Advanced C++ Programming
Datastructuren 2018

Jonathan K. Vis

j.k.vis@lumc.nl

Vorige week

- Vragen?
- Tentamen, programmeeropgaven en eindcijfer;
- Partiële resultaten van voorgaande jaren;
- Programmeeropgave 1:
  - Deadline: 17 oktober (strict!);
  - Voorbeeld.

Waarom?

In grote lijnen volgen we alle *best practices* uit:

1. Programmeermethoden;
2. Algoritmiek;
3. Programmeertechnieken.

Maar nu met een focus op:

- Datastructuren;
- Programmeren voor andere programmeurs;
- Libraries.

1. Object Oriented Programming
2. Inheritance
3. Polymorphism
4. Standard Template Library
5. Templates
6. `const` Correctness
7. Code Style
8. Comments
9. `make` Build System
10. Testing

Object Oriented Programming

- Organiseren van data en methoden:
  1. interne data representatie: attributen (membervariabelen);
  2. interface (voor interactie):
     - procedures (memberfuncties);
     - defineren het gedrag, maar de implementatie is verborgen.

- Constructors en destructors;
- Access levels: `public`, `private`, `protected`, `static`;
- Getters en setters;
- Operator overloading.

Definitie in .h

```
#ifndef lijst_h
#define lijst_h

int const MAX = 500;

class Lijst
{
    public:
        Lijst(void);
        void voegtoe(int const nieuw);
    private:
        int data[MAX];
        int laatste;
}; // Lijst

#endif
```

Implementatie in .cc/.cpp

```
#include "lijst.h"

Lijst::Lijst(void)
{
    laatste = 0;
} // Lijst::Lijst

void Lijst::voegtoe(int const nieuw)
{
    if (laatste < MAX - 1)
    {
        laatste += 1;
        data[laatste] = nieuw;
    } // if
} // Lijst::voegtoe
```

Meer over headers en `namespace`s

- Libraries hebben soms overlappende elementen (dezelfde naam);
- Denk ook aan programmeeropgave 1;
- Gebruik een `namespace` om conflicten te voorkomen;
- `using namespace std;` — zeker niet in headers;
- Header guards (voor hergebruik);
- `extern` en `static`.

```
namespace karel {
std::string const naam = "Karel";
}

std::cout << "Hello World: " << karel::naam << std::endl;
```

Inheritance of overerving

- Hergebruik van functionaliteit;
- Eenheid en structuur (relaties) tussen klasses;
- Subklasses nemen memberfuncties en -variabelen over van de superklasse (m.u.v. `private`);
- Subklasses kunnen members overschrijven;
- Extra access level: `protected`: niet van buiten, wèl vanuit een subklasse;
- `friend`; geeft toegang tot specifieke `private` members;
- Multiple inheritance.

```
class rechthoek
{
    public:
        rechthoek(int l, int b);
        void setLengte(int l);
        void setBreedte(int b);
        int getLengte();
        int getBreedte();
    protected:
        int lengte, breedte;
}; // rechthoek

class gekleurdeRechthoek : public rechthoek
{
    public:
        gekleurdeRechthoek(int l, int b, int k);
    private:
        int kleur;
}; // gekleurdeRechthoek
```

Polymorphism

```
class Polygon
{
    protected:
        int width, height;
    public:
        void set_values (int a, int b) { width = a; height = b; }
}; // Polygon

class Rectangle: public Polygon
{
  public:
    int area()
      { return width * height; }
}; // Rectangle

class Triangle: public Polygon
{
  public:
    int area()
      { return width * height / 2; }
}; // Triangle
```

Polymorphism

```
int main(int, char* [])
{
    Rectangle rect;
    Triangle trgl;
    Polygon* ppoly1 = &rect;
    Polygon* ppoly2 = &trgl;
    ppoly1->set_values(4, 5);
    ppoly2->set_values(4, 5);
    std::cout << rect.area() << std::endl;
    std::cout << trgl.area() << std::endl;
} // main
```

`virtual`-functies geven nog meer controle.

Standard Template Library

Een verzameling van bibliotheken (`https://en.cppreference.com`) verdeelt in:

- Algoritmes;
- Containers;
- Functies;
- Iterators.

De interne werking is bijna altijd onbelangrijk.

```
std::vector<int> ar = { 1, 2, 3, 4, 5 };
std::vector<int>::iterator it;

for (it = ar.begin(); it != ar.end(); it++)
{
    std::cout << *it << std::endl;
} // for
```

Input en Output

- Gebruik `ostream`/`istream`;
- Superklasse van `cout`, `ofstream`, ...
- Gebruik eventueel operator overloading voor: `<<`.

```
void Lijst::drukaf(std::ostream &stream)
{
    stream << data << std::endl;
} // Lijst::drukaf

std::ostream &
Lijst::operator<<(std::ostream &stream, Lijst const &lijst)
{
    lijst.drukaf(stream)
    return stream;
} // Lijst::operator<<
```

Templates

- Templates worden gebruikt om generieke functies (en klassen) te defineren;
- Één implementatie voor (vele) verschillende data typen;
- Implementatie door de compiler;
- Veelal handig voor rekenkundige operaties voor `int`, `float`, `complex`, `breuk`, ...
- of voor container klassen: `lijst`, `verzameling`, ...
- In heel veel andere gevallen is het nut beperkt.

```cpp
#include <iostream>

template <typename T>
T max(T const &a, T const &b)
{
    return a > b ? a : b;
} // max

int main(int, char* [])
{
    int a = 4, b = 5;
    float c = 4.3, d = 4.4;

    std::cout << max<int>(a, b)   << std::endl
              << max<float>(c, d) << std::endl
              << max<int>(c, d)   << std::endl;
} // main
```

```
template <typename T>
class Lijst
{
    public:
        void voegtoe(T const &nieuw);
    private:
        T data[500];
        int laatste;
}; // Lijst

template <typename T>
void Lijst<T>::voegtoe(T const& nieuw)
{
    data[laatste++] = nieuw;
} // Lijst::voegtoe

int main(int, char* [])
{
    Lijst<int> lijst;
    lijst.voegtoe(4);
} // main
```

Meer Templates

- Template specialization;
- Variadic templates;
- Template aliases (from C++11):

```
template <typename T>
using str_map = std::unordered_map<T, std::string>;
```

- Header file.

Template Metaprogramming

```
template <int N>
struct Factorial
{
    static const int value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0>
{
    static const int value = 1;
};

int main(int, char* [])
{
    std::cout << Factorial<6>::value << std::endl;
} // main
```

`const` Correctness

- Laat de programmeur (en de compiler) weten dat een functie geen aanpassingen doet aan een klasse (bijv. getters);
- Geef aan dat parameters niet veranderen in de functie;
- Vaak is het doorgeven van klasses (of grote data typen) efficienter als call-by-reference, denk ook aan templates;
- `const` werkt van rechts naar links;
- Uitzonderingen? `const_cast<>`.

```
bool Lijst::voegtoe(int const &waarde);
bool Lijst::haaluit(int const &plaats, int &waarde);
bool Lijst::isin(int const &waarde) const;
```

Code Style

- Veel is goed, zo lang het maar consequent is;
- “Elegante” code;
- Leesbaarheid (begrijpbaarheid);
- Inspringen (spaties of tabs) (`https://stackoverflow.blog/2017/06/15/developers-use-spaces-make-money-use-tabs/`);
- Namen: variabelen, klasses, constanten, ...
- Lees eens een “echte” code style;
- Gebruik een redelijke editor, IDE.

Comments

- Altijd commentaar bovenaan iedere file:

```
/**
* klassenaam: beschrijving van klasse/programma
* @author naam (studentnummer)
* @author naam (studentnummer)
* @file filenaam
* @date datum laatste wijziging
**/
```

- En commentaar bij iedere (member)functiedeclaratie (header file):

```
class dinges {
    public:
        // commentaar hier!
        void doeIets();
}; // dinges
// Niet hier!
void dinges::doeIets() { }
```

```
/**
* @function functienaam
* @abstract beschrijving wat de functie doet
* @param parameternaam beschrijving rol parameter
* @return beschrijving van het resultaat
* @pre exacte beschrijving preconditie
* @post exacte beschrijving postconditie
  wat is er veranderd na het uitvoeren van de functie?
**/
```

- Precondities: wat moet er gelden voor een functie kan worden aangeroepen (testbaar!):
  - De datastructuur is niet leeg;
  - Er zijn minimaal n elementen;
  - De operator $<$ is gedefinieerd voor type `T`.
- Postconfities: wat geldt er bij het afsluiten van de functie;
- Bijzondere gevallen, en wat te doen bij errors?

`make` Build System

Code layout:

```
include/
    *.h
src/
    *.cc
tests/
    test_*.cc
LICENSE
Makefile
README.md
```

```
INC_DIR  = include
SRC_DIR  = src
SOURCES  = $(shell find $(SRC_DIR)/ -name '*.cc')
OBJECTS  = $(SOURCES:.cc=.o)
DEPS     = $(OBJECTS:.o=.d)
TARGET   = programma
CC       = g++
CFLAGS   = -march=native
CPPFLAGS = $(addprefix -I, $(INC_DIR)) -Wall -Wextra -pedantic

.PHONY: all clean debug release
release: CFLAGS += -O3 -DNDEBUG
release: all
debug: CFLAGS += -O0 -DDEBUG -ggdb3
debug: all
all: $(TARGET)
clean:
        rm -f $(OBJECTS) $(DEPS) $(TARGET)
$(TARGET): $(OBJECTS)
        $(CC) $(CFLAGS) $(CPPFLAGS) -o $@ $^
-include $(DEPS)
%.o: %.cc
        $(CC) $(CFLAGS) $(CPPFLAGS) -MMD -o $@ -c $<
```

Testing

- Test driven development;
- Memory leaks: `valgrind`;
- Unit tests;
- Use `assert`:

```
#include <cassert>

int main(int, char* [])
{
    assert(2 + 2 == 4);
    assert(2 + 2 == 5);
} // main
```