Изключения

Грешките са неизбежни

- Доколкото изпълнението на програма зависи от външен фактор, то има възможност да се получи грешка
  - некоректен вход от потребител
  - неочакван случай
  - физическа повреда
- Някои грешки могат да бъдат...
  - поправени
  - или поне избегнати
  - или поне засечени
  - или са фатални и просто трябва да се примирим с тях

Обработка на грешки

- Добрият стил на програмиране изисква „разумна“ реакция на програмата при възникване на грешка
  - предложение за поправка, ако е възможно
  - блокиране на грешката и повторен опит, ако е възможно
  - съобщаване при засичане на грешка, ако е възможно
  - „възпитано“ приключване при фатална грешка (graceful exit)
- Крайностите не са желателни
  - „оптимистична“ програма без никаква обработка на грешки
  - „параноична“ програма с прекомерна обработка на грешки
  - златната среда: повече внимание на по-често срещаните грешки

Техники за обработка на грешки

- Проверка за успех
- Код за грешка
- Класове за грешки
- Изключения

Проверка за успех

- Всяка функция/метод се пише така, че върнатият резултат да е истина при успех и лъжа при неуспех
- Пример:
  bool Stack<T>::pop(T& x);
  if (!pop(x)) { cerr << „Грешка при изключване от стек!“; }
- Предимства:
  - моментално приключване на функцията при грешка
  - проста обработка (if)
- Недостатъци:
  - липса на информация за същността на грешката
  - използване на псевдоним за връщане на „истинския“ резултат

Код за грешка

- Всяка функция/метод се пише така, че върнатият резултат да е целочислен код за грешка
- Пример:

```
int work(Task** tasks, int n, int time, int& remaining_time);
switch (work(tasks, n, t, rt)) {
  case 0 : break; // няма грешка
  case 1 : cerr << „Недостатъчно време!“; break;
  case 2 : cerr << „Достигнат е NULL указател!“; break;
  case 3 : cerr << „Задача за отрицателно време!“; break;
  ....
}
```

Код за грешка

- Предимства:
  - позволява специфична обработка на повече от една възможна грешка
- Недостатъци:
  - нужда от познаване на кодовете за грешка
  - отново върнатият резултат на функцията се използва за странична дейност
  - усложнена обработка (switch)
  - невъзможност за детайлна информация за грешката (коя задача?)
  - необходимо е грешката да се обработи веднага (от извикващата функция)

Код за грешка: борба с недостатъците

- нужда от познаване на кодовете за грешка
  - могат да бъдат дефинирани в заглавен файл

    enum work_error { NO_ERROR, NO_TIME, NULL_TASK, NEGATIVE_TIME };

    или

    #define NO_ERROR 0
    #define NO_TIME 1
    #define NULL_TASK 2
    #define NEGATIVE_TIME 3
  - изброеният тип е по-чистият подход

Код за грешка: борба с недостатъците

- върнатият резултат на функцията се използва за странична дейност
  - кодът за грешка може да бъде подаван като параметър-псевдоним
    int work(Task** tasks, int n, int time, work_error& error_code);
    work_error ec;
    cout << „След работа остава „ << work(tasks, n, time, ec);
    switch (ec) { ... }
  - отново изисква допълнителна променлива

Код за грешка: борба с недостатъците

- усложнена обработка (switch)
  - може да се добави параметър за съобщение на грешката

    int work(Task** tasks, int n, int time, work_error& ec, char const*& em);

    work_error ec; char const* em;
    work(tasks, n, time, ec, em);
    if (ec != 0)
      cerr << „Възникна грешка с код „ << ec << „: „ << em << endl;

Код за грешка: борба с недостатъците

- невъзможност за детайлна информация за грешката
  - може да се добави параметър за детайли към грешката

```
int work(Task** tasks, int n, int time, work_error& ec, void*& ed);

work_error ec; void* ed;
work(tasks, n, time, ec, ed);

if (ec == NULL_TASK) {
 cerr << „Задача с номер „ << *(int*)ed << „ е NULL!“ << endl;
}
if (ec == NEGATIVE_TIME) {
  Task* bad_task = (Task*)ed;
  cerr << „Тази задача е отнела отрицателно време: „;
  bad_task->print();
}
```

Код за грешка: борба с недостатъците

- Необходимо е грешката да се обработи веднага
  - Вариант 1: ако извикващата функция не знае как да обработи грешката, на свой ред я връща като резултат
  - Вариант 2: глобална променлива за последната възникнала грешка
  - проблем: броят на възможните грешки нараства много бързо
  - възможно решение: глобален списък с възможни грешки

Класове за грешки

- Когато грешката включва допълнителна информация, може тя да бъде организирана в клас
- ```
  class Error {
    int code;
    char message[100];
  public:
    Error(int,char const*);
    virtual void print() const {
      cout << „Възникна грешка с код: „ << code;
      cout << „ и съобщение: „ << message;
  };
  ```
  ```

Класове за грешки

- class NullTaskError : public Error {
  int index;
  public:
  NullTaskError(int _index) :
  Error(NULL_TASK, „Намерена е NULL задача с индекс „),
  index(_index) {}
  void print() const { Error::print(); cout << index; }
  };
- struct NegativeTimeError : public Error {
  Task* task;
  public:
  void print() const { Error::print(); task->print(); }
  };

Класове за грешки

- int work(Task** tasks, int n, int time, Error*& e) {
  ...
  if (tasks[i] == NULL) e = new NullTaskError(i);
  ...
  if (time_after > time_before) e = new NegativeTimeError(tasks[i]);
- Error* e;
  int remaning = work(tasks, n, time, e);
  if (e != NULL) {
  e->print();
  delete e;
  }

Класове за грешки

- int work(Task** tasks, int n, int time, Error*& e) {
  ...
  if (tasks[i] == NULL) return new NullTaskError(i);
  ...
  if (time_after > time_before) return new NegativeTimeError(tasks[i]);
- Error* e;
  int remaning = work(tasks, n, time, e);
  if (e != NULL) {
  e->print();
  delete e;
  }

Класове за грешки

- Предимства:
  - възможност за връщане на разнообразни грешки
  - специфична информация за всяка грешка
- Недостатъци:
  - все още обработката трябва да е веднага след възникване на грешка
  - все още трябва допълнителен параметър
    - как да върнем грешка при предефиниране на операция?
  - не е ясно какво трябва да направим ако няма какво да върнем!
    - T& Array<T>::get(int i, Error*& e) {
      if (a == NULL) { e = new EmptyArrayError; return ?; }
      if (i < 0) { e = new InvalidIndexError(i); return ?; }
      }

Изключения

- Изключенията са гъвкава възможност за генериране на грешки и тяхната обработка на произволно място в програмата
- При генериране на изключение е позволено да се наруши обичайната последователност на изпълнение на програмата
  - функцията се прекратява веднага
  - не е нужно да се връща резултат
  - изключението може да „прескочи“ няколко извикващи функции

Хвърляне на изключения

- throw [<израз>];
- <израз> може да е от произволен тип (rvalue)
- ако е пропуснато, хвърля текущо обработваното изключение
- Примери:
  - throw 10;
  - throw „Потребителят въведе некоректно име!“;
  - throw student;
  - throw &task;
  - throw new NullTaskError(i);
  - throw NullTaskError(i);

Обработка на изключения

- try <оператор> { catch(<тип> [<име>]) <тяло> }
  [catch(...) <тяло>]
- <оператор> и <тяло> може да са блокове
- ако при изпълнението на <оператор> възникне изключение
- търси се първият подходящ catch(), чийто <тип> е съвместим с типа на хвърленото изключение
- ако <име> е зададено, то се свързва с хвърлената стойност
- изпълнява се <тяло>
- catch(...) и универсален — прихваща всички типове изключения
- ако не бъде намерен подходящ обработчик, изключението се хвърля отново
  - същият ефект се получава, ако в <тяло> има оператор throw; (без параметър)

Примери

- try {
  int remaining = work(tasks, n, t);
  cout << „Изпълнението е успешно, остават “;
  cout << remaining << „ единици време“;
 } catch (NullTaskError const& nte) {
  e.print();
  tasks[nte.i] = new QuickTask(„Колкото да има нещо“);
 } catch (Error const& e) {
  e.print();
 }

Развиване на стека

- При изпълнение на throw започва развиване на програмния стек:
  - преминава се към края на текущо изпълнявания блок
  - унищожават се всички обекти, заделени на стека
    - евентуално се извикват техните деструктори
  - ако краят на блока е край на тяло на функция
    - нейното изпълнение приключва
    - стековата рамка на функцията се освобождава
    - преминава се към извикващата функция
    - ако функцията е main(), програмата спира
  - ако краят на блока е оператор try и има подходящ catch(), се преминава към обработка на изключението
  - иначе процесът се повтаря

Предимства на изключенията

- Разнообразни типове изключения
- Прескачане директно до мястото за обработка
- Автоматично управление на паметта
- Възможност за споделяне на отговорностите по обработка на изключение (обработи и хвърли отново)

Указване на изключения в сигнатурата

- Можем да укажем за всяка функция какви обекти може да хвърля като изключения
- <тип> <име>(<параметри>) [throw({<тип_изключение>})];
- указва, че функцията може да хвърля само изброените типове
- ако списъкът е празен: функцията не хвърля изключения
- ако липсва указване на изключения: може да хвърли всичко
- Примери
  - int work(Tasks**,int,int) throw(NullTimeError,NegativeTimeError);
  - T& Array<T>::operator[](int) throw(RangeException);
  - void print() const throw();

Стандартен клас exception

- Стандартната библиотека на C++ (std) дефинира препоръчителен базов клас за изключения exception
  - има голяма четворка (с виртуален деструктор)
  - има метод virtual const char* what() const, който връща съобщението на грешката
  - включва се чрез #include <exception>
- Дефиниране на собствени изключения:
  - class NullTaskException : public exception {
    const char* what() const { return „Намерена е NULL задача!“; }
    };

Стандартни изключения

- bad_alloc — грешка при заделяне на памет с new
- bad_cast — невалидно преобразуване с dynamic_cast
- bad_exception — неприхванато изключение
- bad_typeid — при опит за typeid на NULL указател към полиморфен клас