

ЯЗЫКИ ПРОГРАММИРОВАНИЯ и СТРУКТУРЫ ДАННЫХ

Лекция 11

Обеспечение надёжности.

Обработка ошибок.

Обработка исключений.



<https://do.ssau.ru/moodle/course/view.php?id=1375>

<https://do.ssau.ru/moodle/mod/forum/view.php?id=34435>

Надежность ПС.

Показатели **надежности** характеризуют – *способность программного средства в конкретных областях применения выполнять заданные функции в соответствии с программными документами в условиях возникновения отклонений в среде функционирования, вызванных сбоями технических средств, ошибками во входных данных, ошибками обслуживания и другими дестабилизирующими воздействиями.*

Отказ в ПС – проявление в нем ошибки.

Надежное ПС не исключает наличия в ней ошибок.

Убедиться, что ПС обладает таким свойством можно при его испытании путем тестирования, а также при практическом применении.

Обеспечение надежности

Принципы и методы обеспечения надежности

- *Предупреждение ошибок*
 - принципы и методы, позволяющие минимизировать или вообще исключить ошибки
- *Обнаружение ошибок*
 - функции программного обеспечения, помогающие выявлять ошибки
- *Исправление ошибок*
 - функции программного обеспечения, предназначенные для исправления ошибок или их последствий
- *Обеспечение устойчивости к ошибкам*
 - способность ПС продолжать функционирование при наличии ошибок.

Предупреждение ошибок

Лучший способ обеспечить надежность — прежде всего не допустить возникновения ошибок.

Предупреждение ошибок — оптимальный путь к достижению надёжности программного обеспечения.

Методы предупреждения ошибок:

1. Методы, позволяющие справиться со сложностью, свести её к минимуму.
2. Методы достижения большей точности при разработке.
3. Методы улучшения обмена информацией.
4. Методы немедленного обнаружения и устранения ошибок.

Другие три группы методов опираются на предположение, что ошибки все-таки будут

Обнаружение ошибок

- Если предполагать, что в программном обеспечении какие-то ошибки все же будут, то лучшая (после предупреждения ошибок) стратегия — включить **средства обнаружения ошибок** в само программное обеспечение.
- Большинство методов направлено по возможности на незамедлительное обнаружение сбоев.
- Немедленное обнаружение имеет два преимущества: можно минимизировать как влияние ошибки, так и последующие затруднения для человека, которому придется извлекать информацию об этой ошибке, находить ее место и исправлять.

Меры по обнаружению ошибок

- *Взаимное недоверие.*
 - Каждая из компонент должна предполагать, что все другие содержат ошибки. Когда она получает какие-нибудь данные от другой компоненты или из источника вне системы, она должна предполагать, что данные могут быть неправильными, и пытаться найти в них ошибки.
- *Немедленное обнаружение.*
 - Ошибки необходимо обнаружить как можно раньше. Это не только ограничивает наносимый ими ущерб, но и значительно упрощает задачу отладки.
- *Избыточность.*
 - Все средства обнаружения ошибок основаны на некоторой форме избыточности (явной или неявной).

Исправление ошибок

- После того как ошибка обнаружена, либо она сама, либо ее последствия должны быть исправлены программным обеспечением.
- **Исправление ошибок самой системой** — плодотворный метод проектирования надежных систем аппаратного обеспечения. Некоторые устройства способны обнаружить неисправные компоненты и перейти к использованию идентичных запасных.
- Аналогичные методы неприменимы к программному обеспечению вследствие глубоких внутренних различий между сбоями аппаратуры и ошибками в программах.
- Если некоторый программный модуль содержит ошибку, идентичные «запасные» модули также будут содержать ту же ошибку.

Исправление ошибок или предупреждение ошибок ?

- Если методы ликвидации последствий сбоя не могут быть обобщены для работы со многими типами искажений, лучше всего направлять силы и средства на предупреждение ошибок.
- Вместо того чтобы, разрабатывая операционную систему, оснащать ее средствами обнаружения и восстановления цепочки искаженных таблиц или управляющих блоков, следовало бы лучше спроектировать систему так, чтобы только один модуль имел доступ к этой цепочке, а затем настойчиво пытаться убедиться в правильности этого модуля.

Устойчивость к ошибкам

- Методы этой группы ставят своей целью обеспечить функционирование программной системы при наличии в ней ошибок.

Они разбиваются на три подгруппы:

1. динамическая избыточность
2. методы отступления
3. методы изоляции ошибок.

Некоторые примеры

- **Проверка атрибутов любого элемента входных данных.**

- Если входные данные должны быть числовыми или буквенными, проверьте это.
- Если число на входе должно быть положительным, проверьте его значение.
- Если известно, какой должна - быть длина входных данных, проверьте ее.

- **Проверка соответствия входных значений установленным пределам.**

- Если входной элемент — адрес в основной памяти, проверяйте его допустимость. Всегда проверяйте поле адреса или указателя на ноль и считайте, что оно неверно, если равно нулю. Если входные данные — таблица вероятностей, проверьте, находятся ли все значения между нулем и единицей.

- **Проверка допустимости всех вариантов значений.**

- Если входное поле — код, обозначающий один из десяти районов, никогда не предполагайте, что если это не код ни одного из районов 1, 2, ... 9, то это обязательно код района 10.

Обеспечение устойчивости

Самое худшее, что может сделать модуль, — это принять неправильные входные данные и затем вернуть неверный, но правдоподобный результат.

Защитное программирование – проверка его входных и выходных данных на их корректность в соответствии со спецификацией этого модуля, и создание обработчиков соответствующих исключительных ситуаций.

ЗП приводит к снижению эффективности ПС как по времени, так и по памяти.

Обработка ошибок

- Функция сообщает об ошибке возвращением установленного значения. Каждый раз при вызове этих функций происходит проверка возвращаемых значений.
- ```
if(somefunc() == ERROR_RETURN_VALUE)
 //обработка ошибки или вызов обработчика ошибок
else
 //нормальная работа
```
- ```
if( anotherfunc() == NULL )  
    //обработка ошибки или вызов обработчика ошибок  
else  
    //нормальная работа
```
- ```
if(thirdfunc() == 0)
 //обработка ошибки или вызов обработчика ошибок
else
 //нормальная работа
```

## Проблемы:

- Каждый вызов функции должен проверяться программой — увеличение объёма кода
- При использовании классов ошибки могут возникать и при неявном вызове функции, т.е. при работе конструкторов

# Исключения

## **ИСКЛЮЧЕНИЯ**

— возникновение непредвиденных ошибочных условий

## **ОБРАБОТКА ИСКЛЮЧЕНИЙ**

— стандартное языковое средство для реакции на аномальное поведение программы во время выполнения.

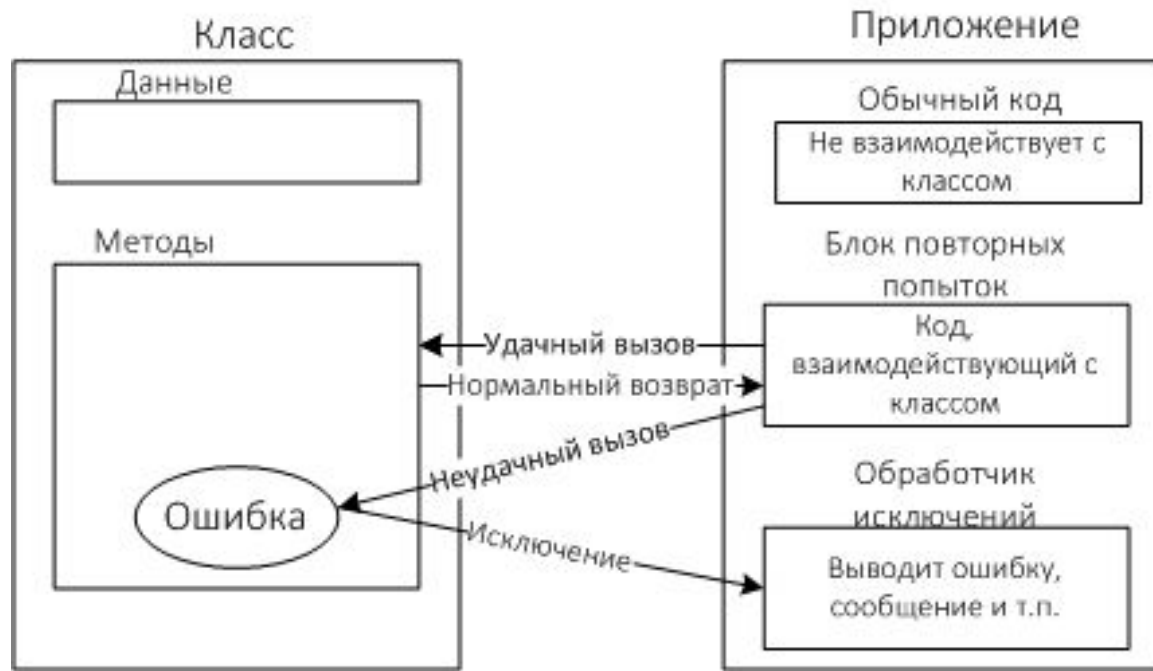
- Например, деление на ноль при операциях с плавающей точкой. Обычно эти условия завершают программу пользователя с системным сообщением об ошибке.
- C++ даёт программисту возможность восстанавливать программу из этих условий и продолжать её выполнение.
- Исключения устанавливаются в блоке *try*, используя выражение *throw*.
- Исключение обрабатывается вызовом соответствующего обработчика *catch*, который выбирается из списка обработчиков, находящегося сразу после их блока *try*.

# Синтаксис исключений

```
try {
 // код, подлежащий контролю
 // функции могут генерировать исключительные ситуации
 // может содержать несколько оператор или целую программу
}
catch (тип1 аргумент) {
 // перехват исключительных ситуаций
}
catch (тип2 аргумент) {
 //
}
...
catch (типN аргумент) {
 //
}
```

- С одним блоком **try** может быть связано несколько операторов **catch**.
- Из нескольких операторов выбирается оператор **catch**, тип аргумента которого совпадает с типом исключительной ситуации.
- Аргумент может быть объектом встроенного типа или класса

# Механизм исключений



- Пусть приложение создает объекты некоторого кода и работает с ними
- Если метод обнаруживает ошибку, то он информирует приложение об этом или *генерирует исключительную ситуацию*
- Отдельная секция кода приложения содержит операции по обработке ошибок – *обработчик исключительных ситуаций*, который отлавливает исключения
- Код приложения, использующий объекты класса, заключается в *блок повторных попыток*

# Обработка исключений

- Установленное выражение — статический, временный объект, который хранится до тех пор, пока не производится выход из ветви обработки особых ситуаций. Выражение захватывается обработчиком, который может использовать его значение.

```
VOID FOO ()
{
 INT I ;
 THROW I ;
}

MAIN ()
 TRY {
 FOO () ;
 }
 CATCH (INT N) { . . . }
```

Значение целого числа, выданное через *throw i*, хранится до завершения работы обработчика с целочисленной сигнатурой *catch (int n)*. Это значение доступно для использования внутри обработчика в виде аргумента.



# Обработка исключений

```
vect::vect(int n)
{
 if (n < 1) throw (n) ;
 p = new int[n];
 if (p == 0) throw ("OUT OF MEMORY");
}

void g(int m)
{
 try { vect a(m); }
 catch (int n)
 { cerr << "SIZE ERROR " << n << endl;}
 catch (const char* error)
 { cerr << error << endl; abort();}
}
```

- При генерации исключительной ситуации управление передаётся оператору **catch**, а выполнение блока **try** прекращается. При этом блок **catch** не вызывается, а просто программа переходит к его выполнению.
- Обычно оператор **catch** пытается исправить ошибку. Если это возможно, то выполнение программы возобновляется с оператора, следующего за блоком **catch**. Однако часто ошибку исправить невозможно, и блок **catch** прекращает выполнение программы, вызывая **exit()** или **abort()**.

# Синтаксис исключений — `throw`

`throw` исключительная ситуация;

- Оператор `throw` генерирует указанную исключительную ситуацию. Если в программе есть ее перехват, оператор `throw` должен выполняться либо внутри блока `try`, либо внутри функции, явно или неявно вызываемой внутри блока `try`.
- Исключительная ситуация может иметь любой тип, в том числе быть объектом класса, определённого пользователем
- Если генерируется исключительная ситуация, для которой не предусмотрена обработка, программа может прекратить свое выполнение. В этом случае вызывается стандартная функция `terminate()`, которая по умолчанию вызывает функцию `abort()`.

# Синтаксис исключений — `try`

```
try { // код, подлежащий
 контролю} ;
```

- Исключение может генерироваться вне блока `try` только в том случае, если оно генерируется функцией, которая содержит оператор `catch` и вызывается внутри блока `try`
- Блок `try` может находиться внутри функции. В этом случае при каждом входе в функцию обработка исключительной ситуации выполняется заново.

# Синтаксис исключений — `catch`

- Код блока `catch` выполняется только при перехвате исключительной ситуации
- Если исключительные ситуации описываются с помощью базового и производных классов, оператор `catch`, соответствующий базовому классу, одновременно соответствует всем производным классам
- Если необходимо обрабатывать не отдельные типы исключительных ситуаций, а перехватывать все подряд, то применяется следующий вид оператора `catch`:

`catch` (...)

```
{
// обработка всех исключительных ситуаций
}
```

# Демонстрация исключения `bad_alloc`

```
int main()
{
 const unsigned long SIZE = 10000; //объем памяти
 char* ptr; //указатель на адрес в памяти
 try
 {
 ptr = new char[SIZE]; //разместить в памяти SIZE байт
 }
 catch (bad_alloc) //обработчик исключений
 {
 cout << "\n Исключение bad_alloc:";
 cout << "\n невозможно разместить данные в памяти.\n";
 return(1);
 }
 delete[] ptr; //освободить память
 cout << "\n Память используется без сбоев. \n";
 return 0;
}
```

# Умный указатель

Будем использовать объект, хранящий указатель и освобождающий объект в своём деструкторе.

```
template<class T>
class auto_ptr {
public:
 auto_ptr(T *p = 0) {} ;
 ~auto_ptr() { delete ptr; }
private:
 T *ptr;
};
```

**Домашнее задание:** для класса «Умный указатель» добавить в определение класса обработку исключений.

# Интеллектуальные указатели в стандартной библиотеке

- **`std::unique_ptr`**

- если вы не собираетесь хранить несколько ссылок на один и тот же объект. Например, используйте его для указателя на память, которая выделяется при входе в некоторую область и освобождается при выходе из неё.
- будет вызван деструктор объекта если указатель будет уничтожен, либо ему будет присвоено новое значение.

- **`std::shared_ptr`**

- если вы хотите сослаться на свой объект из нескольких мест.
- будет вызван деструктор объекта если последний из указателей будет уничтожен, либо последнему указателю будет присвоено новое значение.

- **`std::weak_ptr`**

- когда вы действительно хотите сослаться на свой объект из нескольких мест - для тех ссылок, которые можно игнорировать и освобождать (поэтому они просто заметят, что объект исчез, когда вы попытаетесь разыменовать).