Число дорівнює нулю
Кінець
```

Список дозволених виняткових ситуацій поширюється лише на типи значень, що повертаються функцією в модуль виклику. Інакше кажучи, ці обмеження не стосуються блоків `try`, що знаходяться усередині функції. Якщо ми перенесемо обробку виняткових ситуацій усередину функції `Check()`, їй можна взагалі заборонити генерувати виняткові ситуації і передавати їх по ланцюжку модулів виклику.

Заборона генерування виняткових ситуацій

```
#include <iostream>

#define MAX 100000

using namespace std;
int Check(int) throw();
int Divide(int, int);

int main()
{
   int n = 0, m=MAX, l;

   printf("Початок\n");
```

```
    l = Divide(n,m+1);
    printf("Кінець\n");

    return 0;
 }

 int Check(int i) throw ()
 {
    printf("Усередині функції Check\n");
    try
    {
      printf("Усередині блоку try\n");
      if(i==0) throw("Число дорівнює нулю!");
      if(i>=MAX) throw MAX;
    }
    catch (const char* s)
    {
    printf("%s\n",s);
    return -1;
    }
    catch (int k)
    {
    printf("Число чи більше дорівнює %d\n",k);
    return -1;
    }

    printf("Кінець функції Check\n");
    return i;
 }

 int Divide(int n, int m)
 {
    printf("Усередині функції Divide\n");
    n=Check(n);
    m=Check(m);
    if (n==-1 || m==-1) { printf("Аварійне завершення!"); return -1;}
    printf("Кінець функції Divide\n");
    return n/m;
 }
```

Результат роботи цієї програми виглядає в такий спосіб.

```
Початок
Усередині функції Divide
Усередині функції Check
Усередині блоку try
Число дорівнює нулю!
Усередині функції Check
Усередині блоку try
Число чи більше дорівнює 100000
Аварійне завершення
```

10.2.3. Повторні виняткові ситуації

Іноді зручно обробляти виняткову ситуацію в декількох розділах `catch`, ніби передаючи її по конвеєру. Уявимо собі, що на першому етапі оброблювач просто констатує наявність проблеми і пропонує користувачу вирішити, що робити — намагатися виправити чи помилку припинити роботу програми. Якщо програміст вирішить продовжувати роботу, необхідно згенерувати ситуацію знову й ужити заходів, що дозволяють врятувати положення. Якщо ж програміст вирішить припинити роботу програми, викликається відповідна чи функція виконується оператор `return`.

Передача виняткової ситуації по ланцюжку

```
 #include <iostream>
 #define MAX 100000

 using namespace std;
```

```
int Check(int) throw();
int Divide(int, int);

int main()
{
   int n = 0, m=MAX, l;

   printf("Початок\n");
   l = Divide(n,m+1);
   printf("Кінець\n");

   return 0;
}

int Check(int i) throw ()
{
   int choice;
   printf("Усередині функції Check\n");

   try
   {
     printf("in try Усередині блоку try\n");
     if(i==0) throw("Zero Число дорівнює нулю!");
     if(i>=MAX) throw MAX;
   }
   catch (const char* s)
   {
     printf("%s\n",s);
     printf("Ваш вибір? 0 - Вихід; 1 - Продовжити");
     scanf("%d",&choice);
     if(!choice) { printf("Виходимо\n"); return -1;}
        else throw;
    }
   catch (int k)
   {
     printf("Число чи більше дорівнює %d\n",k);
     return -1;
   }

   printf("Кінець функції Check\n");
   return i;
}

int Divide(int n, int m)
{
   printf("Усередині функції Divide\n");
   try
   {
         n=Check(n);
   }
   catch (const char* s)
   {
   printf("%s\n",s);
   printf("Продовжуємо\n");
   return -1;
   }

   try
   {
     m=Check(m);
   }
   catch (const char* s)
   {
     printf("%s\n",s);
     printf("Продовжуємо\n");
     return -1;
   }
```

```
    if (n==-1 || m==-1)
       {printf("Аварійне завершення!\n"); return -1;}
    printf("Кінець функції Divide\n");
    return n/m;
 }
```

Проаналізуємо хід виконання цієї програми.

```
Початок
Усередині функції Divide
Усередині функції Check
Усередині блоку try
Число дорівнює нулю!
Ваш вибір? 0 - Вихід; 1 - Продовжити
 1
Число дорівнює нулю!
Продовжуємо
Кінець
```

Чисельник дробу дорівнює нулю, а знаменник — максимальному значенню. Усередині функції `Divide()` викликається функція `Check()`, що повинна перевірити коректність чисельника. Знайшовши, що чисельник дорівнює нулю, функція `Check()` генерує виняткову ситуацію типу `const char*`. Її перехоплює блок `catch`, усередині якого користувачу пропонується або вийти з програми, увівши число 0, або продовжити виконання, увівши 1 чи будь-яке інше число. У даному випадку користувач ввів одиницю. У відповідь на це функція `Check()` повторно згенерувала виняткову ситуацію типу `const char*`. Для цього використовується оператор `throw` без параметрів, що генерує повторно поточну ситуацію. Оскільки тому самому блоку `try` не можна поставити у відповідність декілька блоків `catch` з однаковим набором аргументів, виняткова ситуація передається по ланцюжку в модуль виклику. Про це свідчить рядок `Продовжуємо`.

Аналогічна обробка виняткової ситуації передбачена і для знаменника. Якщо у відповідь на запит програми користувач уведе нуль, на екрані з'являться наступні рядки.

```
Виходимо
Усередині блоку Check
Усередині блоку try
Число чи більше дорівнює 100000
Аварійне завершення!

Кінець
```

10.2.4. Непередбачені виняткові ситуації

Для того щоб реагувати на виняткові ситуації, обробка яких програмістом не передбачена, у мові C++ використовуються функції `terminate()` і `unexpected()`, оголошені в заголовку `<exception>`, а також функція `abort()`, прототип якої міститься в заголовках `<process.h>` і `<stdlib.h>`.

Механізм роботи цих функцій виглядає в такий спосіб. Виняткова ситуація, що не перехопив жодний оператор `catch`, передається нагору по ланцюжку викликів. Якщо вона не обробляється в жодному з модулів виклику, викликається `terminate()`, що за замовчуванням викликає функцію `abort()`. Такі виняткові ситуації називаються *непередбаченими* (unexpected).

С допомогою стандартних засобів простежити, коли і які функції викликаються, неможливо. Звичайно вважається, що спочатку викликається функція `unexpected()`, що автоматично викликає функцію `terminate()`, а та у свою чергу — функцію `abort()` за замовчуванням. На щастя, у мові C++ існує механізм заміни функцій `terminate()` і `unexpected()` власними оброблювачами непередбачених ситуацій. Для цього призначені функції `set_terminate()` і `set_unexpected()`, специфікація яких міститься в заголовку `<exception>`.

```
new_unexpected set_unexpected(new_unexpected) throw();
new_terminate set_terminate(new_terminate) throw();
```

Ідентифікатори `new_unexpected` і `new_terminate` — імена нових оброблювачів. Аргументами функцій `set_unexpected()` і `set_terminate()` є вказівники на ці оброблювачі. Оскільки цими вказівниками автоматично є імена нових функцій, синтаксична конструкція виглядає досить просто. Одержуючи адресу нового оброблювача, функції `set_unexpected()` і `set_terminate()` повертають адреси старих оброблювачів. Це дозволяє зберегти їх у пам'яті і відновити при необхідності стару систему обробки непередбачених ситуацій. Заступники функцій `unexpected()` і `terminate()` не повинні повертати керування програми. Вони зобов'язані поводитися, як їхні прототипи.

Продемонструємо систему обробки непередбачених ситуацій наступними прикладами. Спочатку розглянемо програму, у якій не передбачена обробка виняткової ситуації, що виникає, коли число перевищує максимально припустиме.

Реакція на непередбачені виняткові ситуації

```
#include <iostream>

#define MAX 100000

using namespace std;
int Check(int) throw();
int Divide(int, int);

int main()
{
    int n = 0, m=MAX, l;

    printf("Start\n");
    l = Divide(n,m+1);
    printf("Finish\n");
    return 0;
}

int Check(int i) throw ()
{
    printf("In Check\n");

    try
    {
      printf("In try\n");
      if(i==0) throw("Zero!");
      if(i>=MAX) throw MAX;
    }
    catch(const char* s)
    {
        printf("%s\n",s);
    }
    printf("End Check\n");
    return i;
 }

 int Divide(int n, int m)
 {
    printf("In Divide Усередині функції Divide\n");
    n=Check(n);
    m=Check(m);
    if (n==-1 || m==-1) { printf("Abnormal termination!\n"); return -1;}
    printf("End Divide\n");
    return n/m;
 }
```

У цьому випадку на екрані з'являться такі рядки.

```
Початок
Усередині функції Divide
Усередині функції Check
Усередині блоку try
Число дорівнює нулю!
Кінець функції Check
Усередині функції Check
Усередині блоку try
```

На цьому виконання програми аварійно завершується. Справа в тім, що в ній не передбачена реакція на виняткову ситуацію типу `int`. Пошук оброблювачів у модулях виклику нічого не дав, і операційна система викликала функції `unexpected()` і `terminate()`. Цікаво, а в якому порядку? Щоб відповісти на це питання, замінимо стандартні функції `unexpected()` і `terminate()` своїми версіями. Зверніть увагу на те, що знайти

неопрацьовану виняткову ситуацію можна також за допомогою функції `uncaught_exception()`, що повертає значення типу `bool`.

Застосування функції uncaught_exception()

```
#include <exception>
 #include <iostream>
 #include <process.h>
 #define MAX 100000
 using namespace std;

 int Check(int) throw (const char*, int);
 int Divide(int, int);
 void new_unexpected();
 void new_terminate();

 int main()
 {
    int n = 0, m=MAX, l;

    typedef void (*pNew_unexpected)();
    typedef void (*pNew_terminate)();

    pNew_unexpected oldAddress1;
    pNew_terminate oldAddress2;

    set_unexpected(new_unexpected); // Установлюємо новий обработчик unexpected
    set_terminate(new_terminate);   // Установлюємо новий обработчик terminate

    printf("Початок\n");
    l = Divide(n,m+1);
    printf("Кінець\n");

    oldAddress1 = set_unexpected(oldAddress1);
    oldAddress2 = set_terminate(oldAddress2);

    return 0;
 }

 int Check(int i) throw (const char*, int)
 {
    printf("Усередині функції Check\n");

    try
    {
      printf("Усередині блоку try\n");
      if(i==0) throw("Число дорівнює нулю!");
      if(i>=MAX) throw MAX;
    }
    catch(const char* s)
    {
        printf("%s\n",s);
    }
       printf("Кінець функції Check\n");
    return i;
 }

 int Divide(int n, int m)
 {
    printf("Усередині функції Divide\n");
    n=Check(n);
    m=Check(m);
    if (n==-1 || m==-1) { printf("Аварійне завершення!\n"); return -1;}
    printf("Кінець функції Divide\n");
    return n/m;
 }
```

```
void new_unexpected()
{
   printf("Новий оброблювач unexpected\n");
   exit(-1);
}

void new_terminate()
{
    if(!uncaught_exception())
    printf("Виявлена непередбачена виняткова ситуація!\n");
      else printf("Ok!\n");
   printf("Новий оброблювач terminate\n");
   exit(-1);
}
```

Хід виконання програми ілюструється наступними рядками.

```
Початок
Усередині функції Divide
Усередині функції Check
Усередині блоку try
Число дорівнює нулю!
Кінець функції Check
Усередині функції Check
Усередині блоку try
Виявлена непередбачена виняткова ситуація!
Новий оброблювач terminate
```

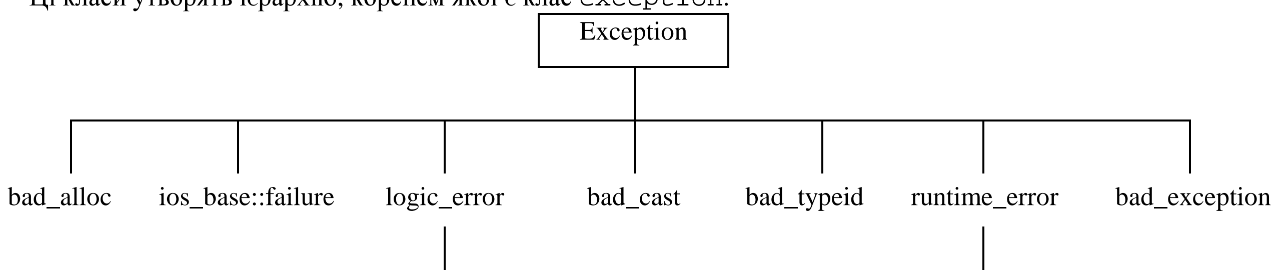
Як бачимо, при виявленні непередбаченої виняткової ситуації викликається оброблювач `new_terminate`, що завершує роботу програми.

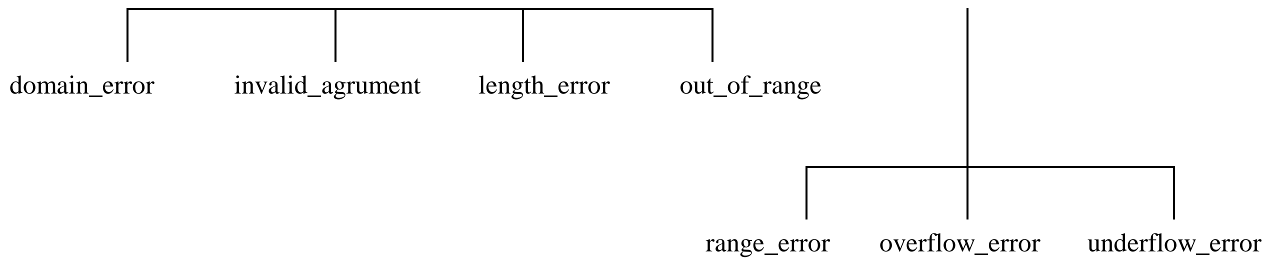
10.3.Стандартні виняткові ситуації

У мові C++ передбачено вісім стандартних виняткових ситуацій, що генеруються операторами і конструкторами стандартних класів.

Виняткова ситуація	Зміст	Заголовок
	Помилки при роботі з пам'яттю і типами	
`bad_alloc`	Помилка розподілу пам'яті	`<new>`
`bad_cast`	Некоректне приведення типів	`<typeinfo>`
`bad_typeid`	Неправильне застосування оператора `typeid`	`<typeinfo>`
`bad_exception`	Непередбачена виняткова ситуація	`<exception>`
	Логічні помилки	
`out_of_range`	Вихід за межі припустимого діапазону	`<stdexcept>`
`invalid_argument`	Невірний аргумент функції	`<stdexcept>`
`length_error`	Перевищення припустимих розмірів об'єкта	`<stdexcept>`
	Помилки при виконанні програми	
`domain_error`	Вихід за межі припустимого діапазону	`<stdexcept>`
`overflow_error`	Переповнення	`<stdexcept>`
`underflow_error`	Утрата значимості	`<stdexcept>`
`domain_error`	Вихід за межі припустимого діапазону	`<stdexcept>`
`ios_base::failure`	Помилка введення-висновку	`<stdexcept>`

Ці класи утворять ієрархію, коренем якої є клас `exception`.





Частина з них зв'язана з класами, оголошеними в стандартній бібліотеці шаблонів. Зупинимося поки на тих стандартних виняткових ситуаціях, що не належать до бібліотеки `STL`: `bad_alloc`, `bad_cast`, `bad_typeid` і `bad_exception`. Виняткові ситуації, що належать цим класам, поєднує одна загальна властивість — усі вони є об'єктами, а не вказівниками. Отже, вони передаються або за значенням, або по посиланню.

10.3.1. Клас bad_alloc

Мабуть, найбільш важлива виняткова ситуація описується класом `std::bad_alloc`.

Виняткова ситуація класу bad_alloc

```
#include <iostream>
#include <malloc.h>
#include <exception>

using namespace std;

int main()
{
   long* q = NULL;
   try { q = new long[5000000000000];}
   catch(const bad_alloc&){ cout << "Bad_alloc"; }
   catch(...){cout << "Інші";}
   cout << q << endl;
   delete[] q;

   return 0;
}
```

Ця програма не обов'язково викликає виняткову ситуацію. Однак якщо виникне недостача пам'яті, буде згенерирований об'єкт класу `bad_alloc`, а на екран буде виведена назва виняткової ситуації і нульове значення вказівника `q`.

10.3.2. Клас bad_cast

Виняткова ситуація `bad_cast` виникає, коли спроба приведення типу є некоректною. Наприклад, не можна приводити поліморфний тип до посилання на інший поліморфний тип.

Виняткова ситуація класу bad_cast

```
#include <iostream>
#include <typeinfo>

using namespace std;

class A
{
   int n;
public:
   A(int k):n(k){}
   virtual void view(){ cout << n << endl;}
};

class B: public A
{
   int m;
public:
```

```
   B(int l): A(l){m=l;}
   virtual void view(){ cout << m << endl;}
};

int main()
{
   A *p = new A(0);
   try
   {
      B& r = dynamic_cast<B&>(*p);  // Генерує виняткову ситуацію
   }
   catch (const bad_cast& ex)
   {
       cout << ex.what() << endl;
   }

   return 0;
}
```

При виконанні другого оператора в блоці `try` буде згенерована виняткова ситуація, повідомлення про яку виводить на екран функція `what()`, що належить класу `bad_alloc`. Зверніть увагу на те, що об'єкт класу `bad_cast` передається за допомогою константного посилання.

10.3.3. Клас bad_typeid

Виняткова ситуація, що належить класу `bad_typeid`, генерується, якщо оператор `dynamic_cast` застосовується до нульового вказівника. Повернемося до попереднього прикладу, злегка змінивши його.

Виняткова ситуація класу bad_typeid

```
#include <iostream>
#include <typeinfo>

using namespace std;

class A
{
   int n;
public:
   A(int k):n(k){}
   virtual void view(){ cout << n << endl;}
};

class B: public A
{
   int m;
public:
   B(int l):A(l){ m = l;}
   virtual void view(){ cout << m << endl;}
};

int main()
{
   A* p = new A(0);
   try
   {

       B* p = dynamic_cast<B*>(p);         // Некоректне приведення
       cout << typeid(*p).name() << endl;
   }
   catch (const bad_typeid& ex)
   {
       cout << ex.what() << endl;
   }
```

```
   return 0;
}
```

У цій програмі виконується спроба привести вказівник на базовий клас до вказівника на похідний клас. Однак у даному випадку це приведення неможливе, оскільки вказівник `p` посилається на об'єкт базового класу `A`. Для того щоб це приведення стало коректним, варто було б установити цей вказівник на об'єкт класу `B`. Таким чином, реагуючи на помилку, оператор `dynamic_cast` повертає нульовий вказівник. Спроба розіменувати його, виконуючи оператор `typeid`, породжує виняткову ситуацію `bad_typeid`.

10.3.4. Клас bad_exception

Виняткова ситуація `bad_exception` генерується функцією `unexpected()` у тих випадках, для яких не була передбачена обробка. Повернемося приміром, що ілюструє застосування списку дозволених виняткових ситуацій. Припустимо, у нас немає бажання перелічувати усі виняткові ситуації, що можуть виникнути при виконанні програми, оскільки програма повинна однаково реагувати на них (наприклад, виводити повідомлення на екран).

Виняткова ситуація класу bad_exception

```
#include <iostream>
#define MAX 1000000
using namespace std;

int Check(int) throw (const char*, bad_exception);
int Divide(int, int);

int main()
{
   int n = MAX, m=0, l;

   printf("Початок\n");
   try
   {
     printf("Усередині блоку try\n");
     l = Divide(n+1,m);
   }
   catch (const char* s)
   {
     printf("%s\n",s);
   }
   catch (int k)
   {
     printf("Число чи більше дорівнює %d\n",k);
   }

   printf("Кінець\n");

   return 0;
}

int Check(int i) throw (const char*, bad_exception)
{
   printf("Усередині функції Check\n");
   if(i==0) throw("Розподіл на нуль усередині функції main!");
   if(abs(i)>=MAX)  throw "Bad_exception...";
   printf("Кінець функції Check\n");
   return i;
}


int Divide(int n, int m)
{
   printf("Усередині функції Divide\n");
   n=Check(n);
   m=Check(m);
   printf("Кінець функції Divide\n");
```

```
    return n/m;
}
```

Нагадаємо, що ця програма виконує розподіл двох цілих чисел. Тепер для порівняння знаменника з нулем і максимально припустимим значенням використовується одна функція — `Check()`. Вона реагує на ситуацію, коли знаменник дорівнює нулю. В усіх інших випадках функція `unexpected()` виявляє непередбачену виняткову ситуацію і генерує об'єкт класу `bad_exception`.

```
Початок
Усередині блоку try
Усередині функції Divide
Усередині функції Check
Bad_exception...
Кінець
```

У завершення відзначимо, що обробка виняткових ситуацій приводить до додаткових витрат ресурсів комп'ютера. Програми, що передбачають обробку виняткових ситуацій, містять додатковий код і працюють повільніше. Таким чином, до цього механізму мови C++ варто прибігати тільки в дійсно *виняткових випадках*, коли без нього не можна обійтися. Наприклад, у попередній програмі ігнорування виняткових ситуацій зменшує час її виконання в два рази.

10.4. Резюме

- *Винятковою ситуацією* називається будь-яка подія, що вимагає особливої обробки
- Перевірка умов, що описують виняткову ситуацію, і реакція на її виникнення називається *обробкою виняткової ситуації*. Ця задача покладається на *оброблювача виняткової ситуації.*
- Обробка виняткових ситуацій у мові C++ є об'єктно-орієнтованою. Це значить, що виняткова ситуація є *об'єктом*, що генерується при виникненні незвичайних умов, передбачених програмістом, і передається *оброблювачу*, що неї *перехоплює*. Об'єктом, що описує природу виняткової ситуації, може бути будь-як сутність — літерал, рядок, об'єкт класу, число і т.д. Не слід думати, що виняткова ситуація обов'язково повинна бути об'єктом якого-небудь класу.
- В основі обробки виняткових ситуацій три ключових слова: `try`, `catch` і `throw`. Оператор `try` визначає область програми, де може виникнути виняткова ситуація, оператор throw генерує об'єкт — виняткову ситуацію, а оператор `catch` перехоплює цей об'єкт.
- Розмір блоку `try` не обмежений. У нього можна занурити як один оператор, так і цілу програму.
- Кожен блок `catch` відповідає окремому типу виняткової ситуації. Програма сама визначає, який з них виконати. У цьому випадку інші блоки `catch` не виконуються. Кожен блок `catch` має аргумент, що приймає визначене значення. Цей аргумент може бути об'єктом будь-якого типу. Якщо програма виконана правильно й у блоці `try` не виникло жодної виняткової ситуації, усі блоки `catch` будуть зігноровані.
- Якщо в програмі виникла подія, що програміст вважає небажаним, оператор `throw` генерує виняткову ситуацію. Для цього оператор `throw` повинний знаходитися усередині блоку `try` або усередині функції, викликуваної усередині блоку `try`.
- Якщо в програмі виникла виняткова ситуація, для якої не передбачені перехоплення й обробка, викликається стандартна функція `terminate()`, що, у свою чергу, викликає функцію `abort()`
- Якщо блок `try` знаходиться усередині функції, обробка виняткової ситуації виконується при кожнім виклику.
- Блоки `try` і `catch` нерозривні. Не можна помістити блок `try` у функцію, залишивши блок `catch` у функції `main()`. Необхідно або обробити виняткову ситуацію усередині функції, як це зроблено в більш ранньому прикладі, або перенести обробку в модуль виклику. У першому випадку функція, завершивши обробку, повертає визначене її специфікацією значення, а в другому — *виняткову ситуацію*.
- Оскільки виняткова ситуація може бути об'єктом класу, у мові C++ існує можливість створювати ієрархію виняткових ситуацій. У цьому випадку блок `catch` перехоплює об'єкти не тільки базового, але і похідних класів. При генерації похідних виняткових ситуацій це приводить до непорозумінь — їх перехоплює блок `catch`, призначений для обробки базових виняткових ситуацій.

- Для того щоб реагувати на виняткові ситуації, обробка яких програмістом не передбачена, у мові С++ використовуються функції `terminate()` і `unexpected()`, оголошені в заголовку `<exception>`, а також функція `abort()`, прототип якої міститься в заголовках `<process.h>` і `<stdlib.h>`.

- Ідентифікатори `new_unexpected` і `new_terminate` — імена нових оброблювачів. Аргументами функцій `set_unexpected()` і `set_terminate()` є вказівники на ці оброблювачі. Оскільки цими вказівниками автоматично є імена нових функцій, синтаксична конструкція виглядає досить просто. Одержуючи адресу нового оброблювача, функції `set_unexpected()` і `set_terminate()` повертають адреси старих оброблювачів. Це дозволяє зберегти їх у пам'яті і відновити при необхідності стару систему обробки непередбачених ситуацій. Заступники функцій `unexpected()` і `terminate()` не повинні повертати керування програми. Вони зобов'язані поводитися, як їхні прототипи.

10.5. Контрольні питання

1. Що називається винятковою ситуацією?
2. Що називається обробкою виняткової ситуації?
3. Як здійснюється обробка виняткової ситуації?
4. Опишіть зміст ключових слів `try`, `catch` і `throw`.
5. Опишіть особливості блоку `try`.
6. Опишіть особливості блоку `catch`.
7. Опишіть особливості вживання ключового слова `throw`.
8. Опишіть ієрархію виняткових ситуацій.
9. Як обробляються непередбачувані ситуації?
10. Як задати нові обробники непередбачуваних ситуацій?
11. Назвіть і опишіть стандартні виняткові ситуації.