SYSO1

algoritmen en datastructuren

bd.thrijswijk.nl

THRijswijk

Werkvormen

112 SBU

- 21 uur theorie
- 14 uur practicum
- 77 uur zelfstudie

Inhoud SYSO1

- Algoritmen en datastructuren
- Advanced C++
  - Exceptions
  - Namespaces
  - Casting en RTTI
- STL (standaard C++ library)
- Generiek programmeren

Leermiddelen

- **Boek**
  - Data Structures and Problem Solving Using C++, 2/E, Mark Allen Weiss, ISBN: 0-201-61250-X
- **Dictaat**
  - studiewijzer
  - aanvullingen op theorie
  - extra voorbeelden
  - deel van de practicumhandleiding
- **http://bd.thrijswijk.nl/syso1/**
  - uitgebreide studiewijzer
  - sheets
  - uitgebreide practicumhandleiding
  - sourcecode van alle voorbeelden
  - links

Voorbeeld

Statische datastructuur

```
struct deelnemer {
   int  punten;
   char naam[80];
};
struct stand {
   int aantalDeelnemers;
   deelnemer lijst[100];
};
stand s;
```

De **nadelen** van het gebruik van de ingebouwde datastructuren **struct** en **array** zijn:

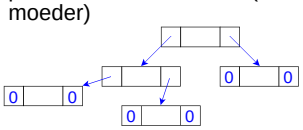
- de grootte van de array's lijst en naam moet bij het vertalen van het programma bekend zijn en kan niet aangepast worden (=**statisch**).
- elke deelnemer neemt evenveel ruimte in onafhankelijk van de lengte van zijn naam (=**statisch**).
- het verwijderen van een deelnemer uit de stand is een heel bewerkelijke operatie.

Lijst

- leeg of
- element met pointer naar **lijst**

Stamboom

- leeg of
- persoon met pointer naar **stamboom** (van vader) en pointer naar **stamboom** (van moeder)



Algorithms

+

Datastructures

=

Programs

Klassieke datastructuren en C++

- Klassieke datastructuren kunnen m.b.v. OOP technieken als **herbruikbare componenten** worden geïmplementeerd.
- Er zijn diverse component-bibliotheken verkrijgbaar:
  - BIDS (Borland International Data Structures)
  - STL (Standard Template Library) **Opgenomen in ISO/ANSI C++ std** (sept 1998) Aanwezig in C++ Builder 6

Datastructuren

- Ding om data “gestructureerd” in op te slaan.
- Een **template** is erg geschikt voor het implementeren van een datastructuur.
- Basisbewerkingen:
  - **insert, remove, find**
  - empty, full, size
- Een bepaalde datastructuur kan op verschillende manieren geïmplementeerd worden.
- Een **Abstracte Base Class** per datastructuur maakt gebruik onafhankelijk van de implementatie.

Datastructures

- Stack
- Queue
- Vector
- Sorted vector
- Linked List
- Sorted list
- Binary Tree
- Search tree
- Hash Table
- Priority Queue (Binary Heap)

Stack

- LIFO (Last In First Out) buffer
  slechts op 1 plaats toegankelijk

- **Memberfuncties:**
  - void push(const T& t) insert O(1)
  - void pop() remove O(1)
  - const T& top() const find O(1)

- **Voorbeeld van gebruik:**
  - expressie evaluator

THRijswijk

ADT Stack

```
#ifndef _THR_Bd_Stack_
#define _THR_Bd_Stack_

template <typename T> class Stack {
public:
   Stack();
   virtual ~Stack();
   virtual void push(const T& t) =0;
   virtual void pop() =0;
   virtual const T& top() const =0;
   virtual bool empty() const =0;
   virtual bool full() const =0;
private:
   // Voorkom toekennen en kopiëren
   void operator=(const Stack&);
   Stack(const Stack&);
};

template <typename T> Stack<T>::Stack() {
}
template <typename T> Stack<T>::~Stack() {
}

#endif
```

THRijswijk

Queue

- FIFO (First In First Out) buffer
  slechts op 2 plaatsen toegankelijk

- **Memberfuncties:**
  - void enqueue(const T& X) insert O(1)
  - void dequeue() remove O(1)
  - const T& front() const find O(1)

- **Voorbeeld van gebruik:**
  - print queue

THRijswijk

Toepassingen van stacks

- Balanced symbol checker
- A simple calculator

Een eenvoudige calculator is een goed te (her)gebruiken component. Denk aan numerieke invoervelden.

THRijswijk

Balance

```
// Contoleer op gebalanceerde haakjes.
// Invoer afsluiten met een punt.
#include <iostream>
#include "stacklist.h" // zie dictaat
using namespace std;

int main() {
   StackWithList<char> s;
   char c;
   cin.get(c);
   while (c!='.') {
      if (c=='('||c=='{'||c=='[')
         s.push(c);
      else
         if (c==')'||c=='}'||c==']')
            if (s.empty())
               cout<<"Fout"<<endl;
            else {
               char d(s.top());
               s.pop();
               if (d=='('&&c!=')'||d=='{'&&c!='}'||d=='['&&c!=']') {
                  cout<<"Fout"<<endl;
               }
            }
      cin.get(c);
   }
   if (!s.empty())
      cout<<"Fout"<<endl;
...
```

THRijswijk

Postfix

Wij zijn gewend *infix* notatie te gebruiken voor het noteren van expressies. Er bestaat nog een andere methode: reverse polish of *postfix* genoemd.
Infix: operand **operator** operand
Postfix: operand operand **operator**

Voordelen postfix t.o.v. infix:
- **Geen prioriteitsregel** nodig (Meneer Van Dalen Wacht Op Antwoord)
- **Geen haakjes** nodig
- **Eenvoudiger te bereken** m.b.v. Stack

We zullen eerst een **postfix calculator** maken en daarna een **infix naar postfix convertor**.

THRijswijk

Postfix

```
// Evalueer postfix expressie.
// Invoer afsluiten met =.
#include <iostream>
#include <cctype>
#include "stacklist.h" // zie dictaat
using namespace std;

int main() {
   StackWithList<int> s;
   char c;
   int i;
   cin>>c;
   while (c!='=') {
      if (isdigit(c)) {
         cin.putback(c);
         cin>>i;
         s.push(i);
      }
      else if (c=='+') {
         int op2(s.top()); s.pop();
         int op1(s.top()); s.pop();
         s.push(op1+op2);
      }
```

... Zie volgende sheet ...

THRijswijk

Postfix

Vervolg ...

```
      else if (c=='*') {
         int op2(s.top()); s.pop();
         int op1(s.top()); s.pop();
         s.push(op1*op2);
      }
      else
         cout<<"Syntax fout"<<endl;
      cin>>c;
   }
   cout<<" = "<<s.top()<<endl;
   s.pop();
   if (!s.empty())
      cout<<"Syntax fout"<<endl;
   cin.get();
   cin.get()
   return 0;
}
```

THRijswijk

Infix => Postfix

gebruik een stack met karakters.

- Lees karakter voor karakter in.
- Als een ingelezen karakter geen haakje of operator is dan kan dit meteen worden doorgestuurd naar de uitvoer.
- Een haakje openenen wordt altijd op de stack geplaatst.
- Als we een operator inlezen dan moeten we net zo lang operatoren van de stack halen en doorsturen naar de uitvoer totdat we:
  - een operator op de stack tegenkomen met een lagere prioriteit of
  - een haakje openen tegenkomen of
  - totdat de stack leeg is.
- Daarna moet de ingelezen operator op de stack worden geplaatst.
- Als we een haakje sluiten inlezen dan moeten we net zo lang operatoren van de stack halen en doorsturen naar de uitvoer totdat we een haakje openen op de stack tegenkomen. Dit haakje openen moet wel van de stack verwijderd worden maar wordt niet doorgestuurd naar de uitvoer.
- Als we einde van de invoer bereiken moeten we alle operatoren van de stack halen en doorsturen naar de uitvoer.

THRijswijk

Stack

Implementations

Een stack kan op verschillende manieren geïmplementeerd worden:

- d.m.v. een **array** (of vector).
- d.m.v. een **gelinkte lijst**.

THRijswijk

Stack

met array

```
#ifndef _THR_Bd_StackWithArray_
#define _THR_Bd_StackWithArray_

#include "stack.h"// zie dictaat

template <typename T> class StackWithArray:
public Stack<T> {
public:
   explicit StackWithArray(int size);
   ~StackWithArray();
   virtual void push(const T& t);
   virtual void pop();
   virtual const T& top() const;
   virtual bool empty() const;
   virtual bool full() const;
private:
   T* a;   // pointer naar de array
   int s;  // size van a
           // (max aantal elementen op de stack)
   int i;  // index in a van de top van de stack
};
```

... Zie volgende sheet ...

THRijswijk

Stack

met array

```
template <typename T>
StackWithArray<T>::StackWithArray(int size):
      a(0), s(size), i(-1) {
   if (s<=0) {
      cerr<<"Stack size should be >0"<<endl;
      s=0;
   }
   else
      a=new T[s];
}
template <typename T>
StackWithArray<T>::~StackWithArray() {
   delete[] a;
}
template <typename T> void
StackWithArray<T>::push(const T& t) {
   if (full())
      cerr<<"Can't push on an full stack"<<endl;
   else
      a[++i]=t;
}
```

... Zie volgende sheet ...

THRijswijk

Stack

met array

```
template <typename T> void
StackWithArray<T>::pop() {
   if (empty())
      cerr<<"Can't pop an empty stack"<<endl;
   else
      --i;
}
template <typename T> const T&
StackWithArray<T>::top() const {
   if (empty()) {
      cerr<<"Can't top an empty stack"<<endl;
      exit(-1);  // no valid return possible
   }
   return a[i];
}
template <typename T> bool
StackWithArray<T>::empty() const {
   return i==-1;
}
template <typename T> bool
StackWithArray<T>::full() const {
   return i==s-1;
}

#endif
```

THRijswijk

Stacktest

```
#include <iostream>
#include "stackarray.h"
using namespace std;

int main() {
   StackWithArray<char> s(32);
   char c;
   cout<<"Type een tekst,sluit af met ."<<endl;
   cin.get(c);
   while (c!='.') {
      s.push(c);
      cin.get(c);
   }
   while (!s.empty()) {
      cout<<s.top();
      s.pop();
   }
   cin.get();
   cin.get();
   return 0;
}
```

THRijswijk

Stack

met lijst

```
#ifndef _THR_Bd_StackWithList_
#define _THR_Bd_StackWithList_

#include "stack.h"

template <typename T> class StackWithList:
public Stack<T> {
public:
   StackWithList();
   virtual ~StackWithList();
   virtual void push(const T& t);
   virtual void pop();
   virtual const T& top() const;
   virtual bool empty() const;
   virtual bool full() const;
private:
   class Node {
   public:
      Node(const T& t, Node* n);
      T data;
      Node* next;
   };
   Node* p; // pointer naar de top van de stack
};
```

THRijswijk

... Zie volgende sheet ...

Stack

met lijst

```
template <typename T>
StackWithList<T>::StackWithList(): p(0) {
}

template <typename T>
StackWithList<T>::~StackWithList() {
   while (!empty())
      pop();
}

template <typename T> void
StackWithList<T>::push(const T& t) {
   p=new Node(t, p);
}

template <typename T> void
StackWithList<T>::pop() {
   if (empty())
      cerr<<"Can't pop an empty stack"<<endl;
   else {
      Node* old(p);
      p=p->next;
      delete old;
   }
}
```

... Zie volgende sheet ...

THRijswijk

Stack

met lijst

```
template <typename T> const T&
StackWithList<T>::top() const {
   if (empty()) {
      cerr<<"Can't top an empty stack"<<endl;
      exit(-1);  // no valid return possible
   }
   return p->data;
}

template <typename T> bool
StackWithList<T>::empty() const {
   return p==0;
}

template <typename T> bool
StackWithList<T>::full() const {
   return false;
}

template <typename T>
StackWithList<T>::Node::Node(const T& t,
            Node* n):
      data(t), next(n) {
}
#endif
```

THRijswijk

Stack

Array versus gelinkte lijst

- Array is **sneller**.
- Array is **statisch**. Niet gebruikte gedeelte is overhead.
- Lijst is **dynamisch**. Heeft overhead van 1 pointer per element

THRijswijk

Dynamisch stack kiezen

```
   Stack<char>* s(0);
   cout<<"Welke stack? (l = list, a = array): ";
   char c;
   do {
      cin.get(c);
      if (c=='l' || c=='L')
         s=new StackWithList<char>;
      else if (c=='a' || c=='A')
         cout<<"Hoeveel elementen?: ";
         int i;
         cin>>i;
         s=new StackWithArray<char>(i);
   } while (c!='l' && c!='L' && c!='a' && c!='A');
   cout<<"Type een tekst, sluit af met ."<<endl;
   cin.get(c);
   while (c!='.') {
      s->push(c);
      cin.get(c);
   }
// ...
   delete s;
```

THRijswijk

28

Advanced C++.

- **vector.** Zie boek 1.2.
- **static class members.** Zie boek 2.4.2.
- **enum class members.** Zie boek 2.4.3.
- **default template parameters.** Zie boek 3.6.2.
- **overerving en overloading.** Zie boek 4.4.1.
- **covariant return types for overridden methods.** Zie boek 4.4.4
- **stringstream.** Zie boek A.4.3
- **namespace.** Zie boek A.5 dictaat 5.1
- **exceptions.** Zie boek 2.5 dictaat 5.2
- **casting en RTTI.** Zie dictaat 5.3

THRijswijk

29

vector<...>

vervangt C build in array ...[]

- een vector is in tegenstelling tot een build in array een "echt" object.
- ondersteund operator[]
- je kunt een vector "gewoon" vergelijken en toekennen.
- een vector kan groeien en krimpen.
- een vector heeft memberfuncties:
  - size()
  - resize(...)
  - at(...)
  - capacity()
  - push_back(...)
  - ...

THRijswijk

30

vector<...>

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
   vector<int> v(10);
   for (vector<int>::size_type i(0); i<v.size(); ++i) {
      v[i]=i*i;
   }
   vector<int> w(v);
   for (vector<int>::size_type i(0); i<w.size(); ++i) {
      cout<<w[i]<<" ";
   }
   cout<<endl;
   vector<int> x;
   x=w;
   for (vector<int>::size_type i(0); i<x.size(); ++i) {
      cout<<x[i]<<" ";
   }
   cout<<endl;
   if (x!=w)
      cout<<"DIT KAN NIET!"<<endl;
   // v[100]=12; ==> crash (als je geluk hebt!)
   // v.at(100)=12; ==> foutmelding
   // eventueel af te vangen zie exceptions
```

THRijswijk

31

vector resize

```
#include <iostream>
#include <vector>
using namespace std;
// vergelijk met figuur 1.3 uit boek van Weiss.

void getInts(vector<int>& vec) {
   vector<int>::size_type itemsRead(0);
   int inputValue;
   cout<<"Enter any number of integers"<<endl;
   while (cin>>inputValue) {
      if (itemsRead==vec.size())
         vec.resize(vec.size()*2+1);
      vec[itemsRead++]=inputValue;
   }
   vec.resize(itemsRead);
}

int main() {
   vector<int> v;
   getInts(v);
   for (vector<int>::size_type i(0); i<v.size(); ++i) {
      cout<<v[i]<<" ";
   }
   cout<<endl;
```

THRijswijk

32

vector push_back

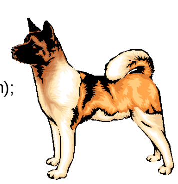
```
#include <iostream>
#include <vector>
using namespace std;
// vergelijk met figuur 1.3 uit boek van Weiss.

void getInts(vector<int>& vec) {
   vec.resize(0);
   int inputValue;
   cout<<"Enter any number of integers:"<<endl;
   while (cin>>inputValue) {
      vec.push_back(inputValue);
   }
}

int main() {
   vector<int> v;
   getInts(v);
   for (vector<int>::size_type i(0); i<v.size(); ++i) {
      cout<<v[i]<<" ";
   }
   cout<<endl;
   cout<<v.size()<<", "<<v.capacity()<<endl;
```

THRijswijk

33

Memberfuncties en datamembers

Elk object heeft zijn eigen datamembers terwijl de memberfuncties door alle objecten van een bepaalde class "gedeeld" worden.



```
class Hond {
public:
   Hond(const string& n);
   void blaf() const;
private:
   string naam;
};

Hond::Hond(const string& n): naam(n) {
}

void Hond::blaf() const {
   cout<<naam<<" zegt: WOEF"<<endl;
}
```

THRijswijk

34

Memberfuncties en datamembers

Stel nu dat we bij willen houden hoeveel objecten van de class Hond er op een bepaald moment bestaan dan zouden we dit als volgt kunnen doen:

```
int aantalHonden=0; //dit is een globale variabele

class Hond {
public:
   Hond(const string& n);
   ~Hond();
   void blaf() const;
private:
   string naam;
};

Hond::Hond(const string n): naam(n) {
   ++aantalHonden;
}
Hond::~Hond() { --aantalHonden; }
```

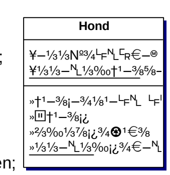
THRijswijk

35

static

Een static datamember is een onderdeel van de class en wordt door alle objecten van de class gedeeld.

```
class Hond {
public:
   Hond(const string& n);
   ~Hond();
   void blaf() const;
   static int aantal();
private:
   string naam;
   static int aantalHonden;
};
```




```
int Hond::aantalHonden=0;

Hond::Hond(const string& n): naam(n) {
   ++aantalHonden;
}
Hond::~Hond() { --aantalHonden; }
int Hond::aantal() { return aantalHonden; }
void Hond::blaf() const {
   cout<<naam<<" zegt: WOEF"<<endl; }
```

THRijswijk

36

Static memberfuncties

- Twee manieren van aanroepen:
  - via een object van de class:
    object_naam.member_functie_naam(parameters)
    Voorbeeld: cout<<h1.aantal()<<endl;
  - direct via de classnaam:
    class_naam::member_functie_naam(parameters)
    Voorbeeld: cout<<Hond::aantal()<<endl;
- Beperkingen t.o.v. een gewone memberfunctie:
  - Een static memberfunctie heeft **geen** receiver (ook niet als hij via een object aangeroepen wordt).
  - Een static memberfunctie heeft dus **geen** this pointer.
  - Een static memberfunctie kan dus **geen** "gewone" memberfuncties aanroepen en ook **geen** "gewone" datamembers gebruiken.

THRijswijk

anonymous enum

definiëren van constanten

```
class Color {
public:
   void setValue(int c);
   static const int BLACK = 0x00000000;
   static const int RED = 0x00FF0000;
// ...
};
```

```
class Color {
public:
   void setValue(int c);
   enum {BLACK = 0x00000000,
         RED = 0x00FF0000 /*, ...*/ };
// ...
};

int main() {
   Color col;
   col.setValue(Color::RED);
// ...
```

THRijswijk

Overloading

niet goed met overerving

```
class Hond {
public:
   Hond(const string& n);
   const string& geefNaam() const;
   virtual ~Hond();
private:
   string naam;
};
class SintBernard: public Hond {
public:
   SintBernard(const string& n);
};

ostream& operator<<(ostream& o,
                    const Hond& h) {
   return o<<"Ik ben een Hond en heet: "
           <<h.geefNaam()<<endl;
}
ostream& operator<<(ostream& o,
                    const SintBernard& b) {
   return o<<"Ik ben een Sint Bernard en heet: "
           <<b.geefNaam()<<endl;
}
```

THRijswijk

Overloading

niet goed met overerving

```
Hond fikkie("Fikkie");
SintBernard boris("Boris");

cout<<fikkie;
cout<<boris;

Hond* hp(0);
hp=&fikkie;
cout<<*hp;
hp=&boris;
cout<<*hp;
```

Boris?

Uitvoer:
Ik ben een Hond en heet: Fikkie
Ik ben een Sint Bernard en heet: Boris
Ik ben een Hond en heet: Fikkie
Ik ben een Hond en heet: Boris

De laatste regel is niet "echt" fout maar toch niet goed! ;-)

THRijswijk

Overriding

```
class Hond {
public:
   Hond(const string& n);
   const string& geefNaam() const;
   virtual ~Hond();
   virtual void print(ostream& o) const;
private:
   string naam;
};

class SintBernard: public Hond {
public:
   SintBernard(const string& n);
   virtual void print(ostream& o) const;
};

ostream& operator<<(ostream& o,
                    const Hond& h) {
   h.print(o);
   return o;
}
```

THRijswijk

Overriding

```
Hond fikkie("Fikkie");
SintBernard boris("Boris");

cout<<fikkie;
cout<<boris;

Hond* hp(0);
hp=&fikkie;
cout<<*hp;
hp=&boris;
cout<<*hp;
```

Boris!

Uitvoer:
Ik ben een Hond en heet: Fikkie
Ik ben een Sint Bernard en heet: Boris
Ik ben een Hond en heet: Fikkie
Ik ben een Sint Bernard en heet: Boris

De laatste regel nu wel goed!

THRijswijk

Invariant return type

```
class Hond {
public:
   Hond(const string& n);
   const string& geefNaam() const;
   virtual ~Hond();
   virtual void print(ostream& o) const;
   virtual Hond* kloon();
private:
   string naam;
};
class SintBernard: public Hond {
public:
   SintBernard(const string& n);
   virtual void print(ostream& o) const;
   virtual Hond* kloon();
};

Hond* Hond::kloon() {
   return new Hond(*this);
}
Hond* SintBernard::kloon() {
   return new SintBernard(*this);
}
```

THRijswijk

Invariant return type

```
Hond fikkie("Fikkie");
SintBernard boris("Boris");
cout<<fikkie;
cout<<boris;

Hond* hp(0);
hp=fikkie.kloon();
cout<<*hp;
delete hp;
hp=boris.kloon();
cout<<*hp;
delete hp;

SintBernard* bp(0);
// bp=boris.kloon();
```

Error: Cannot convert Hond* to SintBernard*

Klonen gaat goed!
Een SintBernard* kan echter niet wijzen naar een gekloonde SintBernard (alleen naar een echte). **Dat is RAAR!**

THRijswijk

Covariant return type

```
class Hond {
public:
   Hond(const string& n);
   const string& geefNaam() const;
   virtual ~Hond();
   virtual void print(ostream& o) const;
   virtual Hond* kloon();
private:
   string naam;
};
class SintBernard: public Hond {
public:
   SintBernard(const string& n);
   virtual void print(ostream& o) const;
   virtual SintBernard* kloon();
};

Hond* Hond::kloon() {
   return new Hond(*this);
}
SintBernard* SintBernard::kloon() {
   return new SintBernard(*this);
}
```

THRijswijk

Covariant return type

```
Hond fikkie("Fikkie");
SintBernard boris("Boris");
cout<<fikkie;
cout<<boris;

Hond* hp(0);
hp=fikkie.kloon();
cout<<*hp;
delete hp;
hp=boris.kloon();
cout<<*hp;
delete hp;

SintBernard* bp(0);
bp=boris.kloon();
```

Klonen gaat helemaal goed!

THRijswijk

Namespace.

Bij grote programma's kunnen verschillende classes "per ongeluk" dezelfde **naam** krijgen.

In C++ kun je classes (en functies etc.) groeperen in zogenaamde **namespaces**.

```
namespace Bd {
  void f(int);
  double sin(double x);
}

// andere file zelfde namespace:
namespace Bd {
  class string { /* ... */ };
}

// andere namespace:
namespace Vi {
  class string {/* ... */ };
}
```

THRijswijk

Using namespaces.

3 manieren van gebruik:

```
// gebruik met scope resolution
Bd::string s1("Harry");
Vi::string s2("John");
std::string s3("Standard");

// gebruik met using declaration
using Bd::string;
string s4("Hallo");
string s5("Dag");

// gebruik met using directive
using namespace Bd;
string s6("Hallo");
double d(sin(0,785398163397448));
```

THRijswijk

Exceptions.

Vaak zal in een memberfunctie gecontroleerd worden op "uitzonderlijke" situaties.

```
double gemSnelheid(double afstand,
                   int uur, int min) {
  if (uur==0 && min==0) {
    cerr<<"Kan gemSnelheid niet berekenen!"
        <<endl;
    exit(GEM_SNELHEID_ERROR);
  }
  return afstand*60/(uur*60+min);
}
```

THRijswijk

Exceptions aangeven met assert.

In de C (en ook in de C++) standaard is de functie assert opgenomen. Deze functie doet niets als de, als parameter opgegeven, expressie true oplevert maar breekt het programma met een passende foutmelding af als dit niet zo is.

```
double gemSnelheid(double afstand,
                   int uur, int min) {
  assert(!(uur==0 && min==0));
  return afstand*60/(uur*60+min);
}
```

Het programma wordt nog steeds abrupt afgebroken. Assert is bedoeld om tijdens de ontwikkeling afspraken te controlleren.

THRijswijk

Exceptions aangeven met returnwaarde.

In C (en ook in C++) werd dit traditioneel opgelost door elke functie een returnwaarde te geven die eventuele errormeldingen vanuit de functie terug meldt.

```
bool gemSnelheid(double& res,
                 double afstand, int uur, int min) {
  if (uur==0 && min==0) {
    return false;
  }
  else {
    res=afstand*60/(uur*60+min);
    return true;
  }
}
```

THRijswijk

Exceptions aangeven met returnwaarde.

Het gebruik van een error return waarde heeft de volgende **nadelen**:

- Bij elke aanroep **moet** de returnwaarde getest worden. Iedereen weet dat het moet, maar bijna niemand doet het!
- Op de plaats waar de fout ondekt wordt kan hij meestal niet opgelost worden.
- De "echte" returnwaarde van de functie moet nu via een call by reference parameter worden teruggegeven.

THRijswijk

Exceptions "gooien".

C++ heeft exceptions ingevoerd voor het afhandelen van "uitzonderlijke" fouten.
Een exception is een **object** dat in de functie waar de fout ontstaat "gegooid" kan worden en dat door de aanroepende functie (of door zijn aanroepende functie enz...) "opgevangen" kan worden. Bij aanroepen van **throw** worden stackframes netjes opgeruimd en de juiste destructors aangeroepen.

```
double gemSnelheid(double afstand,
                   int uur, int min) {
  if (uur==0 && min==0)
    throw domain_error("Reistijd is 00:00");
  return afstand*60/(uur*60+min);
}
```

THRijswijk

Exceptions "opvangen".

De aanroepende functie kan exceptions dan als volgt opvangen:

```
try {
  cout<<gemSnelheid(75.6, 0, 40)<<endl;
  cout<<gemSnelheid(75.6, 0, 0)<<endl;
  cout<<"Dit was het!"<<endl;
} catch (domain_error& e) {
  cerr<<e.what()<<endl;
} catch (...) {
  cerr<<"Andere fout!"<<endl;
}
cout<<"The END."<<endl;
```

THRijswijk

Zelf exceptions definiëren.

In plaats van het gebruik van de standaard gedefinieerde exceptions kun je ook zelf exception classes definiëren:

```
class ReistijdError {};

double gemSnelheid(double afstand,
                   int uur, int min) {
  if (uur==0 && min==0)
    throw ReistijdError();
  return afstand*60/(uur*60+min);
}
```

THRijswijk

Exceptions "opvangen".

De aanroepende functie kan exceptions dan als volgt opvangen:

```
try {
  cout<<gemSnelheid(75.6, 0, 40)<<endl;
  cout<<gemSnelheid(75.6, 0, 0)<<endl;
  cout<<"Dit was het!"<<endl;
} catch (ReistijdError& e) {
  cout<<"Ongeldige reistijd!"<<endl;
} catch (...) {
  cout<<"Andere fout!"<<endl;
}
cout<<"The END."<<endl;
```

THRijswijk

Exceptions hiërarchie.

Doordat exceptions objecten zijn kunnen we ze groeperen in classes. Deze classes kunnen we m.b.v. **overerving** volgens een generalisatie/specialisatie structuur indelen.

```
class SnelheidError {};
class ReistijdError: public SnelheidError {};
class AfstandError: public SnelheidError {};
```

Bij een catch kunnen we nu kiezen of we een **specifieke** of een **generieke** exception willen afvangen. De specifieke zijn polymorf met de generieke.

THRijswijk

Exceptions met functionaliteit.

Doordat exceptions objecten zijn kunnen we ze ook **data** en **gedrag** geven.

We kunnen b.v. een **virtual** memberfunctie definiëren die een bij de exception passende foutmelding geeft. Als we deze memberfunctie in specifiekere exceptions **overridden** dan kunnen we een generieke exception vangen en toch d.m.v. **dynamic binding** de juiste foutmelding krijgen!

THRijswijk

Exception details

- Re-throw.
- De volgorde van catch blokken.
  Zie dictaat 5.2.5.1.
- Exceptions in constructors en destructors.
  - throw is de enige "goede" manier om fouten in een constructor te melden!
  - Gebruik **nooit** throw in een destructor!
- Function try-blok.
- Exception specification.
- Exceptions in de C++ standaard.
  Zie boek figuur 4.1.

THRijswijk

Probleem bij Exceptions

```
void updateDatabase(
       DataBase& db,
       const Record& r) {
   db.lock(r);      // zet record "op slot"
   db.write(r);
   db.unlock(r)  // geef record vrij
}

void showAbout() {
   TForm* fp(new AboutForm());
   fp->showModal();
   delete fp;
}
```

Waarom is het een **probleem** als tijdens write of showModal een exception optreed?

THRijswijk

Onhandige oplossing

```
void updateDatabase(
       DataBase& db,
       const Record& r) {
   db.lock(r);         // zet record "op slot"
   try {
      db.write(r);
   }
   catch(...) {        // vang fout op
      db.unlock(r);    // geef record vrij
      throw;           // verder gooien
   }
   db.unlock(r)        // geef record vrij
}
```

THRijswijk

Handige oplossing

```
class Lock {
public:
   Lock(
      DataBase& db, const Record& r);
   ~Lock();
private:
   Database& db_;
   const Record& r_;
};

Lock::Lock(
   DataBase& db, const Record& r):
      db_(db), r_(r) {
   db_.lock(r_);
}

Lock::~Lock() {
   db_.unlock(r_);
}
```

THRijswijk

Handige oplossing

```
void updateDatabase(
       DataBase& db,
       const Record& r) {
   Lock lock(db, r);
   db.write(r);
}
```

Deze methode wordt "**resource acquisition is initialisation**" genoemd.

Als de functie wordt verlaten (normaal of door exception) wordt de lokale variabele lock netjes "opgeruimd". De destructor van Lock zorgt voor de "unlock" aanroep.

THRijswijk

Exception-save code

- Een programma dat ook bij het optreden van exceptions correct blijft werken wordt *exception-save* genoemd.
- Het schrijven van exception save code wordt nog maar sinds kort goed begrepen.(Zie boek: Exceptional C++ van Herb Sutter © 2000).

THRijswijk

64

Vector (Sem H1)

```
Vector& Vector::operator=(
      const Vector& r) {
   if (size!=r.size) {
      delete[] data;
      data=new int[r.size];
      size=r.size;
   }
   for (int i(0);i<size;++i)
      data[i]=r.data[i];
   return *this;
}

Vector::Vector(const Vector& v):
      size(v.size), data(new int[v.size]) {
   *this=v;
}
```

Bedenk wat er gebeurt als new een **exception** gooit?

THRijswijk

65

Vector (Sem H3)

```
Vector::Vector(const Vector& v):
      size(v.size), data(new int[v.size]) {
   for (int i(0); i<size; ++i)
      data[i]=v.data[i];
}

Vector& Vector::operator=(
      const Vector& r) {
   Vector t(r);
   swap(t);
   return *this;
}

// private hulp functie:
void Vector::swap(Vector& v) throw() {
   std::swap(size, v.size);
   std::swap(data, v.data);
}
```

THRijswijk

66

Casting.

In C kun je met een eenvoudige vorm van “casting” typeconversies doen.

```
int i;
i=(int)"Hallo";
```

In C++ zijn 4 nieuwe vormen van “casting” toegevoegd:

static_cast<T>(e)
zoals oude cast.

const_cast<T>(e)
const erbij of eraf.

reinterpred_cast<T>(e)
compiler afhankelijke casts
b.v. char* ==> int.

dynamic_cast<T>(e)
down cast met run time controle.

THRijswijk

67

Casting.

Voorbeeld van gebruik.

```
#include <iostream>
int main() {
   int i1(1);
   int i2(2);
   double d(i1/i2);
   std::cout<<"d = "<<d<<std::endl;
   return 0;
}

// d = 0
```

```
#include <iostream>
int main() {
   int i1(1);
   int i2(2);
   double d(static_cast<double>(i1)/i2);
   std::cout<<"d = "<<d<<std::endl;
   return 0;
}

// d = 0.5
```

THRijswijk

68

Dynamic casting.

```
class Hond { /*...*/ };

class SintBernard: public Hond {
public:
   Whisky geefDrank();
   /*...*/
private:
   Whisky vat;
};

void hulp(Hond& h) {
   try {
      SintBernard& s;
      s=dynamic_cast<SintBernard&>(h);
      drink(s.geefDrank());
   } catch (bad_cast& e) {
   /* Oops geen Whisky */
   }
}
```

THRijswijk

69

Dynamic casting en RTTI.

Om tijdens **run time** te kunnen controleren of een bepaalde down cast mag of niet moet informatie over het type tijdens run time beschikbaar zijn.
RTTI = **R**un **T**ime **T**ype **I**nformation.
In C++ hebben alle classes met één of meer virtual functions RTTI.

Naast dynamic cast kun je ook de RTTI info behorende bij een object opvragen:

```
print_soort(Hond& h) {
   cout<<typeid(h).name()<<endl;
}
```

RTTI kan ook makelijk misbruikt worden b.v. om een hond te laten blaffen!

THRijswijk

70

Design patterns

in de std C++ library

- Functor
  - objecten die je als een functie kunt gebuiken.
- Adapter (Wrapper)
  - aanpassen van de interface van een class.
- Iterator
  - doorlopen van een datastructuur.

THRijswijk

71

findMax

Deze findMax gebruikt de operator< om elementen te vergelijken

```
template <typename T>
const T& findMax(const vector<T>& v) {
   assert(v.size()>0);  // precondition
   vector<T>::size_type maxIndex(0);
   for (vector<T>::size_type i(1); i<v.size(); ++i)
      if (v[maxIndex]<v[i]) maxIndex=i;
   return v[maxIndex];
}
```

Niet elk type heeft een natuurlijke <

```
class Rectangle {
public:
   Rectangle(int l, int w);
   int getLength() const;
   int getWidth() const;
   void print(ostream& o) const;
private:
   int length;
   int width;
};
```

THRijswijk

72

Functiepointer

```
template <typename T>
const T& findMax(const vector<T>& v,
      bool (*f)(const T&, const T&)) {
   assert(v.size()>0);
   vector<T>::size_type maxIndex(0);
   for (vector<T>::size_type i(1); i<v.size(); ++i)
      if (f(v[maxIndex],v[i])) maxIndex=i;
   return v[maxIndex];
}

bool less(const int& i1, const int& i2) {
   return i1<i2;
}
bool lessLength(const Rectangle& r1,
      const Rectangle& r2) {
   return r1.getLength()<r2.getLength();
}

int main() {
   vector<int> vi;
   vector<Rectangle> vr;
   //...
   cout<<findMax(vi, less)<<endl;
   cout<<findMax(vr, lessLength)<<endl;
```

THRijswijk

Functiepointer

```
template <typename T, typename F>
const T& findMax(const vector<T>& v, F f) {
   assert(v.size()>0);
   vector<T>::size_type maxIndex(0);
   for (vector<T>::size_type i(1); i<v.size(); ++i)
      if (f(v[maxIndex],v[i])) maxIndex=i;
   return v[maxIndex];
}

bool less(int i1, int i2) {
   return i1<i2;
}
bool lessLength(const Rectangle& r1,
      const Rectangle& r2) {
   return r1.getLength()<r2.getLength();
}

int main() {
   vector<int> vi;
   vector<Rectangle> vr;
   //...
   cout<<findMax(vi, less)<<endl;
   cout<<findMax(vr, lessLength)<<endl;
```

THRijswijk

Functieobject

In plaats van een functiepointer kun je nu ook een functieobject (**functor**) aan de functie template findMax meegeven. Nut blijkt later!

```
class LessLength {
public:
   bool operator()(const Rectangle& r1,
      const Rectangle& r2) const;
};

bool LessLength::operator()(
      const Rectangle& r1,
      const Rectangle& r2) const {
   return r1.getLength()<r2.getLength();
}

int main() {
   vector<Rectangle> vr;
   //...
   cout<<findMax(vr, LessLength())<<endl;
```

THRijswijk

Functor

Dit voorbeeld laat zien waarom een functor handiger is dan een functiepointer

```
// De functie findPred geeft de index terug van
// het eerste element dat aan de predicate P
// voldoet of anders v.size()
template <typename T, typename P>
vector<T>::size_type findPred(
      const vector<T>& v, P p) {
   vector<T>::size_type i(0);
   while (i<v.size() && !p(v[i]))
      ++i;
   return i;
}

// Als predicate kunnen we een functiepointer
// meegeven:
bool groterDan_10(int i) { return i>10; }
bool groterDan_11(int i) { return i>11; }
bool groterDan_12(int i) { return i>12; }

int main() {
// ...
   cout<<findPred(vi, groterDan_10)<<endl;
```

THRijswijk

Functor

Als predicate kunnen we ook een **functor** meegeven

```
class GroterDan {
public:
   GroterDan(int i): i_(i) {
   }
   bool operator()(int& i) const {
      return i>i_;
   }
private:
   int i_;
};

int main() {
   vector<int> vi;
//  ...
   cout<<findPred(vi, GroterDan(10))<<endl;
   int x;
   cout<<"Geef x: ";
   cin>>x;
   cout<<findPred(vi, GroterDan(x))<<endl;
```

THRijswijk

Functor

Een **template functor** is nog handiger!

```
template <typename T>
class GroterDan {
public:
   GroterDan(const T& t): t_(t) {
   }
   bool operator()(const T& t) const {
      return t>t_;
   }
private:
   const T& t_;
};

int main() {
   vector<int> vi;
   vector<double> d;
//  ...
   cout<<findPred(vi, GroterDan<int>(10))
         <<endl;
   cout<<findPred(d, GroterDan<double>(1.1))
         <<endl;
```

THRijswijk

Probleem bij Exceptions

```
void showAbout() {
   TForm* fp(new AboutForm());
   fp->showModal();
   delete fp;
}
```

Probleem als tijdens showModal een exception optreed!
Onhandige oplossing:

```
void showAbout() {
   TForm* fp(new AboutForm());
   try {
      fp->showModal();
   catch(...) {
      delete fp;
      throw;
   }
   delete fp;
}
```

THRijswijk

Pointer wrapper

```
template <typename T>
class Pointer {
public:
   Pointer(T* p);
   ~Pointer();
   T* operator->() const;
private:
   T* p_;
};

template <typename T>
Pointer<T>::Pointer(T* p): p_(p) {
}

template <typename T>
Pointer::~Pointer() {
   delete p_;
}

template <typename T>
T* Pointer::operator->() const {
   return p_;
}
```

THRijswijk

Pointer wrapper

Gebruik:

```
void showAbout() {
   Pointer<TForm> fp(
      new AboutForm());
   fp->showModal();
}
```

Standaard alternatief:

```
void showAbout() {
   auto_ptr<TForm> fp(
      new AboutForm());
   fp->showModal();
}
```

Zie boek blz. 164

THRijswijk

auto_ptr

- exception save code

```
void f() {
   auto_ptr<Window> ptr(new Window(...));
   prt->message(...);
}
```

- object source

```
auto_prt<Window> makeWindow() {
   return new Object(...));
}
```

- object sink

```
void closeWindow(auto_ptr<Window> wp) {
   wp->cleanup();
}
```

- **niet** als elementen in een datastructuur!

```
vector <auto_ptr<Window> > vw;
// quick way to crash your system!
```