

# Programmeertechnieken

## Week 5, 6

Kristian Rietveld

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



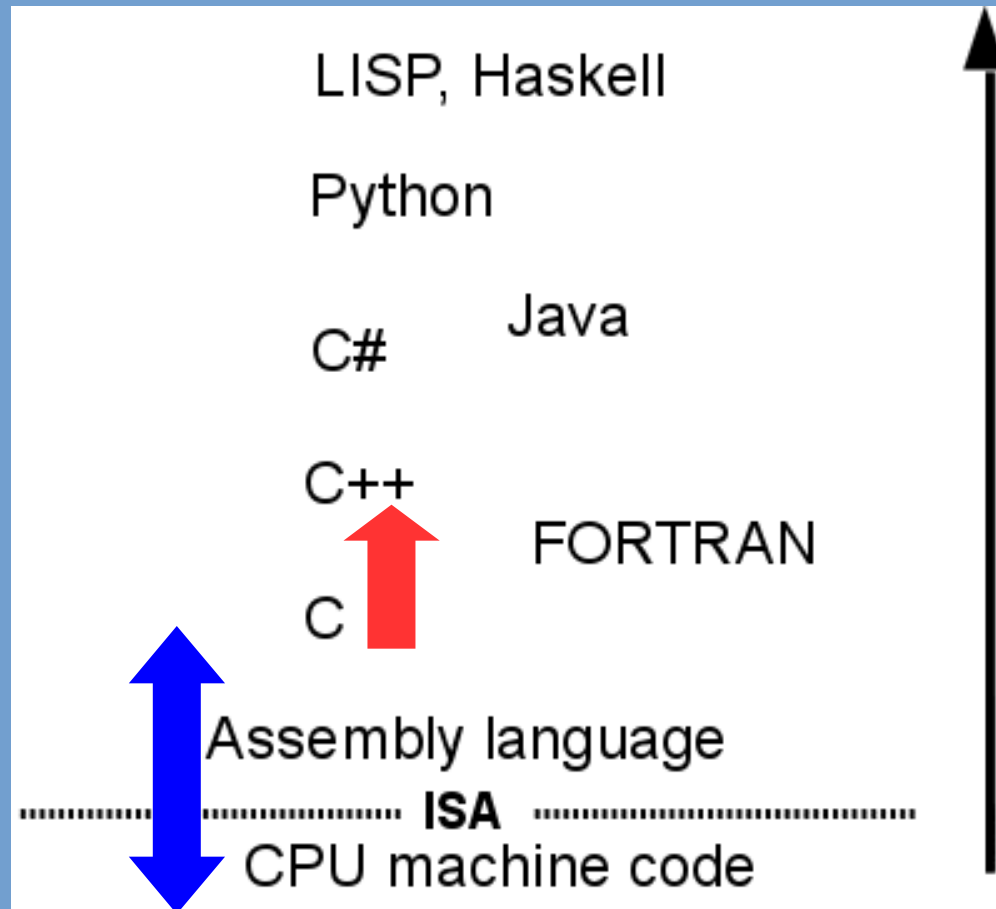
Universiteit Leiden  
The Netherlands

Universiteit Leiden. Bij ons leer je de wereld kennen

# Terug naar C++

- We hebben nu kort kennis gemaakt met de abstractielagen onder C++.
  - Machinecode.
  - Assembly.
  - C en de relatie tussen C en assembly.

# Terug naar C++ (2)



# Terug naar C++ (3)

- Met dat in het achterhoofd gaan we nu abstractielagen toevoegen.
- In het bijzonder: Object Georienteerd Programmeren.
  - Object Oriented Programming (OOP)
- In OOP worden programma's geschreven door objecten te definiëren en objecten met elkaar te laten interacteren.
- Begin traceert terug naar eind jaren 60 - begin 70.

# OOP

- Enige eigenschappen van objecten:
  - Een object is een instantiatie van een klasse.
  - Objecten bevatten data.
  - Objecten hebben "methoden" die kunnen worden aangeroepen.
  - Objecten kunnen weer andere objecten bevatten (compositie).
  - Een klasse kan de eigenschappen van een andere klasse "erven" (inheritance).

# OOP in C

- In C wordt het al snel een hoop gedoe ...

```
struct MyObject { ... };

struct MyObject *myobject_new(void);
void myobject_free(struct MyObject *obj);

int myobject_get_int_field(struct MyObject *obj);
void myobject_set_int_field(struct MyObject *obj,
                           int value);
...
```

# C++

- We willen ons bezighouden met objecten en niet allerlei andere details.
  - Abstractniveau gaat omhoog!
- C++ is ooit ontstaan als "C with Classes".
- De eerste C++ "compilers" genereerden eigenlijk C code.

# C++ (2)

- Het plan: korte herhaling en verdere verdieping in C++.
- Tevens een introductie tot belangrijkste verbeteringen in C++11.
- Daarna maken we de sprong naar Java, en ...
  - ... leren we meer over concepten en ontwerp van OOP software.
  - ... beginnen we met kijken naar "standaard structuren" in OOP code (design patterns).
  - ... leren we werken met (grote) bestaande frameworks om software te schrijven (o.a. Android).
- Nu eerst: een korte herhaling van C++ klassen.



# Klassen

```
class Persoon {  
protected:  
    std::string name;  
    int age;  
  
public:  
    Persoon(const std::string &name,  
            const int age);  
  
    const std::string &getName(void) const;  
    int getAge(void) const;  
};
```

# Klassen (2)

Definitie klasse met naam

```
class Persoon {  
    protected:  
        std::string name;  
        int age;  
  
    public:  
        Persoon(const std::string &name,  
                const int age);  
  
        const std::string &getName(void) const;  
        int getAge(void) const;  
};
```

# Klassen (3)

```
class Persoon {  
protected:
```

```
    std::string name;  
    int age;
```

Declaratie “data members”

```
public:
```

```
    Persoon(const std::string &name,  
            const int age);
```

```
    const std::string &getName(void) const;  
    int getAge(void) const;  
};
```

# Klassen (4)

```
class Persoon {  
protected:  
    std::string name;  
    int age;
```

Declaratie constructor

```
public:
```

```
Persoon(const std::string &name,  
        const int age);
```

```
    const std::string &getName(void) const;  
    int getAge(void) const;  
};
```

# Klassen (5)

```
class Persoon {  
protected:  
    std::string name;  
    int age;  
  
public:  
    Persoon(const std::string &name,  
            const int age);  
  
    const std::string &getName(void) const;  
    int getAge(void) const;  
};
```

Declaratie “member functions” of “methoden”.

# Klassen (6)

```
class Persoon {  
    protected:  
        std::string name;  
        int age;
```

```
    public:
```

```
        Persoon(const std::string &name,  
                const int age);  
  
        const std::string &getName(void) const;  
        int getAge(void) const;  
};
```

“Access specifier”

- **Public**: iedereen mag er bij.
- **Private**: alleen methoden.
- **Protected**: methoden & methoden van subklasse.

# Klassen (7)

```
class Persoon {  
protected:  
    std::string name;    Reference variabele: een  
    int age;             verwijzing naar een object.  
                           C++-only!  
public:  
    Persoon(const std::string &name,  
            const int age);  
  
    const std::string &getName(void) const;  
    int getAge(void) const;  
};
```

# Klassen (8)

```
class Persoon {  
protected:  
    std::string name;  
    int age;  
  
public:  
    Persoon(const std::string &name,  
            const int age);  
  
    const std::string &getName(void) const;  
    int getAge(void) const;  
};
```

Hier komen we nog op terug!



# Instantiatie

```
Persoon p( "Joop Test", 23 );
```

```
Persoon *r = new Persoon( "Karel", 64 );
```

```
...
```

```
delete r;
```

- p is nu een object van type Persoon.
- r is een pointer naar een dergelijk object.

# Inheritance

```
class Student : public Persoon {  
    private:  
        int sid;  
  
    public:  
        Student(const std::string &name,  
                const int age,  
                const int sid);  
};
```

# Inheritance (2)

- Je kunt de constructor van de basisklasse (ouder) aanroepen in de member initialiser list.

```
Student::Student(const std::string &name,  
                 const int age,  
                 const int sid)  
    : Persoon(name, age), sid(sid)  
{ }
```

# Virtuele functies

- Omdat een Student een Persoon is (is-a relatie), mogen we het volgende doen:

```
Persoon *s = new Student("Student 1", 20, 1234512);
```

- Vervolgens kunnen we methoden aanroepen met pointer S.
- Als de klasse Student een methode getSID( ) zou hebben, kunnen we deze zonder cast *niet* aanroepen via S.

# Virtuele functies (2)

- Wat nu als we een functie `drukAf` willen hebben, welke voor een `Student` meer (of andere) informatie afdruckt dan voor een `Persoon`.
- Dus als `S` wijst naar een `Student`, moet de `drukAf` van `Student` worden aangeroepen.
- De compiler weet van te voren ("at compile-time") niet naar wat voor object `S` zal wijzen, behalve dat het een `Persoon` is.

# Virtuele functies (3)

- De oplossing is "dynamic binding", we bepalen tijdens het draaien van het programma ("at run-time") welke methode we precies gaan aanroepen.
- Om dit "aan te zetten" moeten we gebruik maken van het keyword `virtual`.
- (Men heeft het vaak over "polymorfisme", we komen daar later in de collegereeks op terug).

# Virtuele functies (4)

```
class Persoon {  
public:  
    virtual void drukAf(void) const {  
        std::cout << "Persoon " << name  
                   << " " << age << std::endl;  
    }  
};  
class Student {  
public:  
    virtual void drukAf(void) const {  
        std::cout << "Student " << name  
                   << " " << age << std::endl;  
        std::cout << "    SID: " << sid  
                   << std::endl;  
    }  
};
```

```
Persoon *p = new Persoon("Jan Persoon", 63);  
Persoon *s = new Student("Student 1", 20, 1234512);  
p->drukAf(); // Roept Persoon::drukAf aan  
s->drukAf(); // Roept Student::drukAf aan.
```

# Virtuele functies (5)

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

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

class Vierkant : public Rechthoek {
public:
    virtual void zetOpSchermb() const {
        std::cout << "Vierkant!" << std::endl;
    }
};

class Cirkel : public Figuur {
public:
    virtual void zetOpSchermb() const {
        std::cout << "Cirkel!" << std::endl;
    }
};

Figuur *f1 = new Rechthoek();
Figuur *f2 = new Vierkant();
Figuur *f3 = new Cirkel();
f1->zetOpSchermb();
f2->zetOpSchermb();
f3->zetOpSchermb();
```

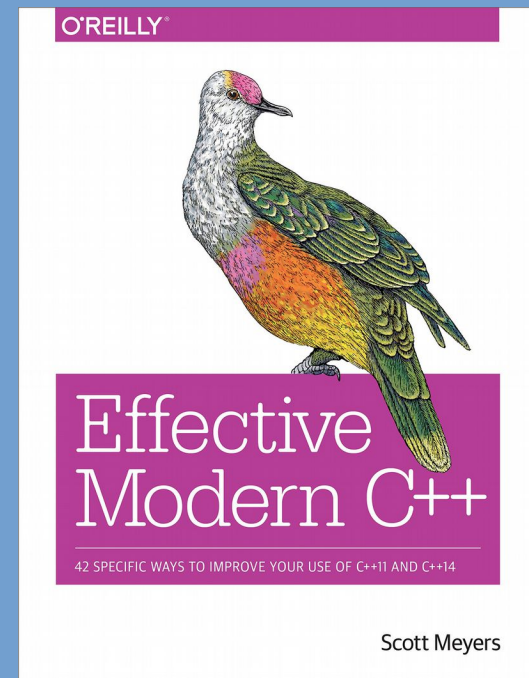
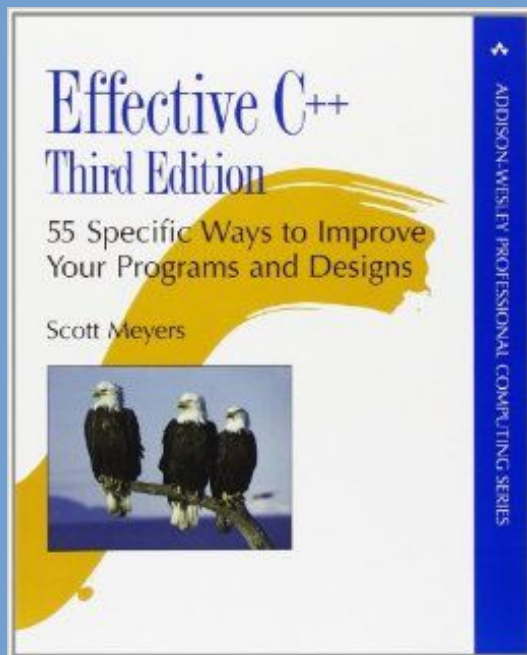


# Advanced C++ Programming

- Als je een C++ boek hebt gelezen ben je nog geen C++ expert.
- Jezelf C++ aanleren kost meerdere jaren.
- Het helpt niet mee dat C++ in vele manieren te ge/misbruiken is...

# Advanced C++ Programming (2)

- Er zijn daarom meerdere boeken geschreven met adviezen en vuistregels voor goed gebruik van C++.
- Vele programmeurs lezen deze boeken. gcc en Clang hebben zelfs een compileroptie voor waarschuwingen over sommige van deze vuistregels: "-Weffc++".



# Multi-paradigm Programming Language

- "Item 1: View C++ as a federation of languages".
- C++ is een federatie van programmeertalen, ben je bewust van de subtalen en welke regels daar toe te passen.
  - C.
  - Object-Oriented C++.
  - Template C++.
  - STL: Standard Template Library. Volgt speciale conventies.

# Class design

- Item 4: "Make sure that objects are initialized before they're used".
- Copy constructor vs. copy assignment operator.
- Item 22: "Declare data members private".

# Object initialisatie

- Bij het initialiseren van objecten moet je goed opletten welke velden wel en niet worden geïnitieerd.
- De regels zijn ingewikkeld ...
  - Primitieve typen ("C typen": int, char, pointers) worden niet geïnitieerd.
  - Voor klassen wordt de default constructor aangeroepen.
- Lezen van niet-geïnitieerde data kan grote gevolgen hebben.
  - In het bijzonder wanneer dit pointers zijn!

# Object initialisatie (2)

- Advies item 4: initialiseer alles in het object in de constructor.
- Maak gebruik van een "member initialiser list".
- Velden moeten in dezelfde volgorde staan als gedeclareerd in de klasse.

```
Persoon::Persoon(const std::string &name,  
                 const int age)  
    : name(name), age(age)  
{ }
```

# Initialisatie vs. assignment

- **Copy constructor:** initialiseer een object, met een ander object van hetzelfde type.
- **Copy assignment operator:** kopieer de waarde van het ene object naar het andere van hetzelfde type.
- Als deze niet zelf definieert, genereert de compiler standaard versies voor je wanneer nodig (Item 5).
- Merk ook op dat pass-by-value een kopie maakt, in het geval van een object zal de copy constructor worden aangeroepen.

# Copy constructor (2)

```
class Test {  
public:  
    /* default constructor */  
    Test();  
    /* copy constructor */  
    Test(const Test &t);  
    /* copy assignment operator */  
    Test &operator=(const Test &t);  
};
```

```
Test t;  
Test r(t);    // copy constr.  
t = r;        // assign  
Test s = r;   // copy constr.!
```



# Copy constructor (4)

- We kunnen nu het voordeel van de member initialisation list beter begrijpen.

```
Persoon::Persoon(const std::string &_name,  
                 const int _age) {  
    name = _name;  
    age = _age;  
}
```

- In dit geval voor `name`: eerst default constructor, gevolgd door copy assignment!

# Declare data members private

- Als de members van een klasse public zijn, kan iedereen er mee rommelen.
- Gebruik van methoden om de members aan te passen geeft je meer controle.
  - Je kunt read-only, write-only members maken.
  - We zagen al een voorbeeld van een "get" methode.
- We noemen dergelijke methoden "getters and setters" of "accessor functions".

# Declare data members private (2)

- Een ander groot voordeel van methoden:
  - Je kunt de private members later veranderen.
  - De methode blijft in stand!
  - Of een berekening toevoegen, aanpassing ander veld toevoegen, aanroepen notificatie routine, etc, etc.
- Gebruikers van de methode hoeven niet te worden aangepast.
- We noemen deze conventie "encapsulation".

# Declare data members private (3)

```
class Coördinaat {  
    public:  
        int x, y;  
  
        Coördinaat(const int x,  
                    const int y)  
            : x(x), y(y)  
        { }  
};
```

```
...  
Coördinaat c(4, 5);  
c.x = c.y;
```

```
class Coördinaat {  
    private:  
        int x, y;  
    public:  
        Coördinaat(const int x,  
                    const int y)  
            : x(x), y(y)  
        { }  
        int getX(void) const {  
            return x;  
        }  
        void setX(void);  
        int getY(void) const;  
        void setY(void);  
};
```

```
...  
Coördinaat c(4, 5);  
//c.x = c.y;  
c.setX(c.getY());
```

# Gebruik van const

- Item 2: Prefer consts, enums, inlines to #defines
- Item 3: Use consts whenever possible
- Item 20: Prefer pass-by-reference-to-const to pass-by-value.

# Geen magic numbers!

- Een eerste advies: laat nooit "losse getallen" slingeren in code. Geef getallen altijd een symbolische naam.

```
/* NEE */  
char buffer[80];  
...  
fgets(buffer, 80, stdin);
```

- Wat als je een "80" vergeet te veranderen?

# Geen magic numbers! (2)

- In C was het altijd gebruikelijk om "#define" te gebruiken.

```
/* OK */
```

```
#define BUF_LEN 80
```

```
char buffer[BUF_LEN];
```

```
...
```

```
fgets(buffer, BUF_LEN, stdin);
```

# Item 2

- Nu zegt Item 2: in plaats van defines, gebruik, consts, enums, inlines.
- Waarom?
  - defines worden verwerkt door de processor en komen niet in de symbol table terecht (debugger zal de naam niet herkennen).
  - defines staan los van scope.
  - ...
- Bij gebruik `const` kun je ook een type meegeven.



# Item 2 (2)

```
#define MODULE_NAME "My Module"
```

```
/* zowel string als pointer zijn const */  
const char * const ModuleName  
    = "My Module";
```

```
/* of misschien liever een C++ string */  
const std::string ModuleNaam("My Module");
```

# Item 2 (3)

- Je kunt in C met "#define" ook macros definieren.
- Soms raar gedrag, oppassen met specificeren, geen type safety ...
- Advies: gebruik inline functies.

```
inline int maxValue(const int a, const int b)
{
    return a > b ? a : b;
}
```

- (En gebruik templates: zie later).

# Item 3

- Item 3: use consts whenever possible.
- Het mooie van const is dat je aangeeft dat een variabele niet mag worden veranderd. De compiler dwingt dit af!
- Maak het een gewoonte om zaken die niet mogen worden aangepast const te declareren.

# Item 3 (2)

- Dingen die je ook als const kunt declareren:
  - Returnwaarde.
  - Functieparameters.
- Maar ook methoden!
- Een const methode mag niets aan een object veranderen.
- Als je een const object hebt, kun je alleen de const methoden aanroepen.

# Item 3 (3)

```
class Persoon {  
public:  
    const std::string &getName(void) const  
    {  
        return name;  
    }  
};
```

# Const correctness

- Er wordt vaak gesproken over "const correctness".
- Dit is het geval als je const gebruikt om "const methoden" aan te geven.
- En ook wanneer je const gebruikt bij functieparameters waar nodig.
- Take home message: gebruik altijd const wanneer je kunt.

# Pass by reference to const

- In C++ is "call-by-value" of "pass-by-value" standaard.
  - Net als in C.
- Functieparameters zijn dus kopieën.
- Voor objecten: aanroep copy constructor!
- Kopieren kost tijd!
  - Denk aan: grote objecten, welke meerdere strings bevatten, andere objecten, etc, etc.
- Voor een enkele functieaanroep, zouden dus vele functieaanroepen (constructoren, destructoren) kunnen worden aangeroepen.

# Pass by reference to const (2)

- Kies voor "pass-by-reference-to-const", er wordt een reference naar het object doorgegeven, welke niet mag worden veranderd.
- Snel!
- Hierdoor is const correctness zo belangrijk!

```
bool isGeslaagd(const Student &student);
```



# C++-style casting

- We kwamen eerder in het hoorcollege al casting tegen:

```
int a = (int)3.45;  
int *myarray = (int *)malloc(sizeof(int)*10);  
void *some_data;  
char *buffer = (char *)some_data;
```

- Casting wordt gebruikt om (expliciet) type conversies af te dwingen.

# Casting & C++

- Voordat we verder gaan ...
- Item 27: minimize casting.
- Het advies is om in C++ *zo weinig mogelijk* gebruik te maken van casting.
- Door expliciet te casten overstem je de type-engine van de compiler.
- Je kan de compiler dingen laten doen die geen goed idee zijn, soms gebeuren er door de casting dingen die je niet zou verwachten.

# Casting & C++ (2)

- Als je vast zit met een compiler warning en je probeert deze op te lossen met een cast, denk dan tweemaal na!
- Is er wellicht iets mis met het ontwerp?
- Kunnen we het zonder cast oplossen?
- Als je moet casten, werk dan met de C++-style casts, omdat het duidelijker is wat er zal gaan gebeuren.
  - Daarnaast kun je er met grep makkelijker naar zoeken.

# C++-style casting (2)

- `const_cast<T>(expr)`, kan de `const` qualifier van objecten wegcasten. Alleen gebruiken bij absolute noodzaak!
- `dynamic_cast<T>(expr)`, hiermee kan worden bepaald of een object van een bepaald type is. Bijvoorbeeld, is dit "Figuur" een "Cirkel"?
  - `Figuur *f; Cirkel *c = dynamic_cast<Cirkel*>(f);`
  - Als `f` niet van type `Cirkel` is, dan wordt `NULL` geretourneerd.
  - Heeft wel een run-time cost!
  - Als je dit moet gebruiken, duidt het vaak wel op een probleem in het ontwerp.

# C++-style casting (3)

- `static_cast<T>(expr)`, kan worden gebruikt voor niet-const naar const conversies, int naar double, conversies van pointer typen zoals void pointer conversies. Oftewel, de meeste oude C-style casts kunnen als static cast worden geschreven.
- `reinterpret_cast<T>(expr)` wordt gebruikt voor low-level casts waarbij de waarde op bit-niveau opnieuw wordt geïnterpreteerd. De resultaten zijn afhankelijk van het platform.
  - Denk terug aan de discussie over pointers van verschillende grootten en little vs. big-endian.
  - Dit type cast komt zeer weinig voor.

# Operator overloading

- Nog even verder met die klasse voor coördinaatparen:

```
class Coördinaat
{
    private:
        int x, y;

    public:
        Coördinaat(void)
            : x(0), y(0)
        { }
        Coördinaat(const int x, const int y)
            : x(x), y(y)
        { }
};
```

# Operator overloading (2)

- Laten we onze klasse gaan gebruiken.

```
Coördinaat a(3, 5);  
Coördinaat b(5, 7);
```

- Het zou aardig zijn als we a en b bij elkaar konden optellen.
- Je zou een methode kunnen schrijven ...

```
a.telop(b);
```

# Operator overloading (3)

- Zou het niet veel aardiger zijn als we gewoon de + operator konden gebruiken?
- Dat kan, als we de compiler vertellen hoe hij de + operator moet toepassen op objecten van type Coördinaat.
- Dit doen we middels "operator overloading".



# Operator overloading (4)

- Een eerste poging ...

```
Coordinaat operator+(const Coordinaat &a,  
                    const Coordinaat &b)  
{  
    Coordinaat c;  
    c.x = a.x + b.x;  
    c.y = a.y + b.y;  
    return c;  
}
```

- Probleem! x en y zijn private! operator+ is geen methode (geen member function).

# Friend functions

- Om dit te oplossen maken we de non-member operator+ een vriend van onze klasse Coördinaat.
- Hierdoor krijgt deze functie toegang tot de private en protected velden.
- In de klassedefinitie:

```
class Coördinaat {  
    ...  
    friend Coördinaat operator+(const Coördinaat &a,  
                                const Coördinaat &b);  
    ...  
};
```

# Friend functions (2)

- Je kunt ook andere klassen een "friend" maken zodat zij toegang krijgen tot de private velden.
- **BELANGRIJK:** gebruik dit alleen met goede redenen en niet om een slecht ontwerp werkbaar te maken.

# Operator overloading (5)

- We kunnen nu schrijven:

```
Coordinaat a(3, 5);  
Coordinaat b(5, 7);
```

```
Coordinaat c = a + b;
```

- We maken de oplossing wat mooier, we definiëren `operator+=` als member functie en maken hier in `operator+` gebruik van.
  - Gaat dupliceren van code tegen.
  - `operator+` hoeft geen friend meer te zijn!

# Operator overloading (6)

```
class Coördinaat {  
    ...  
    Coördinaat &operator+=(const Coördinaat &b)  
    {  
        this->x += b.x;  
        this->y += b.y;  
        return *this;  
    }  
};  
  
Coördinaat operator+(const Coördinaat &a,  
                     const Coördinaat &b)  
{  
    /* Gebruikt impliciete copy constructor */  
    Coördinaat c(a);  
    c += b;  
    return c;  
}
```

# Operator overloading (7)

- We willen nu onze coördinaten afdrukken. Voegen we nu een methode "drukAf" toe?
- Of zouden we ook iets met overloading en "<<" van cout kunnen doen?

# Operator overloading (8)

```
std::ostream &operator<<(std::ostream &out,  
                        const Coordinaat &c)  
{  
    return out << "(" << c.x << ", " << c.y << ")";  
}
```

- Deze functie moet wel als friend worden gemarkeerd!
- Kan niet als member functie worden toegevoegd, waarom niet?
- We kunnen nu doen:

```
Coordinaat a(3, 5);  
std::cout << a << std::endl;
```

# Operator overloading (9)

- We mogen al schrijven:

```
Coordinaat a(3, 5);  
Coordinaat b(5, 7);
```

```
a = b;
```

- Hoe weet de compiler wat te doen voor "="?



# Operator overloading (10)

- Wanneer mogelijk genereert de compiler zelf een "operator=" methode.
- Herinner de discussie over "copy assignment operator".
- Soms nodig dit zelf te doen, dit kan als volgt als member functie:

```
Coördinaat &operator=(const Coördinaat &c)
{
    /* ... kopieer c naar this ... */
    return *this;
}
```

# Operator overloading (11)

- We zagen zojuist en ook al bij "operator+=" een returntype van "Coordinaat &", waarom is dit nodig?
- Dit komt omdat het mogelijk is om te schrijven:

```
a = b = c;  
// en ook  
a = b += c;
```

- Er wordt `b = c` uitgevoerd, het resultaat hiervan wordt gebruikt in de toekenning aan `a`.
- (Item 10: Have assignment operators return a reference to `*this`).

# Operator overloading (12)

- Nog een advies voor assignment operators, zoals = en +=: houd er rekening mee dat `a = a` is toegestaan.
- Als hier geen rekening mee wordt gehouden kan dit in sommige gevallen leiden tot problemen. In het bijzonder wanneer er wordt gewerkt met pointers naar dynamisch gealloceerd geheugen.
- Oplossing:

```
Coordinaat &operator=(const Coordinaat &c)
{
    if (this == &c) return *this;
    /* ... normal operation ... */
    return *this;
}
```

- *(Item 11: Handle assignment to self in operator=)*

# Operator overloading (13)

- Voor welke operatoren kunnen we nu precies een overload schrijven?
  - In principe alle gangbare unaire en binaire operatoren.
  - Ook kunnen er overloads worden geschreven voor () en [].
  - Je mag geen nieuwe operatoren definiëren zoals bijv. \*\*.

# Operator overloading (14)

- Hoe zien de functies er dan uit?
  - De naam van de functie begint altijd met `operator`, gevolgd door de operator die je gaat overladen.
  - Houd in gedachten: de overload functie moet op dezelfde manier functioneren als de operator. Dus laat "+" optellen.
  - Toegestaan is: `c = a + b`, je komt dan uit op:

```
Coördinaat operator+(const Coördinaat &a,  
                    const Coördinaat &b)  
{  
    Coördinaat c(a);  
    c += b; /* hergebruik += is een goed idee! */  
    return c;  
}
```

# Operator overloading (15)

- Standaardvorm voor assignment operatoren zoals =, +=, enz.
- Altijd als memberfunctie.

```
Coordinaat &operator+=(const Coordinaat &b)
{
    /* ... copy b to this ... */
    return *this;
}
```

# Operator overloading (16)

- We zagen ook al de standaardvorm voor een output operator:

```
std::ostream &operator<<(std::ostream &out,  
                        const Coordinaat &c)  
{  
    return out << "(" << c.x << ", " << c.y << " )";  
}
```

- Altijd als non-memberfunctie.

# Operator overloading (17)

- Prefix increment zoals ++C als memberfunctie:

```
Coordinaat &operator++( )  
{  
    /* Doe de increment */  
    return *this;  
}
```



# Operator overloading (18)

- Postfix increment ook als memberfunctie.
- Opletten: wat doet C++ precies? Eerst wordt de huidige waarde van C gebruikt in een expressie, daarna volgt pas de increment.
  - `A[ C++ ]` benadert eerst `A[ C ]` pas daarna wordt C opgehoogd.
- Dit gedrag moet worden verwerkt in de implementatie van de operator.

# Operator overloading (19)

- Gebruikelijk is om de implementatie van de prefix-variant aan te roepen.
- Het integer argument aan de functie blijft vaak ongebruikt.

```
Coordinaat operator++(int dummy)
{
    Coordinaat current(*this);
    operator++();
    return current;
}
```

# Operator overloading (20)

- Relationale operatoren worden vaak geïmplementeerd als non-member en worden friend gemaakt.

```
bool operator<(const Coordinaat &a,  
               const Coordinaat &b)  
{  
    ...  
}
```

# Operator overloading (21)

- Voor een compleet overzicht:
  - (en allerlei subtiele details)

<http://en.cppreference.com/w/cpp/language/operators>

# Templates en generic programming

```
int grootste(int a, int b)
{
    return (a > b ? a : b);
}
```

```
int a = grootste(2134, 34672);
double b = grootste(344.556, 56.2340);
long c = grootste(2314123412341234,
                  123412341234);
std::cout << a << " " << b << " " << c << std::endl;
```

```
// Uitvoer: 34672 344 -671776270
```

➤ Hmm ...

# Templates (2)

- Hoe lossen we dit op?
- C manier: zelf dan maar een zelfde functie schrijven voor double, long en wat er ooit nog gaat komen ...
- C++: we schrijven een “function template” aan de hand waarvan de compiler zelf de benodigde versies kan instantiëren.

# Templates (3)

```
template <typename T>  
T grootste(T a, T b)  
{  
    return (a > b ? a : b);  
}
```

- De compiler zal voor ons voorbeeld versies genereren met `T = int`, `T = double`, `T = long`.

# Templates (4)

*/\* De compiler kiest automatisch de goede code \*/*

```
int a = grootste(2134, 34672);  
double b = grootste(344.556, 56.2340);  
long c = grootste(2314123412341234,  
                  123412341234);
```

*/\* Mag ook een variant forceren \*/*

```
float d = grootste<int>(234.5, 7345.52);
```



# Templates (5)

- Hier wordt ook wel naar gerefereerd als "generic programming" of "generics".
- Code wordt zoveel mogelijk generiek gemaakt.
- Hierdoor kunnen we code makkelijker hergebruiken.

# Templates (6)

- Even terug naar onze Coördinaat klasse:

```
class Coördinaat
{
    private:
        int x, y;

    public:
        Coördinaat(void)
            : x(0), y(0)
        { }
        Coördinaat(const int x, const int y)
            : x(x), y(y)
        { }
};
```

# Templates (7)

- Dit zagen we natuurlijk al aankomen:

```
template <typename T>
class Coördinaat
{
    private:
        T x, y;

    public:
        Coördinaat(void)
            : x(0), y(0)
        { }
        Coördinaat(const T x, const T y)
            : x(x), y(y)
        { }
};
```

# Templates (8)

- Om te gebruiken, specificeer het type T in punthaken:

```
Coordinaat<int> a(1, 4);  
Coordinaat<float> b(34.235, 363.2345);
```

# Templates (9)

- En natuurlijk kunnen we operator overloading combineren:

```
template <typename T>
class Coördinaat
{
    ...
    /* Er verandert niets! De data members zijn
       * al gedeclareerd als T */
    Coördinaat &operator+=(const Coördinaat &b)
    {
        this->x += b.x;
        this->y += b.y;
        return *this;
    }
};
```

# Templates (10)

- Voor non-member operatoren is het in dit geval wel makkelijker om die als friend non-member binnen de klasse te definiëren:

```
template <typename T>
class Coördinaat
{
    ...
    friend std::ostream &operator<<(std::ostream &out,
                                    const Coördinaat &c)
    {
        return out << "(" << c.x << ", " << c.y << ")";
    }
};
```

# Templates (11)

- Een ander voorbeeld: een generieke linked list.
- De C-manier is om een void pointer te gebruiken:

```
struct LinkedList
{
    struct LinkedList *next;
    void *data;
};
```

- Problemen:
  - Je zit vast aan pointers.
  - Niet type-safe! Wie geeft garantie dat er een pointer naar het juiste type wordt gezet?

# Templates (12)

- In C++ gebruiken we templates:

```
template <typename T>
struct LinkedList
{
    struct LinkedList *next;
    T data;
};
```

```
LinkedList<int> *head = new LinkedList<int>( );
head->data = 42;
head->next = new LinkedList<int>( );
head->next->data = 235;
head->next->next = NULL;
```



# Templates (13)

- Een template mag ook meerdere argumenten hebben:

```
template <typename T,  
          size_t N>  
struct FixedArray {  
    T values[N];  
  
    size_t size(void) {  
        return N;  
    }  
  
    T &operator[](size_t offset) {  
        return values[offset];  
    }  
};
```

# Templates (13)

- Een template mag ook meerdere argumenten hebben:

```
template <typename T,  
          size_t N>  
struct FixedArray {  
    T values[N];  
  
    size_t size(void) {  
        return N;  
    }  
  
    T &operator[](size_t offset) {  
        return values[offset];  
    }  
};
```

```
int main(void)  
{  
    FixedArray<float, 10> array;  
  
    for (int i = 0; i < array.size(); ++i)  
        array[i] = i + 3;  
  
    for (int i = 0; i < array.size(); ++i)  
        std::cout << array[i] << " ";  
    std::cout << std::endl;  
  
    return 0;  
}
```

# Templates (14)

- class of typename?

```
template <typename T> MyType;  
template <class C> MyType;
```

- Is er een verschil? Nee, in deze context niet.

# Standard Template Library

- Templates zijn eigenlijk in C++ geïntroduceerd zodat er een type-safe standard library kon worden geschreven.
  - Met daarin type-safe containers en generieke implementaties van algoritmen.
- Standard Template Library (STL).
- We nemen een kijken naar een aantal veelgebruikte functionaliteiten.

# vector

- Een `vector` is in feite een array die automatisch meegroeit.
- Het type objecten dat we opslaan is het template argument.

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

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

```
/* Of initialiseer een vector met  
   * 4 floats met waarde 3.14. */
std::vector<float> floats(4, 3.14);
```

# vector (2)

- Een gebruikelijke manier om elementen in een STL container te bezoeken is door gebruik te maken van een iterator object.

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

- Iterators werken voor verschillende containers op dezelfde manier.

# vector (3)

- In het geval van `vector` is subscription ook toegestaan.

```
std::cout << floats[2] << std::endl;
```

# vector (4)

- Elk object wordt als template argument geaccepteerd, zo ook onze Coördinaat klasse.

```
std::vector<Coördinaat> coördinaten;  
coördinaten.push_back(Coördinaat(3, 4));  
coördinaten.push_back(Coördinaat(16, 1));  
coördinaten.push_back(Coördinaat(1, 5));
```

```
for (std::vector<Coördinaat>::const_iterator it = coördinaten.begin();  
      it != coördinaten.end(); ++it)  
    std::cout << *it << " ";  
std::cout << std::endl;
```



# vector (5)

- We gebruikten een `const_iterator`: het object waar de iterator naar wijst mag dan niet worden aangepast.
- Bij een normale `iterator` kan dat wel:

```
int i = 0;  
for (std::vector<int>::iterator it = integers.begin();  
     it != integers.end(); ++it, ++i)  
    *it = i;
```

# vector (6)

- We kunnen ook een reeks figuren in een vector opslaan. We gebruiken pointers zodat de virtuele methoden correct worden aangeroepen.

```
std::vector<Figuur*> figuren;  
figuren.push_back(new Rechthoek());  
figuren.push_back(new Vierkant());  
figuren.push_back(new Cirkel());  
for (std::vector<Figuur*>::const_iterator it = figuren.begin();  
     it != figuren.end(); ++it)  
    (*it)->zetOpScherm();
```

# list

- STL kent ook een doubly-linked list: `list`.
- Je kunt deze op precies dezelfde manier gebruiken.

```
#include <list>
std::list<int> getallen;
getallen.push_back(521);
getallen.push_back(213);
getallen.push_back(641);

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

# pair

- Met `pair` kun je een paar maken van twee verschillende typen waarden.
- De waarden zijn uit te lezen met `.first` en `.second`.
- `std::make_pair` is een hulpfunctie om paren te maken.

```
std::pair<std::string,int> leeftijd("Joop", 53);  
std::cout << leeftijd.first << " " <<  
          << leeftijd.second << std::endl;  
leeftijd = std::make_pair("Karel", 23);
```

# pair (2)

- We kunnen natuurlijk ook een vector van paren maken!
- Let op! Klassiek C++ vereist een spatie tussen > en >.

```
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;
```

# typedef

- STL is niet altijd bevordelijk voor de gewrichten ...
- Met typedef kun je een soort "nickname" voor een type maken.
- Het is gebruikelijk typedefs op te nemen buiten functies.
- typedef mag ook binnen een klasse definitie worden gebruikt.

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

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

# map

- Ons vorige voorbeeld leent zich meer om te worden opgeslagen in een map.
- Een map is een associatieve container, een collectie van paren (verg. dictionary in Python).

```
typedef std::map<std::string, int> LeeftijdMap;
```

```
LeeftijdMap leeftijden;  
leeftijden.insert(std::make_pair("Joop", 53));  
leeftijden["Karel"] = 23;  
leeftijden["Ida"] = 32;
```

```
for (LeeftijdMap::const_iterator it = leeftijden.begin();  
     it != leeftijden.end(); ++it)  
    /* Iterator wijst naar een key-value paar in de map */  
    std::cout << it->first << ", " << it->second <<  
std::endl;
```

# Algorithms

- De headerfile `algorithm` bevat allerlei generieke algoritmen.
- Deze kunnen worden toegepast op verschillende containers.
- We geven iterators mee om aan te geven op welke reeks van elementen een operatie moet worden uitgevoerd.



# Sorting & Reverse

```
// Nodig: #include <algorithm>
```

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

```
std::sort(integers.begin(), integers.end());
```

```
/* Waarden in container omdraaien */
```

```
std::reverse(integers.begin(), integers.end());
```

# Zoeken

```
std::vector<int> integers;  
  
std::vector<int>::iterator it =  
    std::find(integers.begin(), integers.end(), 643);  
if (it != integers.end())  
    *it = 9234;
```

# Kleinste element

```
std::vector<int>::iterator it =  
    std::min_element(integers.begin(), integers.end());  
std::cout << *it << std::endl;
```

- En er is ook een `max_element`.

# En meer ...

- En zo zijn er nog veel meer algoritmen die klaar staan.
- Dit zijn allemaal "function templates".
- Dus toepasbaar op allerlei containers, ook containers die je zelf schrijft als deze aan de richtlijnen voldoen.

# Naslag

- Op de website van het vak zijn complete (en functionele) codevoorbeelden te vinden, in plaats van de losse stukjes die we op de slides plaatsen.
- Daarnaast zijn er goede websites met documentatie over de volledige STL:
  - [<http://www.cplusplus.com/reference/>]
  - [<http://en.cppreference.com/w/cpp/container>]

# Namespaces

- Kort nog iets over “namespaces”.
- Namespaces worden gebruikt bij het organiseren van grote projecten.
- Het is netjes om grote stukken van een module in een aparte namespace te zetten.
- De echte reden is dat men naamconflicten probeert te vermijden in grote projecten.
  - Wat gebeurt er als een klasse "Rechthoek" in meerdere componenten voorkomt?

# Namespaces (2)

- Je kunt binnen een "namespace" variabelen, functies, klassen, enzovoort declareren.
- Deze vallen dan binnen een zogenaamde "named scope". Eigenlijk krijgt elke naam een prefix.
- Oplossing voor probleem vorige slide: meerdere klassen "Rechthoek" kunnen gewoon voorkomen als deze in verschillende namespaces zijn gedeclareerd.
- Stel we hebben namespaces "moduleA", "moduleB", kan kunnen we als volgt naar de verschillende klassen refereren:
  - moduleA::Rechthoek
  - moduleB::Rechthoek

# Namespaces (3)

- Hoe zit dat dan met `using namespace std` ?
- Alle zaken in de C++ standaard library zijn in de namespace `std` gezet.
- Officieel moet je dan schrijven `std::cout`, `std::endl`, `std::string`, enz.
- Teveel typewerk? Vertel de compiler dat hij standaard ook in de namespace met de naam "std" zoekt:

**`using namespace std;`**

- Nu volstaan `cout`, `string`, enz.



# Namespaces (4)

```
namespace mijnruimte {  
    void zegHallo(void) {  
        std::cout << "hallo daar" << std::endl;  
    }  
  
    static int A = 13;  
}  
  
int main(void) {  
    //zegHallo(); // Fout!  
    //std::cout << A << std::endl; // Fout!  
  
    mijnruimte::zegHallo();  
    std::cout << mijnruimte::A << std::endl;  
  
    using namespace mijnruimte;  
    zegHallo();  
  
    return 0;  
}
```

# Volgende week

- Exception handling.
- Modern C++ features.