Информатика. Семинар №12

Исключения, умные указатели, устройство std::vector

Исключения

- Сложно обработать все проблемные ситуации: деление на 0, нехватка памяти, некорректное обращение к памяти и т.п.
- Для программ, работающих в режиме 24/7 «падения» не допустимы

Исключения

Пример: перехват системного исключения "деление на ноль“

```
int x = 0;
try {
        std::cout <<2/x; //Здесь произойдет выброс
        исключения
        // Последующие операторы выполняться не
        будут
}
catch (...) {
        std::cout << "Division by zero" << std::endl;
}
```

Исключения

Обработка исключений, возбужденных оператором throw, идет по следующей схеме:
1. Создается статическая переменная со значением, заданным в операторе throw.
Она будет существовать до тех пор, пока исключение не будет обработано.
Если переменная-исключение является объектом класса, при ее создании работает конструктор копирования.
2. Завершается выполнение защищенного try-блока:
раскручивается
стек подпрограмм,
вызываются деструкторы для тех объектов, время жизни которых истекает и т.д.
3. Выполняется поиск первого из catch-блоков, который пригоден для обработки созданного
исключения.

Исключения

try {операторы защищенного блока}
{catch-блоки}…

Catch-блок имеет один из следующих форматов:
catch (тип) {обработчик ошибочной ситуации}
catch (тип идентификатор) {обработчик ошибочной ситуации}
catch (…) {обработчик ошибочной ситуации}

Вопрос №1

```cpp
#include <iostream>
using namespace std;
int main()
{
   int x = -1;
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw \n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }

   cout << "After catch \n";
   return 0;
}
```

A

```
Inside try
Exception Caught
After throw
After catch
```

B

```
Inside try
Exception Caught
After catch
```

C

```
Inside try
Exception Caught
```

D

```
Inside try
After throw
After catch
```

Вопрос №1

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = -1;
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }

    cout << "After catch \n";
    return 0;
}
```

A

```
Inside try
Exception Caught
After throw
After catch
```

```
Inside try
Exception Caught
After catch
```

C

```
Inside try
Exception Caught
```

D

```
Inside try
After throw
After catch
```

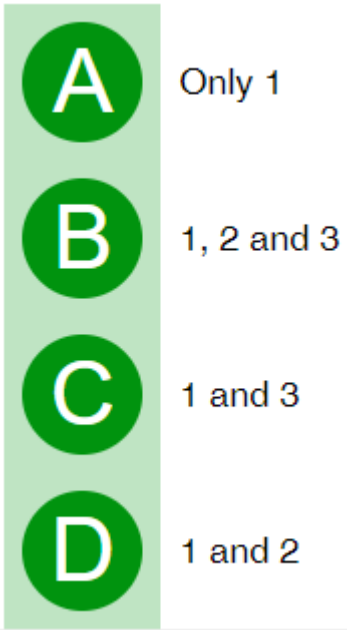
Вопрос №2

What is the advantage of exception handling?

```
1) Remove error-handling code from the software's main line of code.

2) A method writer can chose to handle certain exceptions and delegate
   others to the caller.

3) An exception that occurs in a function can be handled anywhere in
   the function call stack.
```



Вопрос №2

What is the advantage of exception handling?

```
1) Remove error-handling code from the software's main line of code.

2) A method writer can chose to handle certain exceptions and delegate
   others to the caller.

3) An exception that occurs in a function can be handled anywhere in
   the function call stack.
```

Вопрос №3

Output of following program

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

class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   try {
        throw d;
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
   catch(Derived d) {
        cout<<"Caught Derived Exception";
   }
   return 0;
}
```

A. Caught Derived Exception

B. Caught Base Exception

C. Compiler Error

Вопрос №3

Output of following program

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

class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   try {
       throw d;
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
   catch(Derived d) {
        cout<<"Caught Derived Exception";
   }
   return 0;
}
```

A. Caught Derived Exception

Caught Base Exception

C. Compiler Error

Вопрос №4

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

int main()
{
    try
    {
        try
        {
            throw 20;
        }
        catch (int n)
        {
            cout << "Inner Catch\n";
            throw;
        }
    }
    catch (int x)
    {
        cout << "Outer Catch\n";
    }
    return 0;
}
```

A. `Outer Catch`

B. `Inner Catch`

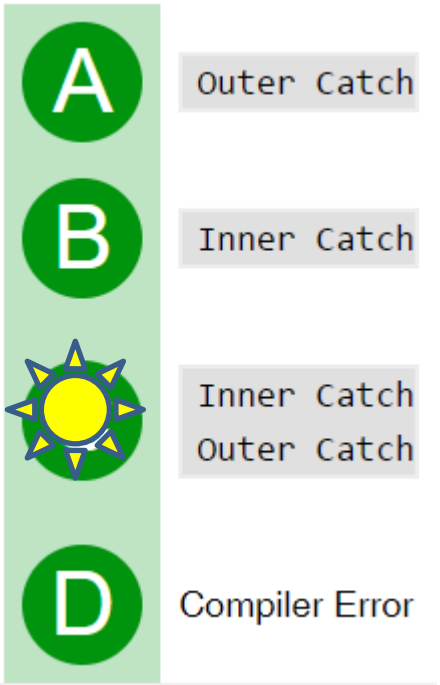
C. `Inner Catch`
`Outer Catch`

D. Compiler Error

Вопрос №4

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

int main()
{
    try
    {
        try
        {
            throw 20;
        }
        catch (int n)
        {
            cout << "Inner Catch\n";
            throw;
        }
    }
    catch (int x)
    {
        cout << "Outer Catch\n";
    }
    return 0;
}
```



Вопрос №5

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

class Test {
public:
   Test() { cout << "Constructing an object of Test " << endl; }
  ~Test() { cout << "Destructing an object of Test "  << endl; }
};

int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

A

```
Caught 10
```

B

```
Constructing an object of Test
Caught 10
```

C

```
Constructing an object of Test
Destructing an object of Test
Caught 10
```

D

Compiler Errror

Вопрос №5

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

class Test {
public:
   Test() { cout << "Constructing an object of Test " << endl; }
  ~Test() { cout << "Destructing an object of Test "  << endl; }
};

int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

A

```
Caught 10
```

B

```
Constructing an object of Test
Caught 10
```

```
Constructing an object of Test
Destructing an object of Test
Caught 10
```

D

Compiler Errror

Вопрос №6

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

class Test {
  static int count;
  int id;
public:
  Test() {
    count++;
    id = count;
    cout << "Constructing object number " << id << endl;
    if(id == 4)
       throw 4;
  }
  ~Test() { cout << "Destructing object number " << id << endl; }
};

int Test::count = 0;

int main() {
  try {
    Test array[5];
  } catch(int i) {
    cout << "Caught " << i << e
  }
}
```

A

```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 1
Destructing object number 2
Destructing object number 3
Destructing object number 4
Caught 4
```

C

```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 4
Destructing object number 3
Destructing object number 2
Destructing object number 1
Caught 4
```

D

```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 1
Destructing object number 2
Destructing object number 3
Caught 4
```

```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 3
Destructing object number 2
Destructing object number 1
Caught 4
```

Вопрос №6

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

class Test {
  static int count;
  int id;
public:
  Test() {
    count++;
    id = count;
    cout << "Constructing object number " << id << endl;
    if(id == 4)
       throw 4;
  }
  ~Test() { cout << "Destructing object number " << id << endl; }
};

int Test::count = 0;

int main() {
  try {
    Test array[5];
  } catch(int i) {
    cout << "Caught " << i << e
  }
}
```

A

```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 1
Destructing object number 2
Destructing object number 3
Destructing object number 4
Caught 4
```

C

```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 4
Destructing object number 3
Destructing object number 2
Destructing object number 1
Caught 4
```

D

```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 1
Destructing object number 2
Destructing object number 3
Caught 4
```

```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 3
Destructing object number 2
Destructing object number 1
Caught 4
```

Вопрос №7

What happens in C++ when an exception is thrown and not caught anywhere like following program.

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

int fun() throw (int)
{
    throw 10;
}

int main() {

  fun();

  return 0;
}
```

A. Compiler error

B. Abnormal program termination

C. Program doesn't print anything and terminates normally

D. None of the above

Вопрос №7

What happens in C++ when an exception is thrown and not caught anywhere like following program.

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

int fun() throw (int)
{
    throw 10;
}

int main() {

  fun();

  return 0;
}
```

A. Compiler error

Abnormal program termination

C. Program doesn't print anything and terminates normally

D. None of the above

When an exception is thrown and not caught, the program terminates abnormally.

Деструкторы не должны бросать исключения

```
{
    std::vector<Widget> v; // 10 окон было создано
    // 0-е бросает исключение -> по правилам нужно сначала почистить все локальные объекты, потом прокинуть вверх это исключение –> при удалении 1-го окна снова исключение … два одновременно живущих исключения – Undefined behavior
}
```

Исключения в конструкторах

- Описание проблемы
- https://habrahabr.ru/post/59349/

Операторы new, delete

- Отличие от malloc, calloc и т.п. в том, что new = malloc + n вызовов конструктора string::string()
- Массивы простых типов (int, float) не инициализированы, т.к. у них нет конструкторов
- delete [] s;

```
int n = 42;
std::string* s = new std::string[n];
```

Зачем нужно выделять память на в куче?

- Размер кучи >> размера стека
- Время жизни данных в куче – вплоть до delete
- Время жизни данных на стеке – до соответствующей }

Как и кто должен выделять память?

- Правило: тот, кто выделил память должен её удалить.

```
void f(float x, float y, float a[2])
{
  a[0] = x + y;
  a[1] = x - y;
}

int main() {

  // float b[2];
  float* b = new float[2];
  f(1.0f, 2.0f, b);

  std::cout << b[0] << " " << b[1];
  delete[] b;
```

Откуда такое правило?

- Если вы пользуетесь «чужой» ф-ей, которая выделяет некоторый объём памяти, вам необходимо помнить/знать о необходимости подчищать за другими.
- Если используете библиотеки, собранные другими версиями компиляторов, то при попытке почистить память, выделенную в ней, программа упадёт.

«Умные» указатели

```
#include·<memory>

struct·A·{
··A()··{·std::cout·<<·"Hi,·I'm·A"·<<·std::endl;·}
··~A()·{·std::cout·<<·"Bye.·A."···<<·std::endl;·}
};

void·f(A*·a)·{
··a·=·new·A();
}

int·main()·{
··{
····A*·a·=·nullptr;·/*·в·C++11·для·нулевого·указателя·
·····сделали·специальную·константу·вместо·целочисленного·0·или·NULL·*/
····f(a);
··}
```

«Умные» указатели

Smart pointer — это объект, работать с которым можно как с обычным указателем, но при этом, в отличии от последнего, он предоставляет некоторый дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).

shared_ptr реализует подсчет ссылок на ресурс. Ресурс освободится тогда, когда счетчик ссылок на него будет равен 0

http://archive.kalnitsky.org/2011/11/02/smart-pointers-in-cpp11/

«Умные» указатели

```
#include <memory>

struct A {
  A(int x, int y) { std::cout << "Hi, I'm A" << std::endl; }
  ~A() { std::cout << "Bye. A." << std::endl; }
  void Do() { std::cout << "Do work" << std::endl; }
};

void f(A* a) {
  a->Do();
}

int main() {
  std::shared_ptr<A> a0;
  {
    std::shared_ptr<A> a = std::make_shared<A>(1, 2);
    f(a.get());
    // a0 = a;
  }
```

Как быть с массивами?

- http://stackoverflow.com/questions/13061979/shared-ptr-to-an-array-should-it-be-used

```
std::shared_ptr<int> a0(new int[10], std::default_delete<int[]>());
std::shared_ptr<int> a1(new int[10], [](int *p) { delete[] p; });
```

P.S. В этом случае лучше использовать std::vector. Если необходим массив из малого числа элементов и область видимости позволяет, то лучше выделить массив на стеке.

Детали реализации std::vector

- Амортизированная стоимость (среднее число операций в худшем случае – T(n) / n) операции push_back: O(1).

Детали реализации std::vector

- Знаем, что у std::vector есть **capacity** (выделено) и **size** (использовано).
- Реализовать на основе T* data = new T[n] не хочется, т.к. лишний раз будет вызван конструктор «по умолчанию»
- Т.е. процесс выделения памяти от процесса вызова конструктора надо отделить: выделить память можно с помощью malloc

Детали реализации std::vector

- Placement new - вызов конструктора на уже выделенной памяти:

```
void SimplePushBack(const T& t) {
        new (data + size) T(t);
        size++;
}
```

- Как удалить объект, который так создали?

T* p = …

Явно вызываем деструктор:

p->~T();

Детали реализации std::vector

- Если не получается выделить память нужного размера, то желательно как-то оповещать пользователя об этом -> бросаем исключение.

```
void PushBack(const value_type& x)
{
  // without reserving
  try
  {
    new (data_ + size_) value_type(x);
    ++size_;
  }
  catch (...)
  {
    throw;
  }
}
```

Детали реализации std::vector

Но лучше использовать умный указатель std::unique … в идеале теперь просто никогда не используем new и delete явно.

```
char* data_;

explicit vector(size_type n):
  capacity_(n),
  size_(0),
  data_(new char[capacity_ * sizeof(value_type)])
{
}
```

Детали реализации std::vector

- Как реализовать оператор присваивания?
- Правило **commit or rollback**.
- Идиома **copy and swap** – простой способ избежать утечек памяти, если неожиданно память кончилась и оператор new не отработал до конца (конструктор копирования должен быть реализован).

P.S. https://ru.wikipedia.org/wiki/Copy-and-swap

Упражнение 1

- Реализовать свой класс String фиксированного размера, реализующий конструктор по умолчанию, конструктор копирования, String(const char* s), String(size_t length, char value = 0), деструктор, оператор [] константный и неконстантный, Length, Swap, оператор = и ==, оператор !=
- Продвинутый вариант: реализовать подобие шаблонного класса Vector (размер динамически увеличивается и уменьшается)