

Семинар 1.8 «Обработка ошибок»

□ Обработка «на месте»:

- if-then-else - для проверки пользовательского ввода;
- abort-exit-terminate - аварийное завершение работы;
- assert - разыменование nullptr, ошибки программиста;
- Передача информации об ошибке: (уведомление) ↗
- коды возврата (C-стиль, 2 способа: return + &);
- механизм исключений (C++-стиль, ООП, классы);
- другие способы (будут рассмотрены в STL);

Стратегию обработки ошибок следует выбирать в начале В крупных проектах эти задачи могут занимать до 60% кода

Коды возврата

+ ПРАВИЛО 80/20

C: см. errno - глобальная переменная?

Алгоритм бинарного поиска: индекс, значение <0 (\Rightarrow ошибка)

Произвольная математическая ф-я: 0.3. $\in \{IR \text{ или } Z\}$ - как? *

* понятие: -1 - хороший результат или код ошибки?

(дополнительно: возможна несовместимость типов)

Решение: передача дополнительного аргумента по ссылке (запись).

Даний подход неудобно поддерживать, расширять и развивать.

Коды возврата не сочетаются с конструкторами и операторами.

Исключения

- этот механизм предназначен не только для обработки ошибок, но вообще для особых (исключительных) ситуаций, но злоупотреблять не стоит, если можно обойтись простым if.

Особенности:

- многоуровневость - (вложенность)
- эффективность
- преобразования

throw - генерация исключения (-1, "error", std::logic_error(...), ...)

try - область перехвата, т.е. зона, где ловятся исключения

catch - обработка исключений заданного типа или всех

- сообщение об ошибке
- действия для отката
- генерация исключения

Пример (live) [try - throw - catch] и многоуровневость исключений

Необработанные исключения - проброс в MS Visual Studio

Пример (live) повторная генерация исключений в catch, - генерация нового исключения или повторно того же самого.

Перехват всех исключений - catch(...) - везде не нужно!

Места установки блоков: *catch(...)*:

- Вокруг ф-ии main
- Вокруг границ модуля
- Вокруг границ потока
- Вокруг стороннего кода

} Крупно-гранулярное расположение

noexcept

- Спецификатор времени компиляции - позволяет генерировать более оптимальный код, см. деструктор или swap.
- Оператор - функция является noexcept согласно условию.

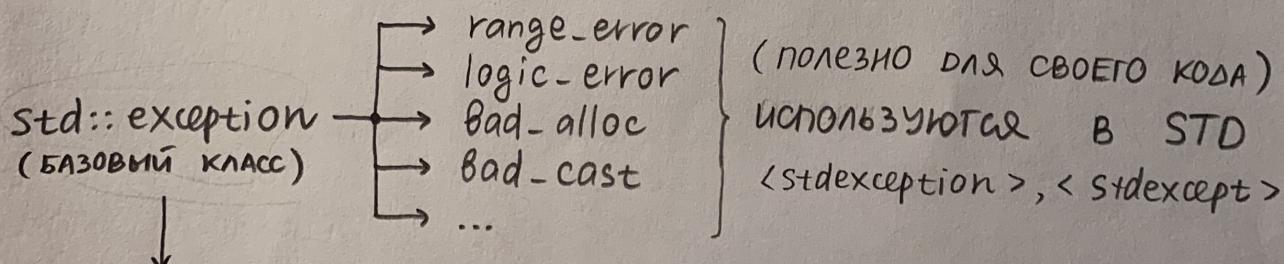
Примечание: исключения не должны генерировать деструктор, т.к. не понятно, что делать дальше - уничтожать объект или прорасыпывать исключение, поэтому деструктор надо делать noexcept.

Примечание: конструктор может генерировать исключения, но деструктор при этом не вызывается + списки инициализации

Классы исключений

Пример (live) простой пользовательский класс, самостоятельный

Есть стандартная иерархия классов:



Также наследуемая от него, получаем в.ф. what() и Base:

Пример (live) пользовательский класс с наследованием exception

Расположение catch-обработчиков: сверху - узкоспециализированное, снизу - общие, catch(...) последний. благодаря ссылкам можем работать с иерархией и в.ф. (сохраняется дин-ий тип).

ГАРАНТИИ БЕЗОПАСНОСТИ ИСКЛЮЧЕНИЙ

- БАЗОВАЯ ГАРАНТИЯ (см. RAII)
 - инварианты не нарушены
 - нет утечек ресурсов
- СТРОГАЯ ГАРАНТИЯ (см. БАИК)
 - ТРАНЗАКЦИОННОЕ ПОВЕДЕНИЕ
- отсутствие исключений (см. swap)
 - исключения не генерируются

} следует писать функции с базовой гарантией минимум

Дополнительно: использование отладчика и профилировщика,
Опционально повторить логирование и работу с дампами.