Глава 9. **Исключения**

МГТУ им. Н.Э. Баумана
Факультет Информатика и системы управления
Кафедра Компьютерные системы и сети
Лектор: д.т.н., проф.
Иванова Галина Сергеевна

9.1 Обработка программных сбоев

Примеры причин сбоев при выполнении отлаженной программы:

- ввод неверных данных для расчета: деление на ноль, переполнение разрядной сетки, ограничение используемых методов и т.п.;
- невозможность выполнения операций ввода-вывода: отсутствие файла с заданным именем, сбой на диске или в сети, недоступность устройства;
- невозможность предоставления памяти программе по ее запросу;
- аварийное отключение электроэнергии;
- невозможность считать быстроменяющиеся данные из сети и т.п.

Виды исключительных ситуаций

- **Синхронные исключения** – могут возникнуть только в определённых, заранее известных точках программы. Так, ошибка чтения файла или коммуникационного канала, нехватка памяти — типичные синхронные исключения, так как возникают они только в операции чтения из файла или из канала или в операции выделения памяти соответственно.
- **Асинхронные исключения** могут возникать в любой момент времени и не зависят от того, какую конкретно инструкцию программы выполняет система. Типичные примеры таких исключений: аварийный отказ питания или поступление новых данных.

Синхронные исключения можно предусмотреть, асинхронные – предусмотреть невозможно!

Действия системы при обнаружении ошибки при выполнении программы

- автоматическое – выдать системное сообщение об ошибке и аварийно завершить программу без сохранения данных;
- программируемое:
  - если ошибка предусмотрена, то можно :
    - по возможности сохранить данные, выдать уточненное сообщение об ошибке и прервать выполнение программы – обработка без возврата;
    - "исправить" ошибку и продолжить работу – обработка с возвратом;
  - если используется механизм исключений, то можно перехватить уже возникшую аварийную ситуацию, предоставить возможность пользователю изменить данные и продолжить выполнение программы с исправленными данными – обработка с возвратом.

Автоматического завершения программы лучше не допускать!

Механизм исключений

Механизм исключений предполагает автоматическое формирование специального блока информации при обнаружении аварийной ситуации.

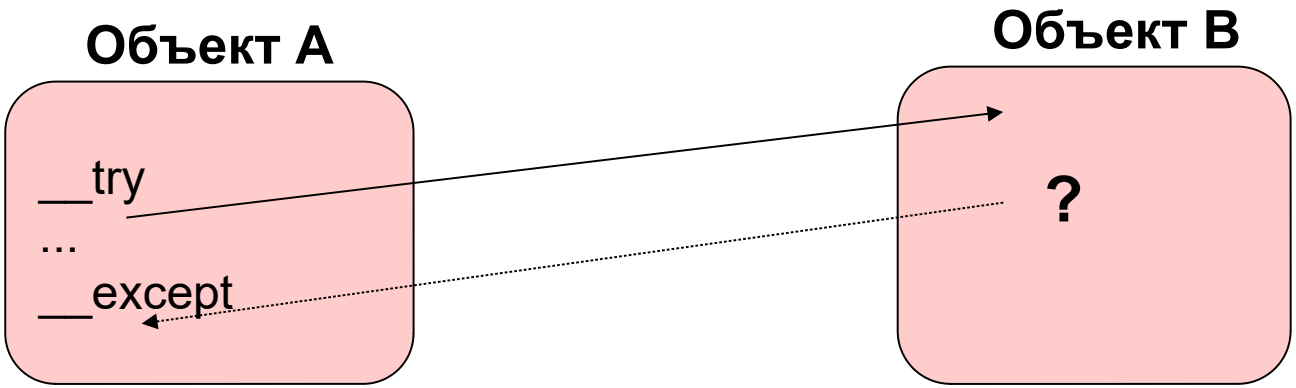
Традиционный подход:

```
if (p!=NULL) *p=3;
else <Действие по устранению ошибки>
```

Механизм исключений:

```
try {*p=3;}
catch (<Ссылка на тип>)
{ Действия по устранению ошибки}
```

Достоинство: позволяет группировать обработку ошибок в том месте, где возможно принятие решения об ее устранении:



9.2 Механизм исключений C++

В языках C и C++ используются разные стандарты обработки исключений.

Генерация исключений:

```
throw [<Тип>](<Аргументы>);
```

где <Тип> – тип или класс генерируемого блока; если тип не указан, то компилятор определяет его исходя из типа аргумента (обычно это один из встроенных типов);

<Аргументы> – одно или несколько выражений, значения которых будут использованы для инициализации генерируемого объекта.

Примеры:

```
а) throw ("Неверный параметр");  /* тип  const char * с указанным
                                    в кавычках значением  */
б) throw (221);  // тип const int с указанным значением

в) class E {   //класс исключения
     public: int num;            // номер исключения
             E(int n): num(n){}   // конструктор класса
   };
…
throw E(5);  //  тип исключения – класс E, значение – 5
```

Перехват и обработка исключений

try {<Защищенный код>}

catch (<Ссылка на тип>){<Обработка исключений>}

При перехвате исключений выполняются следующие правила:

1) исключение типа T будет перехватываться обработчиками исключений типов **T**, **const T**, **T&** или **const T&**;

2) обработчики типа базового класса перехватывают исключения типа производных классов;

3) обработчики типа **void*** перехватывают все исключения типа указателя.

Блок **catch**, для которого в качестве типа указано «…» обрабатывает исключения всех типов.

Пример:

```
try {<Операторы>} // выполняемый фрагмент программы
catch (EConvert& A){<Операторы>} /* перехват исключений
                                    типа EConvert */
catch (char* Mes){<Операторы>} //перехват исключений char*
catch(…) {<Операторы>}        //перехват остальных исключений
```

Пример генерации и перехвата исключений (Ex9_1)

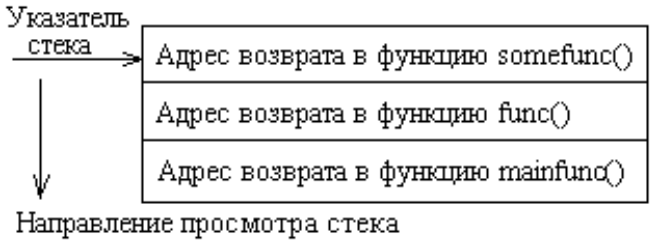
```
#include <iostream>
using namespace std;
void f(int a,int b,int &c){
   if (b!=0) c=a/b; else throw(5);
   if (c>50)throw('c');
}
int main()
{
   int a,b,c;
   e1: cout<<"Enter a,b";
   cin >> a >> b;
   try  { f(a,b,c); }
   catch(int){ cerr<<"Error b=0"<<endl;   goto e1; }
   catch(char){ cerr<<"Error c>50"<<endl; exit(-1); }
   return 0;
}
```

Возобновление исключения

Если перехваченное исключение не может быть обработано, то оно возобновляется.

Пример:

```
class E{};                    // класс исключения
void somefunc()
     { if(<условие> throw E();   }// генерация исключения
void func()
   {   try   { somefunc();   }
       catch(E& e){ if (<условие>) throw; } /* если здесь
                исключение обработать нельзя, то возобновляем его  */ }
void  mainfunc()
   {   try   { func();}
       catch(E& e){ … }     } // здесь обрабатываем исключение
```



Доступ к аргументам исключения

Использование имени объекта в качестве параметра оператора catch позволяет операторам обработки получить доступ к аргументам исключения через указанное имя.

Пример:

```
class E   //класс исключения
{  public: int num;              // номер исключения
          E(int n): num(n){}    // конструктор
};
   …
try { ...
throw E(5);          // генерируемое исключение
   ...}
catch (E& e){if (e.num==5) {…}} // получен доступ к полю
```

Последовательность обработки исключения

- при генерации исключения происходит конструирование временного объекта исключения;
- выполняется поиск обработчика исключения;
- при нахождении обработчика создается копия объекта с указанным именем;
- уничтожается временный объект исключения;
- выполняется обработка исключения;
- уничтожается копия исключения с указанным именем.

Поскольку обработчику передается копия объекта исключения, желательно в классе исключения с динамическими полями предусмотреть копирующий конструктор и деструктор.

Обработка объекта исключения (Ex9_2)

```
#include <iostream>
using namespace std;
class MyException
{    protected: int nError;
     public:  MyException(int nErr) {
     cout << "contructor"<<endl; nError = nErr;
     }
     MyException(MyException const &Obj) {
     cout<<"Copy contructor"<<endl; nError = Obj.nError;
     }
     ~MyException() { cout<<"destructor"<<endl; }
     void ErrorPut() {cout<<"Error "<<nError<<endl; }
};
int main()
{    try
     {  throw MyException(5); }
     catch (MyException E) { E.ErrorPut(); }
     cout<<"program"<<endl;
     return 0;
}
```

```
contructor
Copy constructor
Error 5.
destructor
destructor
program
```

Спецификация исключений

При объявлении функции можно указать, какие исключения она может генерировать:

throw(<тип>,<тип>…).

Пример:

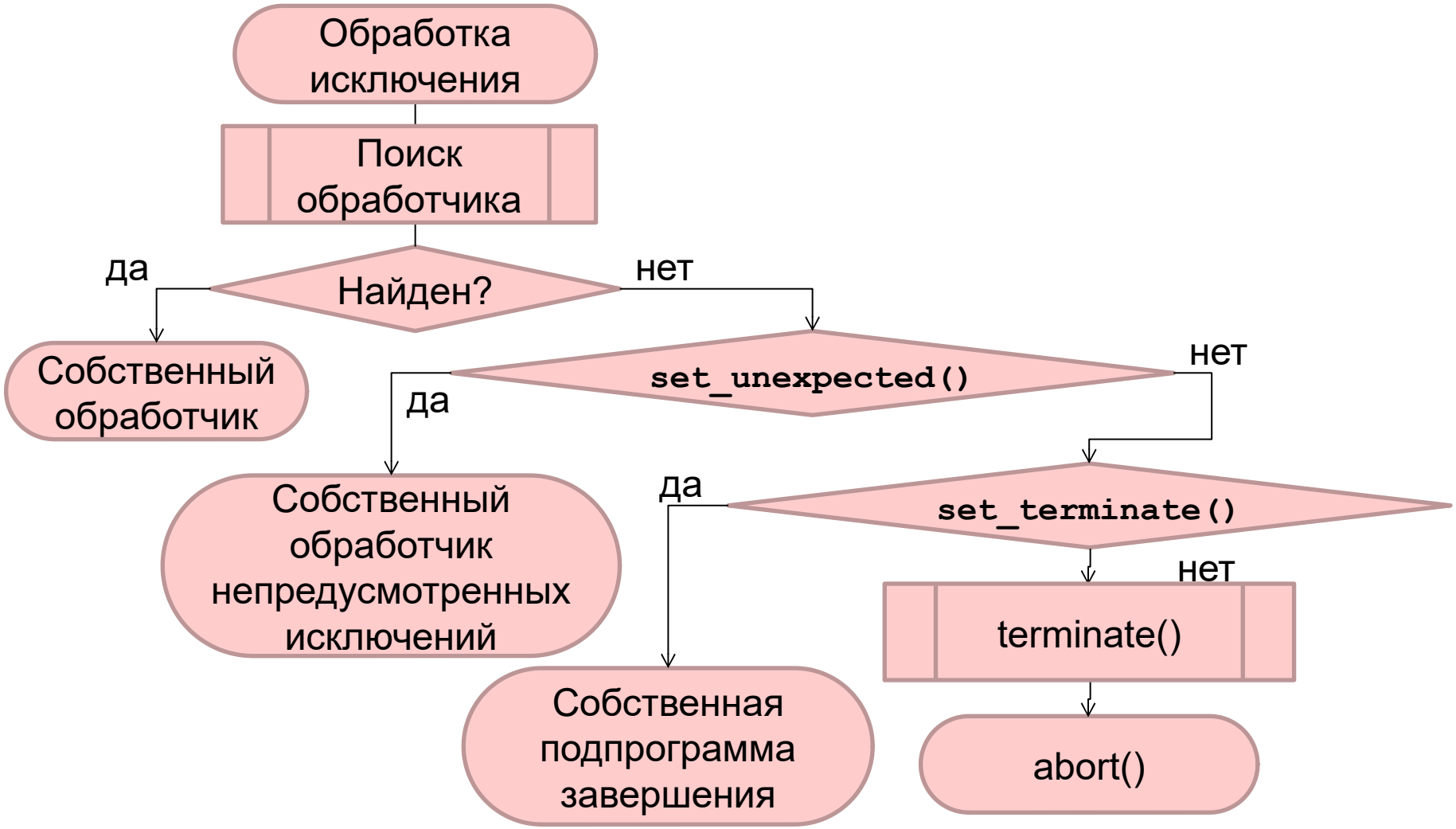
```
void func() throw(char*,int){…} /*функция может генерировать
                                исключения char* и int   */
```

Спецификация исключений не считается частью типа функции и, следовательно, ее можно изменить при переопределении:

```
class ALPHA
{ public:  struct ALPHA_ERR{};
  virtual void vfunc() throw(ALPHA_ERR){}

};
class BETA : public ALPHA
{ public:
        void vfunc() throw(char *) {}
};
```

Обработка исключений



Обработка непредусмотренных исключений

Для определения функции обработки непредусмороенных исключений в программе используется функция **set_unexpected**:

```
void my_unexpected()   {<обработка исключений> }
…
set_unexpected(my_unexpected);
```

Функция set_unexpected() возвращает старый адрес функции – обработчика непредусмотренных исключений.

Если обработчик непредусмотренных исключений отсутствует, то вызывается функция **terminate**(). По умолчанию эта функция вызывает функцию **abort**(), которая аварийно завершает текущий процесс.

Для определения собственной функции завершения используется функция **set_terminate**():

```
void my_terminate(){<обработка завершения>}
…
set_terminate(my_terminate);
```

Функция **set_terminate()** также возвращает адрес предыдущей программы обработки завершения.

Завершающая обработка (Ex9_4)

```
#include <iostream>
using namespace std;
void term_func()
{
   cout << "term_func was called by terminate." << endl;
   system("pause");
   exit( -1 );
}
int main()
{
    try
    { set_terminate( term_func );
      throw "Out of memory!";    }
    catch( int )
    { cout << "Integer exception raised." << endl; }
    return 0;
}
```

Запускать без отладчика !!

9.3 Механизм структурного управления исключениями C

Для перехвата исключения в языке C используется конструкция:

```
__try {<Защищенный код>}
__except(<Фильтрующее выражение>) {<Обработка исключений>}
```

Фильтрующее выражение может принимать следующие значения:

- 1 = EXCEPTION_EXECUTE_HANDLER – управление должно быть передано на следующий за ним обработчик исключения (при этом по умолчанию при обратном просмотре стека вызовов активизируются деструкторы всех локальных объектов, созданных между местом генерации исключения и найденным обработчиком);
- 0 = EXCEPTION_CONTINUE_SEARCH – производится поиск другого обработчика;
- -1 = EXCEPTION_CONTINUE_EXECUTION – управление возвращается в то место, где было обнаружено исключение без обработки исключения (отмена исключения).

В качестве фильтрующего выражения обычно используется функция, которая возвращает одно из указанных выше трех значений.

Получение информации об исключении

Для получения информации об исключении используют:

`_exception_code()` – возвращает код исключения.

`_exception_info()` – возвращает указатель на структуру **`EXCEPTION_POINTERS`**, содержащую описание исключения:

```
struct exception_pointers {
        EXCEPTION_RECORD *ExceptionRecord,
        CONTEXT *ContextRecord }
struct EXCEPTION_RECORD
 {      DWORD ExceptionCode;     // код завершения
        DWORD ExceptionFlags;    // флаг возобновления
        struct EXCEPTION_RECORD *ExceptionRecord;
        void *ExceptionAddress;     // адрес исключения
        DWORD NumberParameters;  // количество аргументов
        DWORD ExceptionInformation
        [EXCEPTION_MAXIMUM_PARAMETERS];  /* адрес массива

                                            параметров */
 };
```

Получение информации об исключении (2)

Существует ограничение на вызов этих функций: они могут вызываться только из блока __except().

Фильтрующая функция не может вызывать **_exception_info()**, но результат этого вызова можно передать в качестве параметра. Так например:

```
__except (filter_func(xp = _exception_code()))
{/* получение информации об исключении */ }
```

или с использованием составного оператора:

```
__except((xp = _exception_info()),
                        filter_func(xp))
```

.

Обработка аппаратных и программных исключений Windows (Ex9_5)

```
#include <EXCPT.H>
#include <iostream>
using namespace std;
int main()
{
    int* p = 0x00000000;    // NULL
    cout << "begin" << endl;
    __try
    {
        cout << "in try" <<endl;
        *p = 13;      // генерация исключения
    }
    __except(_exception_code(),1)
    { cout << "code =" <<hex<< _exception_code() << endl; }
    cout << "end" << endl;
    return 0;
}
```

```
begin
in try
code = c0000005
end
```

Генерация исключений в С

Для генерации исключения используется функция

void RaiseException(DWORD <код исключения**>,**
DWORD <флаг**>,**
DWORD <количество аргументов**>,**
const DWORD *<адрес массива 32 разрядных аргументов**>);**

где <код исключения> – число следующего вида:

биты 30-31: 11 – ошибка; 01 – информация; 10 – предупреждение;
бит 29: 1 – не системный код;
бит 28: 0 – резерв;

<флаг> может принимать значения:

EXCEPTION_CONTINUABLE (0) – обработка возобновима;
EXCEPTION_NONCONTINUABLE – обработка не возобновима (любая попытка продолжить процесс вызовет соответствующее прерывание).

Пример:

```
#define STATUS_1  0xE0000001
   ...
RaiseException(STATUS_1, 0, 0, 0);
```

Завершающая обработка

Структурное управление исключениями поддерживает также завершающую конструкцию, которая выполняется независимо от того, было ли обнаружено исключение при выполнении защищенного блока:

```
__try {<Защищенный блок> }
__finally {<Завершающая обработка>}
```

Совместное использование обычной и завершающей обработки исключений (Ex9_6)

```
#include <iostream>
using namespace std;
int main()
{   int* p = 0x00000000; // NULL
    cout << "Begin" << endl;
    __try{
            cout << "in try1" << endl;
            __try{
                   cout<< "in try2" << endl;
                   *p = 13;    // генерация исключения
                 }
            __finally{ puts("in finally"); }
          }
    __except(1){ cout<<"in except"<<endl; }
    cout<<"end"<<endl;
    return 0;
}
```

```
Begin
in try1
in try2
in finally
in except
end
```

9.4 Совместное использование различных механизмов обработки исключений

1) исключения Win32 можно обрабатывать только try…_ _except (C++) или _ _try…_ _except (C) или соответственно try…_ _finally (C++) или _ _try…_ _finally (C); оператор **`catch`** эти исключения игнорирует;

2) неперехваченные исключения Win32 не обрабатываются функцией обработки неперехваченных исключений и функцией **`terminate()`**, а передаются операционной системе, что обычно приводит к аварийному завершению приложения;

3) обработчики структурных исключений не получают копии объекта исключения, так как он не создается, а для получения информации об исключении используют специальные функции **`_exception_code()`** и **`_exception_info()`**.

Если возникает необходимость перехвата структурных исключений и исключений C++ для одной последовательности операторов, то соответствующие конструкции вкладываются одна в другую.

Пример совместного использования механизмов исключения (Ex9_7)

```
#include <string.h>
#include <iostream>
using namespace std;
class MyException          // класс исключения
 { private:  char* what; // динамическое поле сообщения
    public:  MyException(const char* s);
             MyException(const MyException& e );
             ~MyException();
             char* msg()const;
 };
MyException::MyException(const char* s = "Unknown")
              { what = _strdup(s);  }
MyException::MyException(const MyException& e )
              { what = _strdup(e.what);}
MyException::~MyException()
              { delete[] what;  }
char* MyException::msg() const
              { return what; }
```

Пример совместного использования механизмов исключения (2)

В пределах функции нельзя использовать исключения разного типа!

```
void f()
{int *p=NULL;
  __try    { *p=3;}
  __except(1){throw(MyException("Wrong pointer"));}}

void f1()
   {try { f();}
    catch(const MyException& e){ cout<<e.msg()<<endl;}
   }

int main(int argc, char* argv[])
 {__try
    {f1();}
  __finally  { cout<<"end"<<endl; }
  return 0;
 }
```

Wrong pointer
end

Обработка структурных исключений как исключений C++

Для преобразования структурных исключений в исключения C++ можно использовать функцию **_set_se_translator()**:

```
typedef void (* translator_function)
            ( unsigned int, struct _EXCEPTION_POINTERS* );
translator_function _set_se_translator
                    (translator_function se_trans_func);
```

В качестве параметра функции **_set_se_translator** необходимо передать адрес функции-переходника, которая назначает для структурных исключений вызов соответствующих исключений C++, например:

```
void trans_func( unsigned int u,EXCEPTION_POINTERS* pExp)
   {throw SE_Exception(u);}
```

Функция возвращает указатель на предыдущую функцию **_set_se_translator()**

Трансляция структурных исключений (Ex9_8)

```
#include <stdio.h>
#include <windows.h>
#include <eh.h>

class SE_Exception  // класс для обработки структурных
                    // исключений
{private:  unsigned int nSE;
 public:
    SE_Exception() {}
    SE_Exception( unsigned int n ) : nSE( n ) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};

void SEFunc() // пример функции с исключением
{    _ _try {  int x, y=0; x = 5 / y; } // исключение!
    _ _finally  { printf( "In finally\n" ); }
}
```

Трансляция структурных исключений (2)

```
void trans_func( unsigned int u, EXCEPTION_POINTERS* pExp)
{   printf( "In trans_func.\n" );
    throw SE_Exception(u);
}
void main( void)
{
    try
    {   _set_se_translator( trans_func );
        SEFunc();
    }
    catch( SE_Exception e )
    {   printf( "%x.\n",e.getSeNumber() );   }
}
```

```
In trans_func.
In finally
c0000094.
Press any key to continue
```