


АКАДЕМИЯ АЙТИ

Гарантии безопас-ности исключений

Дубров Денис Владимирович
Ведущий инженер-программист, к.ф.-м.н.

denis.dubrov@harman.com

Задача управления ресурсами

Пример (X.h)

```
class X : Y
{
  // ...
private:
  std::string m_Name;
  Data1 *m_pData1;
  Data2 *m_pData2;
  Data3 *m_pData3;
  void clear();
};
```

Пример (X.cpp)

```
X::~X()
{
  clear();
}

void X::clear()
{
  delete m_pData3;
  delete m_pData2;
  delete m_pData1;
}
```

Задача управления ресурсами (продолжение)

Пример

```
X::X(
  const std::string &rcName, Params p)
  : m_Name(rcName),
    m_pData1(nullptr),
    m_pData2(nullptr),
    m_pData3(nullptr)
{
  try
  {
    m_pData1 = new Data1(p);
    m_pData2 = new Data2(func(p));
```

Пример (окончание)

```
    m_pData3 = new Data3(
      this, m_pData2);
    // Data3(Y *, Data2 *)
  }
  catch (...)
  {
    clear();
    throw;
  }
}    // X::X(name, params)
```

Задача управления ресурсами (продолжение)

Пример (X.h)

```
class X : Y
{
  // ...
private:
  std::string m_Name;
  Data1 *const m_cpData1;
  Data2 *const m_cpData2;
  Data3 *const m_cpData3;
  void clear();
};
```

Пример (X.cpp)

```
X::X(
  const std::string &rcName, Params p)
  : m_Name(rcName),
    m_cpData1(new Data1(p)),
    m_cpData2(new Data2(func(p))),
    m_cpData3(new Data3(this, m_pData2))
{
  // try?..
}
```

Задача управления ресурсами (продолжение)

Пример

```
void f()
{
  // ...
  try
  {
    X x(name, params);
    // ...
  }
```

Пример (окончание)

```
  catch (...)
  {
    // x.clear() ?
  }
}   // f()
```

Задача управления ресурсами (продолжение)

Пример

```
X::X(
  const std::string &rcName, Params p)
try
  : m_Name(rcName),
    m_cpData1(new Data1(p)),
    m_cpData2(new Data2(func(p))),
    m_cpData3(new Data3(this, m_pData2))
{
  // Пусто
}
```

Пример (окончание)

```
catch (...)
{
  // ?..
}
```

Получение ресурса есть инициализация

Определение

Получение ресурса есть инициализация: (Resource Acquisition Is Initialization, RAII) — парадигма программирования, предусматривающая выделение ресурса только в конструкторе некоторого объекта, а освобождение — в его деструкторе.

Интеллектуальный указатель

Определение

Интеллектуальный указатель: (Smart Pointer) — класс, имитирующий синтаксис и семантику обычного указателя на объект и предоставляющий дополнительную функциональность (владение объектом и т. д.)

Определение boost::scoped_ptr

Объявление

```
template <class T> class scoped_ptr
{
  T *px;
public:
  explicit scoped_ptr(T *p = 0) : px(p) { }
  ~scoped_ptr() { delete px; }
  T &operator * () const { return *px; }
  T *operator -> () const { return px; }
  T *get() const { return px; }
  operator bool () const { return px != 0; }
  bool operator ! () const { return px == 0; }
```

Пример (использование)

```
scoped_ptr <Data> ptr(
  new Data());
f(*ptr);
ptr->do_something();
g(ptr.get());
if (ptr) { /*...*/ }
if (!ptr) { /*...*/ }
```

Определение boost::scoped_array

Объявление

```
template <class T> class scoped_array
{
  T *px;
public:
  // ...
  ~scoped_array() { delete [] px; }
  T &operator [] (std::ptrdiff_t i) const
    { return px[i]; }
  // ...
```

Пример (использование)

```
scoped_array <char> ptr(
  new char[100]);

ptr[10] = 'A';
```

Решение задачи управления ресурсами

Пример (X.h)

```
class X : Y
{
  // ...
private:
  std::string m_Name;
  const boost::scoped_ptr <Data1> m_cPData1;
  const boost::scoped_ptr <Data2> m_cPData2;
  const boost::scoped_ptr <Data3> m_cPData3;
};
```

Решение задачи управления ресурсами (окончание)

Пример (X.cpp)

```
X::X(const std::string &rcName, Params p)
  : m_Name(rcName),
    m_cPData1(new Data1(p)),
    m_cPData2(new Data2(func(p))),
    m_cPData3(new Data3(this, m_pData2))
{
  // Пусто
}
```

Уровни безопасности исключений

Определения (Д. Абрахамс)

Базовая гарантия: отсутствуют утечки памяти, данные находятся в согласованном (не обязательно предсказуемом) состоянии.

Строгая гарантия: (семантика принятия или отката) — прерывание операции из-за исключения приводит к состоянию всех данных, существовавшему до начала операции.

Гарантия нейтральности: все генерируемые исключения передаются в неизменном виде вызывающей функции.

Гарантия отсутствия исключений: генерируемые исключения не покидают пределов функции ни при каких обстоятельствах.

Задача Каргилла

Пример

```
template <class T> class Stack
{
public:
  Stack();
  Stack(const Stack &);
  ~Stack();
  Stack &operator = (const Stack &);
  size_t size() const;
  void push(const T &)
  T pop();
```

Пример (окончание)

```
  // ...
private:
  T *m_pT;
  size_t m_uResv, m_uUsed;
};
```

Конструктор по умолчанию

Пример

```
template <class T> Stack <T>::Stack()
  : m_pT(0),
    m_uResv(10),
    m_uUsed(0)
{
  m_pT = new T[m_uResv];
}
```

Деструктор

Пример

```
template <class T> Stack <T>::~Stack()
{
  delete [] m_pT;
}
```

Вспомогательная функция копирования

Пример

```
template <class T> T *newCopy(
  const T *pSrc, size_t uUsedSrc,
  size_t uResvDst)
{
  assert(uResvDst >= uUsedSrc);
  T *pDst = new T[uResvDst];
  try
  {
    std::copy(
      pSrc, pSrc + uUsedSrc, pDst);
  }
```

Пример (окончание)

```
  catch (...)
  {
    delete [] pDst;
    throw;
  }
  return pDst;
}    // newCopy()
```

Конструктор копирования

Пример

```
template <class T> Stack <T>::Stack(
  const Stack &rcStack)
  : m_pT(newCopy(rcStack.m_pT, rcStack.m_uUsed, rcStack.m_uResv)),
    m_uResv(rcStack.m_uResv),
    m_uUsed(rcStack.m_uUsed)
{
  //
}
```

Операция копирующего присваивания

Пример

```
template <class T> Stack <T> &Stack <T>::operator = (const Stack &rcStack)
{
  if (this != &rcStack)
  {
    T *pNew = newCopy(rcStack.m_pT, rcStack.m_uUsed, rcStack.m_uResv);
    delete [] m_pT;
    m_pT = pNew;
    m_uResv = rcStack.m_uResv;
    m_uUsed = rcStack.m_uUsed;
  }
  return *this;
}
```

Получение размера

Пример

```
template <class T> size_t Stack <T>::size() const
{
  return m_uUsed;
}
```

Добавление элемента

Пример

```
template <class T> void Stack <T>::push(
  const T &rcT)
{
  if (m_uUsed == m_uResv)
  {
    size_t uResvNew = 2 * m_uResv + 1;
    T *pNew = newCopy(m_pT, m_uUsed, uResvNew);
    delete [] m_pT;
    m_pT = pNew;
    m_uResv = uResvNew;
  }
```

Пример (окончание)

```
  m_pT[m_uUsed] = rcT;
  ++ m_uUsed;
}    // push()
```

Удаление элемента

Пример

```
template <class T> T Stack <T>::pop()
{
  if (0 == m_uUsed)
    throw std::range_error("Empty stack");
  else
  {
    T t = m_pT[m_uUsed - 1];
    -- m_uUsed;
    return t;
  }
}
```

Пример (окончание)

```
Stack <Data> stack;
Data data1;
// ...
data1 = stack.pop();
```

Удаление элемента, возврат по ссылке

Пример

```
template <class T> Stack void <T>::pop(T &rT)
{
  if (0 == m_uUsed)
    throw std::range_error("Empty stack");
  else
  {
    rT = m_pT[m_uUsed - 1];
    -- m_uUsed;
  }
}
```

Разделение удаления и возврата элемента

Пример

```
template <class T>
  T &Stack <T>::top()
{
  if (0 == m_uUsed)
    throw std::range_error(
      "Empty stack");
  else
    return m_pT[m_uUsed - 1];
}
```

Пример (окончание)

```
template <class T>
  void Stack <T>::pop()
{
  if (0 == m_uUsed)
    throw std::range_error(
      "Empty stack");
  else
    -- m_uUsed;
}
```

Возврат по ссылке на const

Пример

```
template <class T>
  const T &Stack <T>::top() const
{
  // ...
}
```

Требования к типу хранимых данных

Требование	Использование
`T()`	создание буфера
`T(const T &)`	pop(), возвращающее значение
`~T()`, не генерирующий исключений	гарантия безопасности
`T & operator = (const T &)`, безопасная	установка значений в буфере

Таблица: требования к типу хранимых данных (`T`)

Вспомогательный класс

Пример

```
template <class T> class StackImpl
{
protected:
  T *m_pT;
  size_t m_uResv, m_uUsed;
  explicit StackImpl(size_t uResv = 0);
  ~StackImpl();
  void swap(StackImpl &) noexcept;
private:
  StackImpl(const StackImpl &);
  StackImpl &operator = (const StackImpl &);
};
```

Конструктор по умолчанию

Пример

```
template <class T> StackImpl <T>::StackImpl(size_t uResv)
  : m_pT(static_cast <T *> (
      0 == uResv ? nullptr : operator new (uResv * sizeof (T)))),
    m_uResv(uResv),
    m_uUsed(0)
{
  //
}
```

Вспомогательные функции

Пример

```
template <class T1, class T2>
  void construct(T1 *pT1, const T2 &rcT2)
{
  new (pT1) T1(rcT2);
}

template <class T>
  void destroy(T *pT)
{
  pT->~T();
}
```

Пример (окончание)

```
template <class FwdIter>
  void destroy(
    FwdIter first,
    FwdIter last)
{
  while (first != last)
  {
    destroy(&*first);
    ++ first;
  }
}
```

Деструктор

Пример

```
template <class T> StackImpl <T>::~StackImpl()
{
  destroy(m_pT, m_pT + m_uUsed);
  operator delete (m_pT);
}
```

Функция обмена

Пример

```
template <class T>
  void StackImpl <T>::swap(StackImpl &rImpl) noexcept
{
  std::swap(m_pT,    rImpl.m_pT);
  std::swap(m_uResv, rImpl.m_uResv);
  std::swap(m_uUsed, rImpl.m_uUsed);
}
```

Объявление усовершенствованного стека

Пример

```
template <class T> class Stack : private StackImpl <T>
{
public:
  explicit Stack(size_t uResv = 0);
  Stack(const Stack &);
  Stack &operator = (const Stack &);
  void swap(Stack &) noexcept;
  size_t size() const;
  void push(const T &);
  T &top();
  void pop();
};
```

Конструктор по умолчанию

Пример

```
template <class T> Stack <T>::Stack(size_t uResv)
  : StackImpl <T>(uResv)
{
  //
}
```

Конструктор копирования

Пример

```
template <class T> Stack <T>::Stack(const Stack &rcStack)
  : StackImpl <T>(rcStack.m_uUsed)
{
  while (this->m_uUsed < rcStack.m_uUsed)
  {
    construct(this->m_pT + this->m_uUsed, rcStack.m_pT[this->m_uUsed]);
    ++ this->m_uUsed;
  }
}
```

Операция копирующего присваивания

Пример

```
template <class T>
  Stack <T> &Stack <T>::operator = (const Stack &rcStack)
{
  Stack tempStack(rcStack);
  swap(tempStack);
  return *this;
}
```

Операция копирующего присваивания

Пример

```
template <class T>
  Stack <T> &Stack <T>::operator = (Stack stack)
{
  swap(stack);
  return *this;
}
```

Добавление элемента

Пример

```
template <class T>
  void Stack <T>::push(const T &rcT)
{
  if (this->m_uUsed == this->m_uResv)
  {
    Stack tempStack(2 * this->m_uResv + 1);
    while (tempStack.size() < this->m_uUsed)
      tempStack.push(
        this->m_pT[tempStack.size()]);
    tempStack.push(rcT);
    swap(tempStack);
```

Пример (окончание)

```
  }
  else
  {
    construct(
      this->m_pT +
      this->m_uUsed,
      rcT);
    ++ this->m_uUsed;
  }
}    // push()
```

Удаление элемента

Пример

```
template <class T> void Stack <T>::pop()
{
  if (0 == this->m_uUsed)
    throw std::range_error("Empty stack");
  else
  {
    -- this->m_uUsed;
    destroy(this->m_pT + this->m_uUsed);
  }
}
```

Требования к типу хранимых данных

Требование	Использование
`T()`	создание буфера
`T(const T &)`	pop(), возвращающее значение
`~T()`, не генерирующий исключений	гарантия безопасности
`T & operator = (const T &)`, безопасная	установка значений в буфере

Таблица: требования к типу хранимых данных (`T`)

Требования к типу хранимых данных

Требование	Использование
`T(const T &)`	pop(), возвращающее значение
`~T()`, не генерирующий исключений	гарантия безопасности

Таблица: требования к типу хранимых данных (`T`)

Безопасность стандартной библиотеки C++

Гарантии

- Итераторы контейнеров безопасны и копируемы без генерации исключений.
- Контейнеры реализуют базовую гарантию.
- Гарантия отсутствия исключений у `std::swap()`, `std::allocator <T>::deallocate()`, деструкторов и т. д.
- Все операции контейнеров (кроме двух) реализуют строгую гарантию.

Следствие

Все операции уничтожения не генерируют исключений ($\Rightarrow$ деструкторы объектов, хранимых в контейнерах, не должны генерировать исключений).

Базовая гарантия исключений

Методы

- Множественная вставка: `insert(pos, first, last)`
- Вставка/удаление у `std::vector` и `std::deque` строго безопасны, если `T::T(const T &)` и `T::operator = (const T &)` не генерируют исключений.

Базовая гарантия исключений

Методы

- Множественная вставка: `insert(pos, first, last)`
- Вставка/удаление у `std::vector` и `std::deque` строго безопасны, если `T::T(const T &)` и `T::operator = (const T &)` не генерируют исключений.

Примеры (не строго безопасные контейнеры)

- `std::vector <std::string>`
- `std::vector <std::vector <int> >`

Семантика транзакции при множественной вставке

Пример

```
void MyClass::insert(InpIter first, InpIter last)
{
  MyVector tempVector(m_Vector);
  tempVector.insert(tempVector.begin(), first, last);
  std::swap(m_Vector, tempVector);
}
```

Безопасность выделения ресурсов

Пример

```
void f(T1 *, T2 *);

int main()
{
  f(new T1, new T2);
  // ...
}
```

Безопасность выделения ресурсов

Пример

```
void f(std::auto_ptr <T1>, std::auto_ptr <T2>);

int main()
{
  f(std::auto_ptr <T1> (new T1), std::auto_ptr <T2> (new T2));
  // ...
}
```

Безопасность выделения ресурсов

Пример

```
void f(std::auto_ptr <T1>, std::auto_ptr <T2>);

int main()
{
  std::auto_ptr <T1> p1(new T1);
  std::auto_ptr <T2> p2(new T2);
  f(p1, p2);
  // ...
}
```

Точка следования

Пример

```
int main()
{
  int i = 0;
  cout
    << (i += 1) << ' '
    << (i += 1) << ' '
    << (i += 1) << endl;
}
```

Точка следования

Пример

```
int main()
{
  int i = 0;
  cout.operator << (i += 1).operator << (i += 1).operator << (i += 1);
}
```

Точка следования

Пример

```
int main()
{
  int i = 0;
  x.f1(++ i).f2(++ i).f3(++ i);
}
```

Точка следования

Пример

```
int main()
{
  int i = 0;
  // x.f1(++ i).f2(++ i).f3(++ i);
  f3(f2(f1(x, ++ i), ++ i), ++ i);
}
```

Точка следования (продолжение)

Определение (C++98, C++03)

Точка следования: (*S*equence Point) — место программы, в котором все побочные эффекты от предыдущих вычислений должны завершиться, а от последующих — ещё не начаться.

Точка следования (продолжение)

Точки следования

- Окончание вычисления полного выражения;
- После завершения вычисления аргументов функции до начала выполнения её тела.
- После копирования возвращаемого значения функции до вычисления любых выражений вне её.
- После вычисления *первого* выражения в:
  - `expr1 && expr2`
  - `expr1 || expr2`
  - `expr1, expr2`
  - `expr1 ? expr2 : expr3`

Точка следования (окончание)

Правило

Между соседними точками следования скалярное значение должно меняться не более 1 раза, причём предыдущее значение должно считываться только для определения сохраняемого значения.

Примеры (неопределённое поведение)

```
i = an[i ++];
i = ++ i + 1;
n = i ++ + ++ i;
n = f(++ i, ++ i);
```

Порядок вычисления аргументов функций

Правила

- До вызова функции *все* её аргументы должны быть полностью вычислены ($\supset$ побочные эффекты);
- После начала выполнения функции никакие выражения вызывающей функции не начинают и не продолжают выполняться, пока не завершится вызывающая функция ($\Rightarrow$ выполнения функций не чередуются).
- Вычисление аргументов функций возможно в любом порядке, включая чередование (если не оговорено другими правилами).

Пример функции

Пример (Г. Саттер)

```
String EvaluateSalaryAndReturnName(Employee e)
{
  if (e.title() == "CEO" || e.salary() > 100000)
  {
    std::cout <<
      e.first() << " " << e.last() <<
      " is overpaid" << std::endl;
  }
  return e.first() + " " + e.last();
}
```

Пример функции, удовлетворяющей строгой гарантии

Пример (Г. Саттер)

```
std::auto_ptr <String> EvaluateSalaryAndReturnName(Employee e)
{
  std::auto_ptr <String> resPStr =
    new String(e.first() + " " + e.last());
  if (e.title() == "CEO" || e.salary() > 100000)
  {
    String strMessage = *resPStr + " is overpaid\n";
    std::cout << strMessage;
  }
  return resPStr;
}
```

Безопасное присваивание

Пример (X.h)

```
class X
{
public:
  X &operator = (const X &);
  // ...
private:
  Data1 m_Data1;
  Data2 m_Data2;
};
```

Пример (main.cpp)

```
int main()
{
  X x1, x2;
  // ...
  x2 = x1;
  // ...
}
```

Безопасное присваивание с реализацией

Пример (X.h)

```
class X
{
public:
  X();
  X(const X &);
  ~X();    // явно
  X &operator = (const X &); // ...
private:
  class XImpl;
  std::auto_ptr <XImpl> m_PImpl;
};
```

Пример (X.cpp)

```
class X::XImpl
{
public:
  Data1 m_Data1;
  Data2 m_Data2; // ...
};

X::~X()
{
  //
}
```

Реализация обмена и присваивания

Пример

```
void X::swap(X &rX) noexcept
{
  // std::swap(
  //   m_PImpl, rX.m_PImpl);
  //
  std::auto_ptr <XImpl> tempImpl(
    m_PImpl);
  m_PImpl = rX.m_PImpl;
  rX.m_PImpl = tempImpl;
}
```

Пример (окончание)

```
X::X(const X &rcX)
  : m_PImpl(new XImpl(*rcX.m_PImpl))
{
  //
}

X &X::operator = (X x)
{
  swap(x);
  return *this;
}
```

Локальная строгая гарантия

Определение

Локальная строгая гарантия: при генерации исключения состояние программы остаётся неизменным по отношению к контролируемым объектам (включая ранее полученные ссылки и итераторы).

Правила обеспечения безопасности

Правила разработки

- Разрабатываемый код должен передавать наружу все неизвестные исключения для дальнейшей обработки в вызывающей функции.
- Деструкторы, перегружаемые операции **operator delete** () и **operator delete** [] (), а также операции освобождения ресурсов не должны позволять исключениям выходить за свои пределы.
- Каждая функция, класс, модуль должны отвечать за одну чётко поставленную задачу.

Правила обеспечения безопасности (продолжение)

Правила разработки (продолжение)

- В каждой функции следует собирать весь код, который может сгенерировать исключение, и выполнять его отдельно, безопасным с точки зрения исключений способом. Только после его завершения следует менять состояние программы, а также освобождать ресурсы при помощи операций, не генерирующих исключений (например, обмена указателей).
- Для управления ресурсами следует использовать идиому захвата ресурсов при инициализации.
- Неуправляемое выделение ресурсов можно выполнять только в теле конструктора, но не в списке инициализации (освобождение — только в локальном блоке **`try`**).

Правила обеспечения безопасности (продолжение)

Правила разработки (продолжение)

- Явное выделение ресурсов необходимо производить в отдельном операторе, который сразу передаёт ресурс владеющему объекту.
- Все сгенерированные в конструкторе исключения от подобъектов покидают его пределы $\Rightarrow$ объект не создаётся. Для реализации необязательных частей объекта следует пользоваться идиомой скрытой реализации.
- Если для корректной работы перегруженной операции присваивания необходима проверка самоприсваивания, то скорее всего, операция не является строго безопасной.

Правила обеспечения безопасности (окончание)

Правила разработки (окончание)

- Обеспечение безопасности исключений влияет на структуру классов (в частности, на интерфейс), его не следует откладывать на более поздние сроки.
- Даже при использовании небезопасных классов код может всегда обеспечить локальную строгую гарантию безопасности (возможно, изменением объявления класса).
- Следует избегать лишних отношений наследования (также влияют на безопасность исключений).




АКАДЕМИЯ АЙТИ

Спасибо за внимание!

Центральный офис:
Москва, Варшавское шоссе 47,
корп.4, 10 этаж
Тел: +7 (495) 662-7894, 662-7895
Факс: +7(495) 974-7990
e-mail: academy@it.ru

www.academy.it.ru