Грешке

Врсте гршака

- Грешке у компајлирању
  - Синтаксне грешке
  - Семантичке грешке (у вези са типом)
- Грешке у повезивању
- Грешке у извршавању
  - Открије их рачунар (систем) – „слеши се“
  - Открије их библиотека – „шта си ми ово послао?“
  - Открије их кориснички код – „ово није требало да се деси“
- Логичке грешке
  - Открива их корисник, испитивач (програм се извршава али не даје добар резултат)

Врсте грешака

- Грешке у компајлирању
  - Синтаксне грешке
  - Семантичке грешке (у вези са типом)
- Грешке у повезивању
- Грешке у извршавању
  - Открије их рачунар (систем) – „слеши се“
  - Открије их библиотека – „шта си ми ово послао?“
  - Открије их кориснички код – „ово није требало да се деси“
- Логичке грешке
  - Открива их корисник, испитивач (програм се извршава али не даје добар резултат)

Врсте грашака

- Грешке у компајлирању
  - Синтаксне грешке
  - Семантичке грешке (у вези са типом)
- Грешке у повезивању
- Грешке у извршавању
  - Открије их рачунар (систем) – „слеши се“
  - Открије их библиотека – „шта си ми ово послао?“
  - Открије их кориснички код – „ово није требало да се деси“
- Логичке грешке
  - Открива их корисник, испитивач (програм се извршава али не даје добар резултат)

Врсте грашака

- Грешке у компајлирању
  - Синтаксне грешке
  - Семантичке грешке (у вези са типом)
- Грешке у повезивању
- Грешке у извршавању
  - Открије их рачунар (систем) – „слеши се“
  - Открије их библиотека – „шта си ми ово послао?“
  - Открије их кориснички код – „ово није требало да се деси“
- Логичке грешке
  - Открива их корисник, испитивач (програм се извршава али не даје добар резултат)

Грешке

- Грешке су неизбежне!
- Питање је само: како ћемо се носити са њима?
  - Организовати и писати програм тако да се смањи могућност за прављење грешака и олакша њихово отклањање.
  - Добро тестирати. Дебаговати. Исправити.
  - Трудити се да се спрече озбиљне грешке.
- Процена је да избегавање, проналажење и исправљање грешака чини преко 90% напора у прављењу неког програма.
  - За мале програме ово није толико видљиво! Зато почетници програмери често стичу погрешан осећај.

Ваш програм треба да:

1. да очекиване резултате за све ваљане улазе
2. пријави смислену грешку за лоше улазе
3. не брине о исправности хардвера
4. не брине о исправности системског софтвера (то укључује и системске библиотеке)
5. се заврши у случају детектовања грешке (не мора покушавати да се од ње опорави)

3, 4 и 5 можемо сматрати разумним очекивањима за почетничке програме. Међутим, то није баш тако код озбиљнијих програма.

Наша функција – коју ће неко други користити

```
// Рачуна површину правоугаоника
int area(int x, int y)
{
    return x * y;
}
```

Шта су исправни улази?
Шта је ваљан излаз?

Шта ако улази нису исправни?

```
int x1 = area(7);           // грешка: погрешан број аргумената
int x2 = area("seven", 2); // грешка: погрешан тип
int x3 = area(7, 10);       // ОК
int x5 = area(7.5, 10); // ОК, али опасно: 7.5 се своди на 7
                        // већина компајлера ће дати упозорење
int x = area(10, -7);          // ?

int x = area(50’000, 50’000); // ?
```

Наша функција – коју ће неко други користити

```
int area(int x, int y)
{
    return x * y;
}
```

Шта су исправни улази?
Шта је ваљан излаз?

Шта ако улази нису исправни?
Да ли треба проверавати ваљаност улаза?

Наша функција – коју ће неко други користити

```
int area(int x, int y)
{
    return x * y;
}
```

Једна група приступа: Функција проверава ваљаност улаза.

```
int area(int x, int y)
{
    if (x<=0 || y<=0 || x>46340 || y>46340) ???

    return x * y;
}
```

Наша функција – коју ће неко други користити

```
int area(int x, int y)
{
    return x * y;
}
```

Први приступ из групе:

Функција проверава ваљаност улаза и реагује на грешку.

```
int area(int x, int y)
{
    if (x<=0 || y<=0 || x>46340 || y>46340) {
        cout << "Bad arguments" << endl; //?
        return -1;
    }

    return x * y;
}
```

Шта треба да буде реакција на лоше улазе?

Наша функција – коју ће неко други користити

```
int area(int x, int y)
{
    if (x<=0 || y<=0 || x>46340 || y>46340) return -1;

    return x * y;
}
```

Други приступ из групе: Остављамо позивајућој функцији да реагује на грешку. Повратна вредност – код грешке.

```
int p = area(a, b);
if (p == -1) {
    // some error handling code
}
else {
    // do some useful stuff
}
```

```
int p = area(a, b);
if (p != -1) {
    // do some useful stuff
}
else {
    // some error handling code
}
```

Три проблема: а) функција не мора проверити код грешке
б) који би био код грешке за, рецимо, max(x, y)?
в) код је „загађен“ проверама

Наша функција – коју ће неко други користити

```
int area(int x, int y)
{
    if (x<=0 || y<=0 || x>46340 || y>46340) {
        errno = 7;
        return -1;
    }

    return x * y;
}
```

Трећи приступ из групе: Остављамо позивајућој функцији да реагује на грешку. Глобално видљива променљива – код грешке.

```
int p = area(a, b);
if (errno == 7) {
    // some error handling code
}
else {
    // do some useful stuff
}
```

```
int p = area(a, b);
if (errno == 0) {
    // do some useful stuff
}
else {
    // some error handling code
}
```

Делимично решава проблем б.

Наша функција – коју ће неко други користити

Четврти приступ из групе: Бацање и хватање изузетака.

```
class Bad_area {};

int area(int x, int y)
{
    if (x<=0 || y<=0 || x>46340 || y>46340) throw Bad_area();

    return x * y;
}
```

```
int framed_area(int x, int y)
{
    const int frame_size = 2;
    int p = 0;
    try {
        p = area(x-frame_size, y-frame_size);
    }
    catch (Bad_area) {
        error("Bad area"); // из std_lib_facilities.h
    }
    return p;
}
```

Механизам изузетака

- Могу се пријавити све врсте грешки
- Могу се ухватити на било ком нивоу (и то више пута)
- Склањају код који рукује грешкама од главног тока програма (што доприноси разумљивости)
- Користе га стандардне библиотеке
- Али и даље не даје одговор како треба реаговати на грешку.

Механизам изузетака

Це++	Јава
Објекти **свих типова** могу бити бачени.	Објекти **само одређених** типова (throwable) могу бити бачени
Не специфицира се листа изузетака које одређена функција може да баци. (**Нема** кључне речи **throws**.)*	Специфицира се листа могућих изузетака помоћу кључне речи **throws**.
Нема никаквог **проверавања** да ли се сви могући изузеци хватају.	За већину изузетака се обавља та провера (**checked**), али за неке не (**unchecked**)
Постоји **catch(...)** које хвата све изузетке (осим оних који су изнад експлицитно наведени). **Нема finally** блока.	**Постоји finally** блок.

* Постоји само назнака да функција не баца никакве изузетке: noexcept. Али, то је напредна тема.

Пример у раду са вектором

```
vector<int> v(10);

try {
    cout << v[i] << endl;
}
catch (out_of_range) {
    error("Index out of range");
}
catch (...) {
    error("Some other exception");
}
```

- out_of_range је врста изузетка дефинисана у стандардној библиотеци. Који још изузеци су тамо дефинисани?
- Изузетак ће бити бачен само у Дебаг режиму. (Погледајте опис .at() методе.)
- catch(...) служи да ухвати било који изузетак
- За сада је довољно на сваки изузетак позвати error функцију

Наша функција – коју ће неко други користити

```
int area(int x, int y)
{
    if (x<=0 || y<=0 || x>46340 || y>46340) return -1;

    return x * y;
}
```

Да ли треба проверавати ваљаност улаза?

```
for (int i=1; i<n; ++i) {
    cout << area(i, i+1) << ", ";
}
```

Друга група приступа: Функција не проверава ваљаност улаза.
Можда је довољно само напоменути у коментару или документацији?

```
// Напомена: И x и y морају бити у опсегу (0, 46340]
int area(int x, int y)
{
    return x * y;
}
```

Предуслови (Pre-conditions), тј. очекивања, претпоставке

- Шта функција очекује од аргумената?
  - Често је добро проверити да ли су та очекивања испуњена
  - У најмању руку их треба навести у коментару, тј. документацији

- Шта су очекивања, тј. предуслови, функције framed_area?
- Шта су очекивања оператора за индексирање вектора?
- Кад год желите да употребите неку функцију запитајте се шта су њени предуслови.

- Предуслови се још називају и ограничења (енгл. constraints).
- Ако функција има нека ограничења кажемо да има „узак уговор“.
- Ако нема ограничења, кажемо да има „широк уговор“.

Постуслови (Post-conditions), тј. гаранције

- Шта функција гарантује да ће бити тачно након њеног завршетка?

- Шта су гаранције функције framed_area?
- Шта су гаранције оператора за индексирање вектора?
- Шта су гаранције методе .at()?
- Кад год желите да употребите неку функцију запитајте се шта су њене гаранције.

Кратак осврт на питање „да ли треба увек проверавати предуслове (претпоставке)?“

- Једна врста компромиса је да се неке провере врше само у Дебаг режиму, а да се (када је програм „довољно добро“ испитан) у крајњој употреби (Рилис верзија) уклоне.

- Стандардна библиотека нуди механизам за тако нешто: assert.h

- Тај механизам служи за проверу тврдњи које би увек требало да важе ако су раније компоненте система у току извршавања добро урадиле свој посао.

Наша функција – коју ће неко други користити

Провера претпоставки

```
// Напомена: И x и y морају бити у опсегу (0, 46340]
int area(int x, int y)
{
    assert(x>0 && y>0 && x<=46340 && y<=46340);

    return x * y;
}
```

Откривање претпоставки које нису задовољене

- Најбоље тачке су места позива функција. Проверите пре позива функције, или на почетку позване функције.
  - Ова одлука може да зависи и од тога који од та два кода можете мењати
  - Нпр. **sqrt** прво проверава да ли је улаз ненегативна вредност
- Провера може бити проблематична:
  - Рецимо: **binary_search(a,b,v);** // да ли је v унутар [a:b)?
    - За неке итераторе (нпр. за листу), не можемо проверити да ли је **a<b** – операција < није дефинисана
    - Не можемо проверити да ли су **a** и **b** итератори истог објекта
  - Може бити скупо: Проверавати сваки пут да ли је низ над којима се обавља бинарна претрага уређен је скупље од саме бинарне претраге

Дебаговање

- Како то **не треба** радити (изражено у псеудо коду):

```
while (програм не даје добар резултат)
{
 Разгледати код у потрази за делом који изгледа „чудно“
 Исправити га да изгледа „боље“.
}
```

Смернице за писање кода који је мање подложан грешкама

- Програм треба да буде читљив!
  - Пишите коментаре
    - Објасните идеје у коду
  - Користите смислена имена
  - Пишите уредан код
    - Доследно се придржавајте правила увлачења, именовања и осталих аспеката стандарда кодирања.
    - Развојно окружење је овде од велике помоћи, али није свемоћно.
  - Разделите програм у више мањих функција
    - „Правило десне руке“ за почетак нека буде да функција не би требало да прелази једну страницу/екран.
  - Избегавајте компликоване конструкције
    - Уколико нису баш неопходне
  - Користите стандардну библиотеку

Главне смернице за дебаговање

- Разумети програм
- Репродуковати баг
- Не претпостављати - погледати
- „Подели па владај”

- Покушати решити проблем!!!