Programmeertechnieken
Week 6

Kristian Rietveld

http://liacs.leidenuniv.nl/~rietveldkfd/courses/pt2017/

Universiteit Leiden
The Netherlands

Exception handling

- Murphy's Law:

"Anything that can go wrong, will go wrong."

Exception handling (2)

- Robuste software moet fouten detecteren en wanneer mogelijk kunnen corrigeren.
- Het schrijven van robuste software is helemaal niet zo gemakkelijk als het lijkt ...
- ... er kan meer fout gaan dan je in eerste instantie zou denken.

Exception handling (3)

- Veel "C-stijl" code werkt met error return values. De aanroeper van de functie moet de return value "checken".
  - Expliciete error checking.
- Bijvoorbeeld malloc(), uit de manual pagina:

```
If there is an error, they return a NULL
pointer and set errno to ENOMEM.
```

Exception handling (4)

- Of bijvoorbeeld open(), weer uit de manual pagina:

```
If successful, open() returns a non-
negative integer, termed a file
descriptor.  It returns -1 on failure, and
sets errno to indicate the
error.
```

- Wat staat hier: checken voor een negatieve waarde en zo ja, dan kun je aan de variabele "errno" zien wat er precies aan de hand is.

Exception handling (5)

- Veel "prototype" software doet helemaal (of bijna) geen error checking!
- Van prototype naar robuste "productie" software is zeer veel werk.
  - Vele scenario's testen.
  - Error checking en recovery toevoegen.
  - Enz.

Exception handling (6)

- C-code met error checking gaat er vaak als volgt uitzien:

```
int res = doe_iets();
if (res < 0) {
  /* handel error af */
  return;
}

Object *obj = alloceer_object();
if (obj == NULL) {
  /* handel error af */
  return;
}

res = doe_het_volgende();
if (res < 0) {
  /* handel error af */
  /* vergeet vooral niet "obj"
     vrij te geven. */
}

...
```

Exception handling (7)

```
int lees_element(...)
{
  lees_data(stream, buffer);
  if (buffer geen valide element)
    return -1;

  ...
}

int lees_rij(...)
{
  rij_lengte = lees(stream);
  for (i = 0; i < n_kol; ++i)
    if (lees_element(stream) < 0)
      return -1;
  ...
}

int lees_matrix(...)
{
  lees_dimensies();
  for (i = 0; i < n_rijen; ++i)
    if (lees_rij(stream) < 0)
      return -1;
  ...
}
```

```
int main(...)
{
  if (lees_matrix() < 0)
    std::cerr << "Er ging iets mis ..."
              << std::endl;
}
```

Exception handling (8)

- Vele moderne object-georienteerde programmeertalen hebben een constructie voor "exception handling".
- In plaats van errors te communiceren via "return values" kun je excepties gooien en later weer "vangen".
- Een exceptie kan meer zijn dan alleen een integer. Op deze manier kun je extra informatie over een fout communiceren.

Exception handling (9)

```
static void voorbeeld1(void)
{
  /* Doe iets nuttigs ... */
  /* Oei, het gaat helemaal fout! */
  throw 13;

  /* meer code ... */
}

int main(void)
{
  try {
    voorbeeld1();
  } catch (int fout) {
    std::cerr << "Het ging fout! Fout: "
              << fout << std::endl;
  }

  ...
```

Exception handling (10)

```
#include <exception>

static void voorbeeld2(void)
{
  /* Probeer een bestand te openen. */
  throw std::exception();

  /* Hier gewoon verder als het was gelukt */
}


  try {
    voorbeeld2();
  } catch (std::exception &e) {
    std::cerr << "Fout: " << e.what()
              << std::endl;
  }
```

Exception handling (11)

```
class OnzeException: public std::exception
{
  virtual const char* what() const noexcept
  {
    return "Sorry, we konden het bestand niet openen.";
  }
};

static void voorbeeld3(void)
{
  throw OnzeException();
}


  try {
    voorbeeld3();
  } catch (std::exception &e) {
    std::cerr << "Fout: " << e.what()
              << std::endl;
  }
```

Exception handling (12)

```
static void voorbeeld41(void) {
  // ...
}

static void voorbeeld421(void) {
  throw std::string("Sorry, er ging iets mis ...");
}

static void voorbeeld42(void) {
  voorbeeld421();
}

static void voorbeeld43(void) {
  // ...
}

static void voorbeeld4(void) {
  voorbeeld41();
  voorbeeld42();
  voorbeeld43();
}
```

```
try {
  voorbeeld4();
} catch (std::exception &e) {
  std::cerr << "Fout: " << e.what()
            << std::endl;
} catch (...) {
  std::cerr << "Overige fout"
            << std::endl;
}
```

Exception handling (13)

```
static void voorbeeld5(void)
{
  /* Kan geen vector met negatieve lengte maken */
  std::vector<double> A(-1);

  A[0] = 1234;
}

try {
  voorbeeld5();
} catch (std::exception &e) {
  std::cerr << "Fout: " << e.what()
            << std::endl;
}
```

Exception handling (14)

- Dit waren korte voorbeelden. Normaal wordt code bijvoorbeeld als volgt gestructureerd:

```
void f()  // Using exceptions
{
  try {
    GResult gg = g();
    HResult hh = h();
    IResult ii = i();
    JResult jj = j();
    // ...
  }
  catch (FooError& e) {
    // ...code that handles "foo" errors...
  }
  catch (BarError& e) {
    // ...code that handles "bar" errors...
  }
}
```

- (Voorbeeld van https://isocpp.org/wiki/faq/exceptions#exceptions-separate-good-and-bad-path)

Exception handling (15)

- Wel of geen excepties gebruiken is onderwerp van doorlopend debat ...
- Genoemde problemen zijn:
  - Functies kunnen ineens "return" doen op plekken waar je het niet verwacht. Kan leiden tot bijv. resource leaks.
    - Sommige talen lossen dit op met een “finally” clause, C++ heeft deze functionaliteit niet.
  - Een exceptie die naar buiten "lekt" sluit het programma abrupt af.
  - Performance tegenover simpele return codes.
  - Goede "exception safe" code schrijven is lastig en vereist discipline.

Exception handling (16)

- Voordelen:
  - Door het scheiden van normale code en error handling code, wordt code duidelijker.
  - "Error propagation" in geneste functie-aanroepen wordt al voor ons gedaan.
  - Je kunt excepties niet stilletjes negeren, wat met error checking wel kan.
  - Constructors kunnen geen error code teruggeven.

Exception handling (17)

```
void lees_array(void)
{
  int *A = new int[1000];

  lees_uit_bestand(A); // kan een exceptie gooien

  delete A;
}

int main(void)
{
  try {
    lees_array();
  } catch (...) {
    std::cerr << "Sorry, mislukt!"
              << std::endl;
  }

  return 0;
}
```

Exception handling (18)

- RAII: Resource Acquisition Is Initialization
- Resources moeten worden beheerd door objecten.
  - Constructor alloceert. (Of opent een bestand of connectie, of ...)
  - Destructor dealloceert.
- In geval exception worden de "stack frames" afgelopen tot we een catch vinden ("stack unwind"). Bij het aflopen worden destructors aangeroepen van objecten die de scope verlaten.

Exception handling (19)

- (Onderstaande bedoeld als voorbeeld: geef normaal voorkeur aan bijvoorbeeld STL.)

```
class OnzeArray {
  int *A;
  public:
    OnzeArray() : A(new int[1000]) { };
    ~OnzeArray() { delete[] A; };
};

void lees_array(void)
{
  OnzeArray A;

  /* Als hier een exceptie optreedt, wordt
   * de destructor nog steeds uitgevoerd. */
  lees_uit_bestand(A);

  /* Doe nog iets met A */
}
```

Exception handling (20)

- Item 8: Prevent exceptions from leaving destructors.
- Zorg ervoor dat destructors:
  - niet zelf excepties gooien.
  - als deze een functie aanroepen die tot een exceptie kan leiden, moet deze worden opgevangen. Geef een error en optioneel stop het programma.
- Waarom?
  - Je komt in een situatie dat er twee excepties actief zijn en dat ondersteunt C++ niet.

Exception handling (21)

- Bijvoorbeeld: exceptie tijdens stack unwinding (dus een tweede exceptie).
- Bijvoorbeeld:

```
void functie(void)
{
  std::vector<Object> objecten;
  ...
  /* einde functie: destructoren aangeroepen. */
}
```

- Wat als de eerste destructor een exceptie geeft? De andere objecten moeten ook worden vrijgegeven en daar kan nog een exceptie plaatsvinden.

Exception handling (22)

- Item 29: Strive for exception-safe code.
- Twee eigenschappen "exception-safe functions" wanneer een exceptie optreedt:

  -Leak no resources. Al gezien: RAII.

  -Don't leave objects / data structures in an undefined state.

Exception handling (23)

- Voor dat laatste drie mogelijkheden:
  - **Basic guarantee**: in geval exceptie blijft alles in een valide toestand, maar niet duidelijk wat voor toestand precies.
  - **Strong guarantee:** in geval exceptie blijft de toestand van het programma onveranderd. Dus een functie-aanroep lukt, of er gebeurt niets (vergelijk databasetransactie).
  - **nothrow**: functie geeft garantie nooit een exceptie te veroorzaken.
- nothrow of strong niet altijd praktisch om te implementeren.
- Documenteer altijd het gedrag.

Modern C++

Evolutie van programmeertalen

- C (1972)
- "C with classes" (1979)
- C++ (1983)
- C89
- C++ 2.0 (1989)
- C++98
- C99
- ... (TR1 (2005 - 2007))
- "C++0x" -> C++11
- C++14
- Upcoming: C++17

Evolutie C++

- Na C++98, in de beginjaren 2000 is er eigenlijk weinig aan C++ zelf gebeurd.
- Begin 2000 begon wel het "Boost" project, een open source library waarin vele handige C++ klassen zitten. Datastructuren, algoritmen, threading, image processing, regular expressions, enz., enz.
  - Eigenlijk allerlei zaken die tekortkwamen aan de C++ standard library.

Evolutie C++ (2)

- 2005 - 2007 werd "TR1" gepubliceerd: Technical Report 1.
- Voorstellen voor het uitbreiden van de C++ standard library.
- Werd door sommige compilers (gedeeltelijk) geimplementeerd.
- Voor lange tijd bevatte Boost een implementatie van TR1.

Evolutie C++ (3)

- Begin 2010 werd iedereen weer enthousiast, er gebeurde weer iets!
- C++11
  - Verschillende verbeteringen aan de taal zelf.
  - Het grootste deel van TR1 is in C++11 terechtgekomen.
- Een aantal van de verbeteringen maakt het werken met C++ een stuk eenvoudiger, vandaar dat we deze zullen bespreken.
- We bespreken niet alles, sommige nieuwe elementen zijn dermate complex dat daar meerdere hoorcolleges voor nodig zouden zijn.
- We refereren naar items uit “Effective Modern C++”.

"override" keyword

- Is het volgende valide C++?
- Doet het volgende wat we zouden verwachten?

```
class Figuur
{
  public:
    virtual void zetOpScherm() const {
      std::cout << "Figuur!" << std::endl;
    }
...
};

class Rechthoek : public Figuur
{
  public:
    virtual void zetOpscherm() const {
      std::cout << "Rechthoek!" << std::endl;
    }
};
```

"override" keyword (2)

- Een foutje is snel gemaakt ...
- Met "override" vertel je de compiler dat het je bedoeling is een override van een virtuele functie aan te geven.
- Als dit niet blijkt te gebeuren, compiler error! Handig!
- (Zie ook: Item 12: Declare overriding functions override).

"override" keyword (3)

```
class Rechthoek : public Figuur
{
  public:
    // error!!
    virtual void zetOpscherm() const override {
      std::cout << "Rechthoek!" << std::endl;
    }
};

class Rechthoek : public Figuur
{
  public:
    // ok!
    virtual void zetOpScherm() const override {
      std::cout << "Rechthoek!" << std::endl;
    }
};
```

In-class member initialization

```
class Getal {
  private:
    int ons_getal;
  public:
    Getal() : ons_getal(123) { }
};
```

- In C++11 mag je schrijven:

```
class Getal {
  private:
    int ons_getal = 123;
  public:
    Getal() { }
};
```

nullptr

- In C en C++ zijn 0 en `NULL` eigenlijk integertypen.
- We duiden eigenlijk een pointer aan met adres 0.
- Leer jezelf aan in C++11 `nullptr` te gebruiken ipv 0 of `NULL`.
- Item 8: Prefer `nullptr` to `0` and `NULL`.

nullptr (2)

- Item 8 uit "Effective Modern C++" geeft het volgende voorbeeld:

```
void f(int);
void f(bool);
void f(void *);

f(0);    // Roept f(int) aan.
f(NULL); // Zal in ieder geval niet
         // f(void*) aanroepen.
```

“noexcept” keyword

- We bespraken eerder de "nothrow" guarantee.
- In C++11, markeer functies en methoden die geen exceptie zullen opleveren met `noexcept`:

```
int doeiets(int a, int b, int c) noexcept
{
}
```

- Als er wel een exceptie `doeiets` verlaat, dan grijpt het run-time systeem in: programma wordt afgesloten.

"noexcept" keyword (2)

- Waarom `noexcept`?
  - Het schept duidelijkheid: de functie genereert geen exceptie.
  - Het staat de compiler toe een aantal optimalisaties toe te passen.

Raw-string literals

- Het was in C++ altijd een gedoe om een string te definieren bestaande uit meerdere regels.
- Nu wel mogelijk in C++ met "raw-string literals":

```
std::string my_string(R"STR(asdf jashdf
kjasn asdf
jasdiasd
iasdfn
9snf sdf
asdf)STR");
```

- STR(en STR) worden als begin- en eindmarkering gebruikt en maken geen deel uit van de uiteindelijke string.

Sneller met STL

- We hebben kort geleden kennisgemaakt met STL.
- Een aantal verbeteringen in C++11 maakt het werken met STL een stuk eenvoudiger.

Ter herinnering

```
std::vector<std::pair<std::string, int> > leeftijden;
leeftijden.push_back(std::make_pair("Joop", 53));
leeftijden.push_back(std::make_pair("Karel", 23));
leeftijden.push_back(std::make_pair("Ida", 32));

for (std::vector<std::pair<std::string, int> >::const_iterator
it = leeftijden.begin();
     it != leeftijden.end(); ++it)
  std::cout << it->first << ", " << it->second << std::endl;
```

Initialisatie

- Je mag schrijven

```
int A[] = { 123, 643, 542, 234, 834 };
```

- maar voor `vector`:

```
std::vector<int> integers;

integers.push_back(123);
integers.push_back(643);
integers.push_back(542);
integers.push_back(234);
integers.push_back(834);
```

Initialisatie (2)

- In C++11 ondersteunt de taal en STL het concept van "initializer lists".
- Simpel gezegd, we mogen nu schrijven:

```
std::vector<int> integers{ 123, 643, 542, 234, 834 };
```

Range-based for loops

- Iteratie over STL containers was altijd een hoop typewerk:

```
for (std::vector<int>::iterator it = integers.begin();
     it != integers.end(); ++it)
  std::cout << *it << " ";
std::cout << std::endl;
```

Range-based for loops (2)

- Dat aanroepen van `begin` en `end`, kan dat niet automatisch?
- Ja, C++11 heeft een range-based for loop, die precies dit doet.

```
for (int a : integers)
  std::cout << a << " ";
std::cout << std::endl;
```

- Voor de dubbele punt plaats je de iterator variabele, na de dubbele een container die `begin` en `end` ondersteunt.
- (Aardig om te weten, ook een platte integer array ondersteunt begin en end).

Range-based for loops (3)

- Maar goed, met geneste containers is het nog steeds opletten:

```
std::vector<std::pair<std::string, int> > leeftijden{
    std::make_pair("Joop", 53),
    std::make_pair("Karel", 23),
    std::make_pair("Ida", 32) };

for (std::pair<std::string, int> paar : leeftijden)
  std::cout << paar.first << ", "
            << paar.second << std::endl;
```

"auto" keyword

- C++11 heeft de mogelijkheid gekregen om automatisch het type van de variabele te bepalen aan de hand van een initialisatie.
- Automatic type deduction.

```
auto a = 234;
std::string my_string("hello world");
const auto &ref(my_string);
```

- (Pas wel op: soms vreemde resultaten!)

"auto" keyword (2)

- In ieder geval komt het bij onze nieuwe for-loops goed van pas.

```
for (const auto &paar : leeftijden)
  std::cout << paar.first << ", "
            << paar.second << std::endl;
```

- We gebruiken zelfs een const-reference naar een paar, geen kopieeractie nodig!

Alias declarations

- We leerden al "typedefs" te gebruiken om typewerk te besparen:

```
typedef std::vector<std::pair<std::string, int> > LeeftijdVector;
```

- Typedefs niet altijd makkelijker leesbaar:
  - Hierboven achterstevoren.
  - Functie typedefs lezen vereist oefening.

Alias declarations (2)

- In C++11 kun je "alias declarations" gebruiken:

```
using LeeftijdVector =
std::vector<std::pair<std::string, int> >;
```

- Prettiger leesbaar.
- Technische reden: van typedefs kun je geen template maken, van alias declaraties wel.

```
template<typename T>
using MijnArray = std::array<T, 100>;
```

- (Zie ook: Item 9: Prefer alias declarations to typedefs).

Nieuwe STL containers

- `std::array<T, N>`
  - Vaste grootte in tegenstelling tot `std::vector`.
  - Met methode `at` krijg je een exceptie als je een element buiten de array probeert te benaderen.

Nieuwe STL containers (2)

- `std::unordered_map<Key, T>`
  - `std::map` gebruikt intern een zoekboom, `std::unordered_map` gebruikt een hashtabel.
- `std::unordered_set<T>`
  - Idem.

Nieuwe STL containers (3)

- `std::tuple<>`
  - Zoals `std::pair`, maar kan meer dan 2 items bevatten.

```
std::tuple<int, int, float> t{4, 5, 1.0};
```

Samenvatting

- Wat hebben we tot nu toe gezien?

Samenvatting

- Wat hebben we tot nu toe gezien?
  - Klassen, object, inheritance, virtuele functies.
  - Data encapsulation.
  - Namespaces.
  - Const correctness.

Samenvatting (2)

- (Vervolg)
  - Operator overloading.
  - Templates voor functies en klassen.
  - Standard Template Library.
  - Exceptions.
  - Een aantal C++11 features.

Oefenen, oefenen, oefenen, ...

- Dit zijn een hoop concepten.
- Alleen door deze te gebruiken krijg je de concepten onder de knie.
- Leef je uit in de tweede programmeeropdracht!

Meer leren

- Als je de basis van C++ onder de knie hebt, valt er nog veel meer te leren over C++11:
  - Smart pointers.
  - `constexpr`.
  - Scoped enums.
  - Lambda functies.
  - "Move semantics".
  - `decltype()`
- (En met C++14 kun je er nog een schepje bovenop doen...)

Over twee weken

- “Ons programma is af, maar hoe weten we nu of alles werkt?”
- Hoe krijg je vertrouwen dat je code in orde is?
- Unit testing.