SOPX3

algoritmen en datastructuren modelleren

bd.thrijswijk.nl

THRijswijk

Werkvormen

168 SBU (84 in kw1 en 84 in kw2)

- kwartaal 1
  - 21 uur theorie
  - 7 uur practicum
- kwartaal 2
  - 1 uur projectbegeleiding/groep

Inhoud SOPX3

- algoritmen en datastructuren
  - STL (standaard C++ library)
- inleiding modelleren
  - UML (standaard modelleringstaal)
- diverse toepassingen van datastructuren.
- project modelleren
  - Together (tool)

THRijswijk

Leermiddelen

- **Boeken**
  - Thinking in C++ 2nd Edition, Volume 1 + 2, Bruce Eckel
  - Praktisch UML, 2de editie, Warmer en Kleppe
- **Dictaat**
  - studiewijzer
  - aanvullingen op theorie
  - extra voorbeelden
  - practicumhandleiding
- **http://bd.thrijswijk.nl/sopx3/**
  - uitgebreide studiewijzer
  - sheets
  - uitgebreide practicumhandleiding
  - sourcecode van alle voorbeelden
  - links

THRijswijk

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.

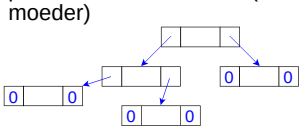
THRijswijk

Lijst

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

Stamboom

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



THRijswijk

Algorithms + Datastructures = Programs

THRijswijk

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

THRijswijk

Modelleren

Werkelijkheid

Modelleren

THRijswijk

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.

THRijswijk

Datastructures

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

THRijswijk

10

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

11

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

12

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

13

Toepassingen van stacks

- Balanced symbol checker
- A simple calculator

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

THRijswijk

14

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

15

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

16

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

17

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

18

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

19

Stack

Implementations

Een stack kan op verschillende manieren geïmplementeerd worden:

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

THRijswijk

20

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

21

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

22

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

23

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

24

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 ...

25

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

26

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

27

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

Queue

Implementations

Een queue kan op verschillende manieren geïmplementeerd worden:

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

THRijswijk

Advanced C++.

- **Namespace.** Zie TICPPV1 H10.
- **Exceptions.** Zie TICPPV2 H1.
- **Casting en RTTI.** Zie TICPPV2 H8.

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:

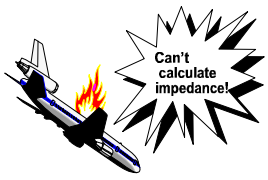
```
// gebruik met scope resolution
Bd::string s1("Harry");
Vi::string s2("John");
std::string s3("Standard");

// 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.

```
complex<double> C::impedance(double f) {
   if (f==0.0 || c==0.0) {
      cerr<<"Can't calculate impedance!";
      exit(C_IMPEDANCE_ERR);
   }
   return complex<double>(0, -1/(2*M_PI*f*c));
}
```



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.

```
complex<double> C::impedance(double f) {
   assert(c!=0.0 && f!=0.0);
   return complex<double>(0, -1/(2*M_PI*f*c));
}
```

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 C::impedance(complex<double>& res,
      double f) {
   if (f!=0.0 && c!=0.0) {
      res=complex<double>(0, -1/(2*M_PI*f*c));
      return true;
   }
   else
      return false;
}
```

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.

```
complex<double> C::impedance(double f) {
  if (c==0.0)
    throw domain_error("Capaciteit == 0");
  if (f==0.0)
    throw domain_error("Frequentie == 0");
  return complex<double>(0, -1/(2*M_PI*f*c));
}
```

THRijswijk

Exceptions "opvangen".

De aanroepende functie kan exceptions dan als volgt opvangen:

```
C c(1e-6);
try {
  cout<<c.impedance(1e3)<<endl;
  cout<<c.impedance(0)<<endl;
  cout<<"Dit was het!"<<endl;
} catch (domain_error& e) {
  cout<<e.what()<<endl;
} catch (...) {
  cout<<"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 FrequencyError {};
class CapacityError {};

complex<double> C::impedance(double f) {
  if (c==0.0)
    throw CapacityError();
  if (f==0.0)
    throw FrequencyError();
  return complex<double>(0, -1/(2*M_PI*f*c));
}
```

THRijswijk

Exceptions "opvangen".

De aanroepende functie kan exceptions dan als volgt opvangen:

```
C c(1e-6);
try {
  cout<<c.impedance(1e3)<<endl;
  cout<<c.impedance(0)<<endl;
  cout<<"Dit was het!"<<endl;
} catch (CapacityError& e) {
  cout<<"Capaciteit == 0"<<endl;
} catch (FrequencyError& e) {
  cout<<"Frequentie == 0"<<endl;
} catch (...) {
  cout<<"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 ImpedanceError {};
class CapacityError:
  public ImpedanceError {};
class FrequencyError:
  public ImpedanceError {};
```

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 TICPPV2 H1.

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

Generieke oplossing

```
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_;
}
```

THRijswijk

Generieke oplossing

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

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

// Standaard alternatief:
void showAbout() {
    auto_ptr<TForm> fp(
        new AboutForm());
    fp->showModal();
}
// Zie TICPPV2 H1
```

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

Vector (SOPX2)

```
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

Vector (SOPX3)

```
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

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

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
```


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 */
   }
}
```

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!

ISO/ANSI Standaard C++ Library

Standaard algoritmen en datastructuren in C++ (voorheen STL)

Zie TICPPV2 hoofdstuk 3 t/m 7.

String in C

nadelen

- statisch (lengte tijdens compileren bepaald).
- null karakter als afsluiting.
- operatoren zijn niet bruikbaar (de str*xxx* functies moeten gebruikt worden).

```
#include <string.h>
#include <iostream.h>
int main() {
   char naam[10];
//  naam="Harry"; // Error!
   strcpy(naam, "Harry"); // OK
   cout<<naam<<endl;
   strcpy(naam, "Willem-Alexander"); // Error!
   cout<<naam<<endl;
//  if (naam=="Kees") { // error
   if (strcmp(naam, "Kees")==0) { // OK
...
```


string in C++

voordelen

- dynamisch (lengte tijdens run-time bepaald).
- operatoren zijn bruikbaar.
- veel extra functionaliteit.

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

int main() {
   string naam;
   naam="Harry";
   cout<<naam<<endl;
   naam="Willem-Alexander";
   cout<<naam<<endl;
   if (naam=="Kees") {
      cout<<"Niet goed!"<<endl;
   }
   return 0;
}
```


string-bibliotheek

- vergelijken met operatoren **> < >= <= == !=**
- optellen en toevoegen met operatoren **+** en **+=**
- **insert** voor invoegen
- **erase** voor verwijderen
- **replace** voor vervangen
- **find** voor zoeken
- **c_str()** voor conversie naar const char*

```
string s;
cout<<"Geef filenaam: ";
cin>>s;
ifstream fin(s.c_str());
getline(fin, s);
```


string

Iterator

Met behulp van een **iterator** kun je door een string lopen op dezelfde manier als dat je met een **pointer** door een char[] kunt lopen.

```
char naam1[ ]="Henk";
for (const char* p(naam1); *p!='\0'; ++p)
   cout<<*p<<" ";
cout<<endl;
cout<<"De eerste letter is:"
      <<naam1[0]<<endl;
cout<<"De laatste letter is: "
      <<naam1[strlen(naam1)-1]<<endl;

string naam2("Harry");
for (string::const_iterator i(naam2.begin());
      i!=naam2.end(); ++i)
   cout<<*i<<" ";
cout<<endl;
cout<<"De eerste letter is:"
      <<naam2[0]<<endl;
cout<<"De laatste letter is: "
      <<naam2[naam2.size()-1]<<endl;
```


STL

- Containers (datastructuren)
- Iterators (om door een container te wandelen)
- Algoritmen (generiek)

Het gebruik van iterators maakt generiek programmeren (**generic programming**) mogelijk. Een generiek algoritme kan op verschillende datatypes (containers) toegepast worden.

Generic programming is dus niet OOP.

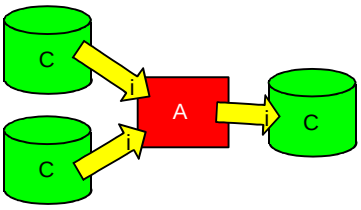
Met behulp van iterators kun je een algoritme op de objecten in een container uitvoeren.

De algoritmen zijn zo efficient mogelijk (geen inheritance en virtuele functies gebruikt).

Je kunt eigen algoritmen, containers en iteratoren toevoegen.

STL

componenten



C = container
A = algoritme
i = iterator

Een algoritme kan ook uit 1 container lezen en in dezelfde container schrijven

THRijswijk

Containers

- Sequentiële containers (linear)
  - []
  - **vector**
  - **list**
  - **deque** (**d**ouble **e**nded **que**ue eigenlijk een devector)
  - bitset (slaan wij over)
  - adapters (in combinatie met vector, list of deque)
    - **stack**
    - **queue**
    - **priority_queue**
- Associatieve containers (elementen op volgorde van sleutelwaarde opgeslagen)
  - **set** (key orgineel)
  - **multiset** (key meerdere malen)
  - **map** (key, value orgineel)
  - **multimap** (key, value meerdere malen)

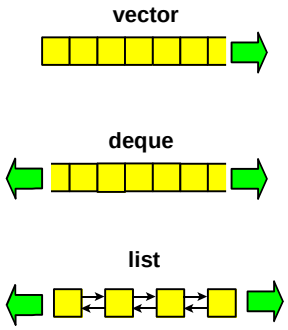
THRijswijk

Containers

- De containers regelen hun eigen **geheugenbeheer**.
- De containers maken **kopietjes** van de elementen die erin gestopt worden. Als je dat niet wilt dan moet je een container met pointers gebruiken.
- Alle objecten in een container zijn van **het zelfde type**. (Dat kan wel een polymorph pointertype zijn!)

THRijswijk

Sequentiële containers



THRijswijk

Sequentiële containers

- **Vector**
  - random access
    - operator[ ]
    - at(...) gooit exception out_of_range
  - toevoegen of verwijderen
    - push_back(...) pop_back() O(1)
    - insert(...) erase(...) O(N)
- **Deque**
  - random access zoals vector
  - toevoegen of verwijderen
    - push_back(...) pop_back() O(1)
    - push_front(...) pop_front() O(1)
    - insert(...) erase(...) O(N)
- **List**
  - **geen** random access
  - toevoegen of verwijderen is O(1)
    - push_back(...) pop_back() O(1)
    - push_front(...) pop_front() O(1)
    - insert(...) erase(...) **O(1)**

THRijswijk

Vector voorbeeld

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

int main() {
   vector<int> v;
   int i;
   cin>>i;
   while (i!=0) {
      v.push_back(i);
      cin>>i;
   }
   print(v); // zelf gedefinieerde functie print
   // Een deel bewerken met iterator.
   if (v.size()>=4)
      for (vector<int>::iterator iter(v.begin()+2);
            iter!=v.begin()+4; ++iter)
         *iter*=2;
   print(v);
   // Een deel bewerken met index.
   if (v.size()>=4)
      for (vector<int>::size_type i(2); i<4; ++i)
         v[i]/=2;
   print(v);
...
```

THRijswijk

Vector voorbeeld

```
// print met behulp van een index:

void print(const vector<int>& vec) {
   cout<<"De inhoud van de vector is:"<<endl;
   for (vector<int>::size_type
         index(0); index!=vec.size(); ++index) {
      cout<<vec[index]<<" ";
   }
   cout<<endl;
}

// print met behulp van een iterator:

void print(const vector<int>& vec) {
   cout<<"De inhoud van de vector is:"<<endl;
   for (vector<int>::const_iterator
         iter(vec.begin()); iter!=vec.end(); ++iter) {
      cout<<*iter<<" ";
   }
   cout<<endl;
}
```

THRijswijk

List voorbeeld

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

class Hond {
public:
   virtual void blaf() const =0;
...
};
class Tekkel: public Hond {
...
}
class SintBernard: public Hond {
...
}

int main() {
   list<Hond*> kennel;
   kennel.push_back(new Tekkel);
   kennel.push_back(new SintBernard);
   kennel.push_back(new Tekkel);
   for (list<Hond*>::const_iterator
         i(kennel.begin(); i!=kennel.end(); ++i)
      (*i)->blaf();
...
```

THRijswijk

Adapters

- **stack**
  - push(...), pop(), top(), empty()
- **queue**
  - kan niet met vector.
  - push(...), pop(), front(), empty()
    Let op: push voegt achteraan toe en pop verwijderd vooraan.
- **priority_queue**
  - kan niet met list.
  - push(...), pop(), top(), empty()
    Let op: top geeft het grootste element terug en pop verwijderd dit element.
  - wordt geïmplementeerd als binary heap.

Deze adapters passen de interface van een vector, deque of list aan.

```
stack<int, vector<int> > s1;
stack<int, deque<int> > s2;
stack<int, list<int> > s3;
stack<int> s4; // using deque by default
```

THRijswijk

Stack voorbeeld

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

int main() {
   stack<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

Associatieve containers

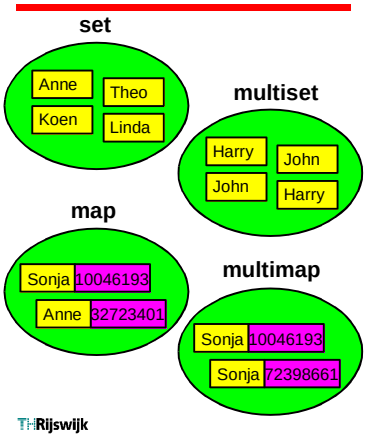
- **set** (verzameling)
- **multiset** (bag)
- **map** (1:N relatie)
- **multimap** (M:N relatie)

Mogelijkheden
- zoeken op key
- doorlopen op volgorde van key

Implementatie
- hash table
- binary search tree (in std gebruikt)

THRijswijk

Associatieve containers



THRijswijk

Set (verzameling)

- Toevoegen met **insert**
  - Een element wordt automatisch op de “goede” plaats ingevoegd.
  - Returntype is een pair<iterator, bool>. De bool geeft aan of het invoegen gelukt is en de iterator geeft de plek aan waar ingevoegd is.
- Verwijderen met **erase**
  - Een te verwijderen element wordt automatisch opgezocht.
- Zoeken met **find**
  - Geeft een iterator terug naar de plaats waar het element staat en geeft end() terug als element niet gevonden is.
- Zoeken met **count**
  - Geeft het aantal keer dat een element voorkomt (0 of 1).

THRijswijk

Set voorbeeld

```
#include <iostream>
#include <string>
#include <set>
using namespace std;

void print(const set<string>& s) {
   cout<<"De set bevat: ";
   for (set<string>::const_iterator i(s.begin());
         i!=s.end(); ++i)
      cout<<*i<<" ";
   cout<<endl;
}

int main() {
   set<string> docenten;
   docenten.insert("John");
   docenten.insert("Paul");
   docenten.insert("Paul");
   docenten.insert("Harry");
   docenten.insert("Bart"); print(docenten);
   pair<set<string>::iterator, bool>
      result(docenten.insert("Harry"));
   if (result.second==false)
      cout<<"1 Harry is genoeg."<<endl;
   docenten.erase("Harry"); print(docenten);
```

THRijswijk

MultiSet (bag)

- Toevoegen met **insert**
  - Een element wordt automatisch op de “goede” plaats ingevoegd.
  - Returntype is een iterator. Deze iterator geeft de plek aan waar ingevoegd is.
- Verwijderen met **erase**
  - Een te verwijderen element wordt automatisch opgezocht. Alle gevonden elementen worden verwijderd.
- Zoeken met **find**
  - Geeft een iterator terug naar de plaats waar het eerste gevonden element staat en geeft end() terug als element niet gevonden is.
- Zoeken met **count**
  - Geeft het aantal keer dat een element voorkomt (>=0).

THRijswijk

Map

1:N relatie

- Elementen zijn:
  pair<const key, value>
- interface gelijk aan set
  - insert
  - erase
  - find
  - count
- extra **operator[ ]**
  - met operator[ ] kun je een key als index gebruiken.
  - Als de key al in de map zit wordt een reference naar de bijbehorende value teruggegeven.
  - Als de key niet aanwezig in de map dan wordt deze key toegevoegd met de default value (default constructor).

THRijswijk

Map voorbeeld

```
#include <iostream>
#include <fstream>
#include <string>
#include <map>
using namespace std;

int main() {
   string w;
   map<string, int> freq;
   cout<<"Geef filenaam: ";
   cin>>w;
   ifstream fin(w.c_str());
   while (fin>>w) {
      ++freq[w];
   }
   for (map<string, int>::const_iterator
         i(freq.begin()); i!=freq.end(); ++i) {
      cout<<i->first<<" "<<i->second<<endl;
   }
   cout<<"Enkele keywords:"<<endl;
   cout<<"if: "<<freq["if"]<<endl;
   cout<<"while: "<<freq["while"]<<endl;
...
```

THRijswijk

Multimap

M:N relatie

- Elementen zijn:
  pair<const key, value>
- interface gelijk aan multiset
  - insert
  - erase
  - find
  - count
- **geen** operator[ ]

THRijswijk

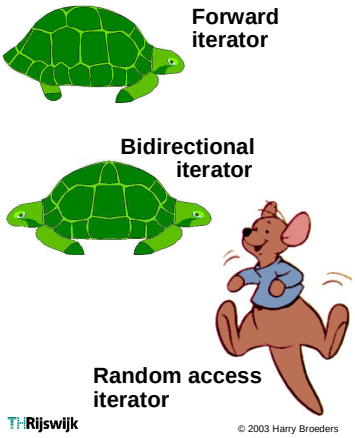
Iterators

Een iterator is geen class maar een (interface) afspraak. Elke class die aan deze afspraak voldoet is als iterator te gebruiken.

- **input** iterator (single pass)
  - een voor een lezen * (rvalue) ++
- **output** iterator (single pass)
  - een voor een schrijven * (lvalue) ++
- **forward** iterator
  - voorwaarts * ++ == !=
- **bidirectional** iterator
  - voor-/achterwaarts * ++ --== !=
- **random access** iterator
  - met sprongen * ++ -- += -= + -== != > >= < <= [ ]
  - een gewone pointer is een random access iterator

THRijswijk

Iteratoren



THRijswijk

Iterators

Relatie met containers

- bidirectional
  - list, set, multiset, map en multimap
- random access
  - vector en deque

Iterators

Relatie met algoritmes

- input bijvoorbeeld find
- output bijvoorbeeld copy
- forward bijvoorbeeld remove
- bidirectional bijvoorbeeld reverse
- random access bijvoorbeeld sort

THRijswijk

Iterators

Speciale iterators

- Insert-iterators (een soort cursor in insert mode).
  - **insert_iterator**
  - **back_insert_iterator**
  - **front_insert_iterator**
- Stream-iterators (koppeling tussen iostream lib en STL).
  - **istream_iterator**
  - **ostream_iterator**

```
vector<int> rij;
ifstream fin("getallen.txt");
istream_iterator<int> iin(fin);
istream_iterator<int> einde;
copy(iin, einde, back_inserter(rij));
sort(rij.begin(), rij.end());
ostream_iterator<int> iout(cout, " ");
copy(rij.begin(), rij.end(), iout);
```

THRijswijk

Algoritmen

Een algoritme werkt op een **sequence**. Een sequence wordt aangegeven door twee iteratoren i1 en i2. De sequence loopt van i1 **tot** (dus **niet** t/m) i2.

- sequence operations
  - non-mutating
  - mutating
- sorteren en aanverwanten
- functie-objecten

```
vector<int> v;
v.push_back(12);
v.push_back(18);
v.push_back(6);
sort(v.begin(), v.end());
for (vector<int>::size_type i(0); i<v.size(); ++i)
   cout<<v[i]<<" ";
```

THRijswijk

Algoritmen

Non-mutating

- **Zoeken** van elementen
  - find
  - find_if
  - find_first_of
- **Tellen** van elementen
  - count
  - count_if
- **Werken** met elementen
  - for_each
- **Zoeken** naar een sequence
  - search
  - find_end
  - adjacent_find
  - search_n
- **Vergelijken** van sequences
  - mismatch
  - equal

THRijswijk

find

```
#include <string>
#include <set>
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
   string s("galgje");
   set<int> v;
   do {
      for (string::size_type i(0); i<s.size(); ++i)
         if (find(v.begin(), v.end(), i)==v.end())
            cout<<'.';
         else
            cout<<s[i];
      cout<<endl<<"Geef een letter: ";
      char c;
      cin>>c;
      string::iterator r(s.begin());
      while ((r=find(r, s.end(), c))!=s.end()) {
         v.insert(r-s.begin());
         ++r;
      }
   }
   while (v.size()!=s.size());
   cout<<"Je hebt het woord gevonden:"<<s;
```

THRijswijk

find_if

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

bool ispos(int i) {
   return i>=0;
}

int main() {
   list<int> l;
   l.push_back(-3);
   l.push_back(-4);
   l.push_back(3);
   l.push_back(4);
   list<int>::iterator r;
   r=find_if(l.begin(), l.end(), ispos);
   if (r!=l.end())
      cout<<"Het eerste positieve element is:"
          <<*r<<endl;
   cin.get();
   return 0;
}
```

THRijswijk

find_if

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

class IsPos {
public:
   bool operator()(int i) const {
      return i>=0;
   }
};

int main() {
   list<int> l;
   l.push_back(-3);
   l.push_back(-4);
   l.push_back(3);
   l.push_back(4);
   list<int>::iterator r;
   r=find_if(l.begin(), l.end(), IsPos());
   if (r!=l.end())
      cout<<"Het eerste positieve element is:"
          <<*r<<endl;
...
```

THRijswijk

find_if

```
#include <iostream>
#include <list>
#include <algorithm>
#include <functional>
using namespace std;

int main() {
   list<int> l;
   l.push_back(-3);
   l.push_back(-4);
   l.push_back(3);
   l.push_back(4);
   list<int>::iterator r;
   r=find_if(l.begin(), l.end(),
             bind2nd(greater_equal<int>(),0));
   if (r!=l.end())
      cout<<"Het eerste positieve element is:"
          <<*r<<endl;
   cin.get();
   return 0;
}
```

THRijswijk

Functie-objecten

Een functie-object is een object met een operator().

- **Predicate** functie-object met een argument van type T dat true of false teruggeeft.
- **Comparisor** functie-object met twee argumenten van type T dat true of false teruggeeft.
  - equal_to<T> not_equal_to<T>
  - greater<T> greater_equal<T>
  - less<T> less_equal<T>
- **Arithmetic and logical**
  - plus<T> return t1+t2
  - ...

Een Comparisor kan worden omgezet in een Predicate met behulp van een **binder**.
bind2nd(greater_equal<int>(), 0)

THRijswijk

for_each

```
#include <vector>
#include <iostream>
#include <iterator>
#include <algorithm>

using namespace std;

void printDubbel(int i) {
   cout<<i<<" "<<i<<" ";
}

int main() {
   vector<int> v;
   v.push_back(-3);
   v.push_back(-4);
   v.push_back(3);
   v.push_back(4);
   ostream_iterator<int> iout(cout, " ");
   copy(v.begin(), v.end(), iout);
   cout<<endl;
   for_each(v.begin(), v.end(), printDubbel);
   cout<<endl;
   cin.get();
   return 0;
}
```

THRijswijk

Algoritmen

mutating

- **Kopiëren**
  - copy copy_if
- **Bewerken**
  - transform
  - replace replace_if
  - replace_copy replace_copy_if
  - rotate rotate_copy
  - random_shuffle
- **Verwisselen**
  - swap_range
  - reverse reverse_copy
- **Verwijderen**
  - remove remove_if
  - remove_copy remove_copy_if
  - unique unique_copy
- **Diversen**
  - fill fill_n
  - generate generate_n

THRijswijk

transform

```
#include <iostream>
#include <vector>
#include <iterator>
#include <functional>
#include <algorithm>
using namespace std;

int telop(int i, int j) { return i+j; }

int main() {
   vector<int> v, w;
   v.push_back(-3); v.push_back(-4);
   v.push_back(3); v.push_back(4);
   w.push_back(1); w.push_back(2);
   w.push_back(3); w.push_back(4);
   ostream_iterator<int> iout(cout, " ");
   copy(v.begin(), v.end(), iout); cout<<endl;
// Bewerking opgeven met een functie.
   transform(v.begin(), v.end(), w.begin(),
                                 v.begin(), telop);
   copy(v.begin(), v.end(), iout); cout<<endl;
// Bewerking opgeven met std functie-objecten.
   transform(v.begin(), v.end(), w.begin(),
                                 v.begin(), plus<int>());
   copy(v.begin(), v.end(), iout); cout<<endl;
   cin.get(); return 0;
}
```

THRijswijk

remove

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <functional>
using namespace std;

int main() {
   vector<int> v;
   for (int i(0); i<10; ++i) {
      v.push_back(i*i);
   }
   ostream_iterator<int> out(cout, " ");
   copy(v.begin(), v.end(), out); cout<<endl;
   vector<int>::iterator end(
      remove_if(v.begin(), v.end(),
            not1(bind2nd(modulus<int>(), 2)))
   );
   copy(v.begin(), end, out); cout<<endl;
   copy(v.begin(), v.end(), out); cout<<endl;
   v.erase(end, v.end());
   copy(v.begin(), v.end(), out); cout<<endl;
// ...
```

THRijswijk

Uitvoer

```
0 1 4 9 16 25 36 49 64 81
1 9 25 49 81
1 9 25 49 81 25 36 49 64 81
1 9 25 49 81
```

THRijswijk

Algoritmen

Sorteren e.d.

- **Sorteren**
  - sort
  - stable_sort
  - binary_search
- **Set operations**
  - includes
  - set_union
  - set_intersection
  - set_difference
  - set_symmetric_difference
- **Diversen**
  - ...

THRijswijk

mem_fun

```
#include <iostream>
#include <list>
#include <algorithm>
#include <functional>
using namespace std;

class Hond {
public:
   virtual void blaf() const =0;
...
};

class Tekkel: public Hond { ... };

class StBernard: public Hond { ... };

int main() {
   list<Hond*> k;
   k.push_back(new Tekkel);
   k.push_back(new StBernard);
   k.push_back(new Tekkel);
   for_each(k.begin(), k.end(),
      mem_fun(&Hond::blaf));
...
```

THRijswijk

Galgje

Met gebruik van sets

```
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
using namespace std;

int main() {
   string w("galgje");
   set<char> geraden, letters;
   copy(w.begin(), w.end(),
      inserter(letters, letters.begin()));
   do {
      for (string::const_iterator i(w.begin());
            i!=w.end(); ++i)
         if (geraden.count(*i)) cout<<*i;
         else cout<<'.';
      cout<<endl<<"Raad een letter: ";
      char c;
      cin>>c;
      geraden.insert(c);
   }
   while (!includes(
      geraden.begin(), geraden.end(),
      letters.begin(), letters.end()));
   cout<<w<<" is geraden!"<<endl;
```

THRijswijk

100

Galgje

Met gebruik van sets en for_each

```
// includes idem

set<char> geraden;

void printLetter(char c) {
   if (geraden.count(c)) cout<<c;
   else cout<<'.';
}

int main() {
   string w("galgje");
   set<char> letters;
   copy(w.begin(), w.end(),
      inserter(letters, letters.begin()));
   do {
      for_each(w.begin(), w.end(), printLetter);
      cout<<endl<<"Raad een letter: ";
      char c;
      cin>>c;
      geraden.insert(c);
   }
   while (!includes(
      geraden.begin(), geraden.end(),
      letters.begin(), letters.end()));
   cout<<w<<" is geraden!"<<endl;
```

THRijswijk

101

Galgje

Met gebruik van strings

```
#include <iostream>
#include <string>

using namespace std;

int main() {
   string w("galgje");
   string geraden(w.length(), '.');
   do {
      cout<<geraden<<endl
         <<"Raad een letter: ";
      char c;
      cin>>c;
      for (string::size_type i(0);
            i<w.length(); ++i)
         if (w[i]==c)
            geraden[i]=c;
   }
   while (geraden!=w);
   cout<<w<<" is geraden!"<<endl;
   cin.get(); cin.get();
   return 0;
}
```

THRijswijk

102

Galgje

Met gebruik van strings en transform

```
// includes idem

class Vulin {
public:
   Vulin(char c): c(c) { }
   char operator()(char c1, char c2) const {
      return c1==c ? c1 : c2;
   }
private:
   const char c;
};

int main() {
   string w("galgje");
   string geraden(w.length(), '.');
   do {
      cout<<geraden<<endl
         <<"Raad een letter: ";
      char c; cin>>c;
      transform(w.begin(), w.end(),
         geraden.begin(), geraden.begin(),
         Vulin(c));
   }
   while (geraden!=w);
   cout<<w<<" is geraden!"<<endl;
```

THRijswijk

103

Galgje

Met gebruik van strings en transform

```
// includes idem
class Raad {
public:
   Raad() {
      cin>>c;
   }
   char operator()(char c1, char c2) const {
      return c1==c ? c1 : c2;
   }
private:
   char c;
};

int main() {
   string w("galgje");
   string geraden(w.length(), '.');
   do {
      cout<<geraden<<endl
         <<"Raad een letter: ";
      transform(w.begin(), w.end(),
         geraden.begin(), geraden.begin(),
         Raad());
   }
   while (geraden!=w);
   cout<<w<<" is geraden!"<<endl;
```

THRijswijk

104