Лекція 12

Тема лекції: Необроблені виключні ситуації. Заміна функцій unexpected( ) та terminate( ). Виключні ситуації та локальні об'єкти. Виключні ситуації та конструктори. Виключні ситуації та розподіл пам’яті

Необроблені виключні ситуації

Необроблені виключні ситуації передаються нагору по ланцюжку викликів одних функцій іншими доти, доки не зустрінеться відповідний їм оператор *catch* або доки більше не залишиться неоглянутих обробників виключних ситуацій. Якщо відбудеться останнє, для обробки виключної ситуації викликається одна з трьох спеціальних функцій, які автоматично приєднуються до кожної програми на C++, що використовує виключні ситуації. Ці функції мають імена *unexpected(), terminate(), abort().*

Вони викликаються відповідно до правил:

Якщо в програмі виникли виключні ситуації, що не обробляються оператором *catch,* то викликається функція unexpected*()*. Виключні ситуації, які не обробляються жодним оператором *catch,* називаються непередбаченими виключними ситуаціями. За замовчуванням *unexpected()* викликає функцію *terminate().* Непередбачені виключні ситуації можуть збуджуватися програмою при виявленні ушкодження стеку дескриптором класу, внаслідок чого викликається функція *terminate()*. За замовчуванням функція *terminate()* викликає функцію *abort()*.

Функція *abort()* негайно перериває виконання програми. Вона ніколи не викликається безпосередньо. Якщо не оброблені всі можливі виключні ситуаціїі та не вжиті заходи для перепрограмування функцій *unexpected()* і *terminate(),* необроблена виключна ситуація аварійно перерве виконання програми шляхом звертання до *abort().*

Можна замінити функції *unexpected()* і *terminate()* своїм кодом для обробки необроблених виключних ситуацій. Наприклад, має сенс замінити

функцію *unexpected()* для повідомлення користувача про будь-які необроблені виключні ситуації на етапі розробки програми. Це може допомогти виявити відсутні обробники помилок у програмі. В інших випадках, можна замінити *terminate()* діагностуючим кодом для перегляду пам'яті, щоб виявити оператори, які руйнують купу.

Замінити функцію *abort()* не можна. Її виклик завжди призводить до завершення програми.

Для завдання адреси власної функції-обробника непередбачених виключних ситуацій треба використовувати функцію *set_unexpected()*, оголошену в заголовочному файлі EXCEPT.H. Функція повинна мати тип *unexpected_function*, не мати аргументів і нічого не повертати.

Обробник користувача непередбачених виключних ситуацій може збуджувати виключну ситуацію. У цьому випадку пошуки оператора *catch* починаються з того місця, звідки спочатку було викликано обробник.

Для завдання власної функції-обробника завершення програми треба використовувати функцію *set_terminate()*, також оголошену в заголовочному файлі EXCEPT.H. Функція повинна мати тип *terminate_function*, не мати параметрів, нічого не повертати. Обидві функції *set_unexpected()* і *set_terminate()* повертають адресу поточної функції-обробника. Можна зберегти і потім відновити існуючі обробники, запам'ятавши їхні адреси в змінних, а потім передаючи їх назад функціям. Наприклад, спочатку оголошується прототип функції-обробника:

```
void unexpectedHandler();
```

Потім встановлюється обробник:

```
set_unexpected(unexpectedHandler);
```

Зберігання і відновлення адреси старої функції-обробника:

```
unexpected_function savedAddress();
savedAddress = set_unexpecred(unexpectedHandler);
//новий обробник
set_unexpected(saveAddress); //відновлення
//новий обробник більше не використовується
```

Функція користувача *terminate()* встановлюється аналогічно, тільки

замість *set_unexpected()* викликається функція *set_terminate()*, що повертає адресу типу *terminate_function*.

Приклад програми встановлення обробників непередбачених виключних ситуацій. В програмі також демонструється прийнятний засіб обробки непередбачених виключних ситуацій невідомих типів, що можуть збуджуватися в погано документованих бібліотечних функціях.

```
#include<iostream.h>
#include<except.h>
#define MAXERR 10
class MaxError{};
class Error
{
 public:
  Error();
  void Say();
 private:
  static int count;
};
void Run() throw(Error);
void trapper();
void zapper();
int Error::count;
void main()
 {
  set_unexpected(trapper);
  set_terminate(zapper);
   for(;;)
    {
     try{Run();
    }
   catch(Error e)
    {
     e.Say();
    }
}
  void Run() throw(Error)
```

```
 {
  throw Error();
  //throw "Невідомий тип об'єкту";
 }
void trapper()
 {
  cout << "Обробник непередбачених ситуаций.  .";
  throw Error();
 }
void zapper()
 {
  cout << "Обробник завершення функції";
  exit(-1);
 }
Error::Error()
 {
  count++;
  if(count>MAXERR)
   throw MaxError();
 }
void Error::Say()
 {
  cout << count << '\n';
 }
```

При запуску *unexpected.cpp* виведе 10 повідомлень про помилки перед завершенням. У програмі використовуються два класи виключних ситуацій. Об'єкт класу *MaxError*, що не має даних і функцій, посилається, коли число помилок перевищує константу *MAXERR*, задану рівною 10.

Конструктор класу *Error* інкрементує статичний член класу *count*. Член *count* - статичний, тому існує тільки один екземпляр цього значення і він існує доти, доки програма не завершиться. Член *count* - закритий член класу, отже він не зміниться в операторах програми поза класом. Функція-член *Say()* відображає значення лічильника помилок програми.

Якщо член *count* >= *MAXERR*, конструктор класу *Error* збуджує виключну ситуацію типу *MaxError*. При збудженні виключної ситуації об'єкт

класу не створюється, тому створення об’єкту класу *Error* переривається.

У функції *main()* установлюються функції-обробники, що заміщують за замовчуванням функції *unexpected() і terminate()*. Потім виконується нескінчений цикл *for*. Всередині циклу в блоці *try* викликається функція *Run()* і оператор *catch* перехоплює усі об'єкти класу *Error*, що посилає *Run()*. При збудженні виключної ситуації *catch* відображає поточне значення лічильника помилок шляхом звертання до функції *Say()* посланого об’єкту.

Функція *Run()* завжди збуджує виключні ситуації, моделюючи виникнення декількох помилок. Користувацький обробник непередбачених виключних ситуацій *trapper()* виводить повідомлення про виклик цієї функції. Це відбувається після збудження 10 виключних ситуацій. Обробник непередбачених виключних ситуацій може збудити ще одну виключну ситуацію для продовження програми. В прикладі оператор

```
throw Error();
```

посилає новий екземпляр класу *Error*. Створення об’єкту класу *Error* змушує конструктор цього класу збудити ще одну ситуацію типу *MaxError*. Цей тип помилки не підтримується в операторі *catch*, тому викликається обробник завершення програми *zapper()*. Функція завершення не може збудити виключні ситуації, а також не може виконати оператор *return*.

Перепрограмуємо функцію *Run()* так, щоб у ній збуджувалася виключні ситуації невідомого типу. Наприклад, у якості невідомого типу може виступити рядок.

Коли програма скомпілює і запустить обробник непередбачених ситуацій *trapper(),* то він буде викликатись для кожної виключної ситуації, що збуджується функцією *Run().* Це відбувається тому, що не існує оператора *catch* для об'єктів виключної ситуації типу *char** або *const char**. Функція *trapper(),* проте, транслює нерозпізнані об'єкти виключної ситуації, посилаючи об'єкт відомого типу, у даному випадку, типу *Error*. Нова збуджена виключна ситуація продовжує виконання програми в операторі *catch* всередині функції *main()*, що обробляє трансльовану виключну ситуацію.

Зрештою, у програмі максимальне число помилок буде перевищено, і

об'єкт класу *MaxError* буде посланий конструктором класу *Error*. В результаті викличеться обробник завершення програми шляхом звертання до бібліотечної функції *exit()*.

Лістинг результату в першому випадку (у *Run()* використовується *throw Error();*) :

```
1
2
...
10
Обробник непередбаченої помилки
Обробник завершення функції
У другому випадку (у Run() - рядок) :
Обробник невідомої помилки
1
Обробник невідомої помилки
2
...
10
Обробник невідомої помилки
Обробник завершення функції
```

Виключні ситуації і локальні функції

За допомогою спеціальної опції компілятора локальні об'єкти, що залишилися у стеку, автоматично знищуються збудженням виключної ситуації, тобто викликаються деструктори цих об'єктів. Для дозволу автоматичного знищення потрібно задати опцію *-xd* для автономного компілятора. Або з інтегрованого середовища вибрати *Options/Project*, відчинити пункт *C++ Options* і вибрати пункт *Exception handling / RTTI.*

RTTI - Runtime Type Information - інформація про час виконання. Повинні бути також встановлені пункти *Enable exception, Enable destructor cleanup*. Для Object-Windows програм обов'язково потрібно встановлювати автоматичне очищення.

Автоматичне знищення необхідно для функцій, що створюють локальні

об'єкти та збуджують виключні ситуації після їхнього створення.

Приклад: ця функція може залишити об'єкт класу *TAnyClass* у стеку:

```
int AnyFunction()
 {
  AnyClass object(123);
  if(condition) throw Error();
  return object.Value();
 }
```

Функція створює в класі об'єкт класу *TAnyClass*. Якщо події розвиваються нормально, при завершенні функції викликається деструктор об'єкту. Але якщо функція завершується виключною ситуацією, деструктор об’єкту не викликається, що може призвести до серйозних проблем. Використання ключа *-xd* або аналогічних установок в інтегрованому середовищі гарантує, що деструктор для всіх об'єктів буде викликаний по завершенню функції в результаті збудження виключної ситуації. Покажчики на об'єкти не видаляться автоматично.

```
int AnyFunction()
 {
  AnyClass *p = new AnyClass(123);
  if(condition)
   {
    delete p;
    throw Error();
   }
  delete p;
  return 123;
 }
```

Уникнути дублювання операторів *delete* можна тільки тоді, коли виключна ситуація збуджується після знищення динамічного об’єкту:

```
int AnyFunction()
 {
  AnyClass *p = new AnyClass(123);
  delete p;
   if(condition) throw Error();
    return 123;
```

```
}
```

Виключні ситуації і конструктори

Конструктори класу можуть збуджувати виключні ситуації для сигналізації про те, що вони не можуть створити об'єкт. Приклад:

```
class AnyClass
 {
  public:
   AnyClass()
    {
     if(condition)throw Error();
    }
   ~AnyClass(){ }
 };
```

Якщо конструтор збуджує виключну ситуацію, що означає ненормальне завершення конструктора, то деструктор цього об’єкту не викликається.

Розглянемо випадок:

```
class AnyClass
 {
  OtherClass x;
  public:
   AnyClass():x(123);
    {
     if(condition) throw Error();
    }
   ~AnyClass(){}
 };
```

Об'єкт *x* типу *OtherClass* створюється конструктором *AnyClass()* до того, як будуть виконуватись оператори в тілі конструктора. Якщо конструктор *OtherClass()* збуджує виключну ситуацію, інші оператори конструктора не виконуються. Оскільки код конструктора не виконався, об'єкт класу *AnyClass* не створюється і, отже, деструктор *~AnyClass()* не викликається.

Клас може збудити виключні ситуації свого ж типу. Зазвичай це робиться

за допомогою функції-члена, яку часто називають *Raise()*. Наприклад:

```
class Error
 {
  public:
   void Raise()
    {
     throw Error();
    }
 };
```

Якщо *e* - об'єкт класу *Error*, то оператор

```
e.Raise();
```

збуджує виключну ситуацію, посилаючи ще один об'єкт цього ж класу. В залежності від сервісу, що надається класом *Error*, створення нового об’єкту виключної ситуації може краще задовольнити запити користувача, ніж повторне збудження виключної ситуаціїі з вже існуючим об'єктом:

```
throw e;
```

Виключні ситуації та керування розподілом пам'яті

Оператор *new* збуджує виключну ситуацію, якщо не може задовольнити запит виділення пам'яті. Тип виключної ситуації - *xalloc*, визначений у заголовочному файлі EXCEPT.H таким чином:

```
class xalloc:public xmsg
 {
  public:
   xalloc(const string &msg, size_t size);
   size_t requested() const;
   void raise() throw(xalloc);
  private:
   size_t siz;
 };
```

Для використання виключної ситуації разом з *new* потрібно укласти оператори, що виділяють пам'ять, до блоку *try*, за яким повинен випливати оператор *catch* для об’єкту класу *xalloc*. Наприклад:

```
try
 {
  *p = new AnyClass();
 }
 catch(xalloc x)
 {
  cout << "Memory error:" << x.why();
  exit(-1);
 }
```

Клас *xalloc* виводиться з класу *xmsg*, оголошеного в тому ж заголовочному файлі. Клас-предок містить текстове повідомлення типу *string*. Для доступу до цього рядку використовується функція-член *why()*. Для визначення розміру невдалого запиту виділення пам'яті в байтах треба використовувати функцію *x.requested()*.

Можна використовувати один блок *try* у функції *main()* для обробки всіх виникаючих помилок.

```
void main()
 {
  for(;;)
   {
    try
    {
     Run();
    }
    catch(Xalloc x)
     {
      cout << "Out of memory";
      exit(-1);
     }
    }
 }
```

Тут ідея полягає у виклику єдиної функції *Run()*, що запускає програму. Будь-які виключні ситуації будуть оброблятись у функції *Run()*, або у функції *main()*.