**Reader V2CPSE2**

**C++ Programming**

**& Software Engineering 2**

**Studiejaar 2016-2017**



Hogeschool Utrecht, opleiding HBO-ICT, afstudeerrichting Technische Informatica

Wouter van Ooijen

**Versie 1.0 - 2017-11-21**

Contents

[2. Inleiding 4](#_Toc499021811)

[3. Een eenvoudige SFML applicatie 5](#_Toc499021812)

[3.1. Een SFML scherm object 5](#_Toc499021813)

[3.2. Een simpele SFML main 6](#_Toc499021814)

[3.3. Reageren op toetsenbord en muis 7](#_Toc499021815)

[4. Naamgeving 10](#_Toc499021816)

[5. Layout 15](#_Toc499021817)

[6. Undefined behaviour 18](#_Toc499021818)

[7. Lambda’s 20](#_Toc499021819)

[7.1. Data-driven programming 20](#_Toc499021820)

[7.2. Lambda’s 21](#_Toc499021821)

[7.3. function-wraps-constructor 22](#_Toc499021822)

[7.4. std::function 23](#_Toc499021823)

[7.5. Lambda-driven programming 25](#_Toc499021824)

[8. Agile versus Waterfall 29](#_Toc499021825)

[9. De heap 34](#_Toc499021826)

[9.1. New en delete 34](#_Toc499021827)

[9.2. Garbage collection 36](#_Toc499021828)

[9.3. De regel van drie 37](#_Toc499021829)

[9.4. Eigenaarschap 40](#_Toc499021830)

[9.5. Smart pointers 40](#_Toc499021831)

[10. Exceptions 45](#_Toc499021832)

[11. For loop 49](#_Toc499021833)

[12. Factory pattern 52](#_Toc499021834)

[13. Standaard Template Library 56](#_Toc499021835)

[14. Waarden 65](#_Toc499021836)

[14.1. Grootheden 65](#_Toc499021837)

[14.2. Schalen 65](#_Toc499021838)

[14.2.1. Nominaal 65](#_Toc499021839)

[14.2.2. Ordinaal 66](#_Toc499021840)

[14.2.3. Interval 66](#_Toc499021841)

[14.2.4. Ratio 66](#_Toc499021842)

[14.3. Enum 66](#_Toc499021843)

[14.4. Integers en floats 67](#_Toc499021844)

[14.5. Eenheden 68](#_Toc499021845)

[14.6. Stelsels 70](#_Toc499021846)

[14.7. Boost::units 71](#_Toc499021847)

[15. Overzicht van Design Patterns 72](#_Toc499021848)

[15.1. Adapter 72](#_Toc499021849)

[15.2. Command 72](#_Toc499021850)

[15.3. Composite 72](#_Toc499021851)

[15.4. Decorator 72](#_Toc499021852)

[15.5. Dirty Flag 72](#_Toc499021853)

[15.6. Double Buffer 72](#_Toc499021854)

[15.7. Factory 73](#_Toc499021855)

[15.8. Flyweight 73](#_Toc499021856)

[15.9. Game Loop 73](#_Toc499021857)

[15.10. Listener (Observer) 73](#_Toc499021858)

[15.11. MVC 73](#_Toc499021859)

[15.12. NVI 73](#_Toc499021860)

[15.13. Proxy 73](#_Toc499021861)

[15.14. RAII 74](#_Toc499021862)

[15.15. Rule-of-3 74](#_Toc499021863)

[15.16. SFINAE 74](#_Toc499021864)

[15.17. Singleton 74](#_Toc499021865)

[15.18. Update Method / State Machine 74](#_Toc499021866)

[16. Appendix : Visual Studio, SFML, Boost 75](#_Toc499021867)

[16.1. Installeren 75](#_Toc499021868)

[16.1. Solution aanmaken 75](#_Toc499021869)

[16.2. Project aanmaken – via template 76](#_Toc499021870)

[16.3. Project aanmaken – met de hand 76](#_Toc499021871)

[16.4. Een project bewerken 76](#_Toc499021872)

[16.1. Een project naam veranderen 77](#_Toc499021873)

[17. Appendix : week schema 78](#_Toc499021874)

# Inleiding

Deze reader hoort bij de module V2CPSE2, die wordt gegeven in het tweede kwartaal van het tweede studiejaar Technische Informatica. Het overkoepelende thema van dit kwartaal is het realiseren van een PC applicatie in C++. Het parallele theorievak richt zich op datastructuren en hun eigenschapen, en het themaproject behelst het in projectvorm realiseren van een game.

Er wordt gewerkt op Windows en gebruik gemaakt van Microsoft Visual Studio 2015 (met de 2014 C++ compiler), de SFML Game Development library, en de C++ Boost library. SFML wordt ook gebruikt bij de themaopdracht die in het zelfde kwartaal wordt gegeven.[[1]](#footnote-1)

In de V2CPSE2 module worden een aantal Programming en/of Software Engineering onderwerpen behandeld die horen bij realiseren van een PC applicatie met gebruik van SFML. Er wordt van uitgegaan dat de student in vorige modules ervaring heeft opgedaan met C++.

Er is in deze reader gekozen voor Nederlands voor de lopende tekst (om het voor de student zo leesbaar mogelijk te houden), en voor Engels voor de code voorbeelden (om de student toch te laten wennen aan Engels, wat de taal is waarin hij zal moeten leren te werken). Dit leidt er soms toe dat twee woorden worden gebruikt voor hetzelfde ding (bv. bal versus ball, overerving versus inheritance). Bij de meeste gevallen is bij voorbeeld code aangegeven in welk project in de bij deze module behorende .zip file de volledige code te vinden is.

Voor zover relevant wordt gebruik gemaakt van de mogelijkheden die de nieuwe C++11 en C++14 standaarden bieden. Let op dat veel voorbeelden die je op het internet en in boeken vindt maken nog geen gebruik van (alle) C++11/14 features, waardoor ze er op details wat anders uit kunnen zien.

# Een eenvoudige SFML applicatie



|  |
| --- |
| behandelde begrippen en technieken:   * een SFML (circle) shape tekenen * een simpele main() met windows event loop * keyboard en mouse input |

## Een SFML scherm object

De klasse ball is een simpele abstractie van een bal: een gekleurde cirkel op het scherm. We maken in de header bal.hpp gebruik van de declaraties in de SFML/graphics header. Die levert de sf::Vector2f (een floating-point x, y paar) en de sf::RenderWindow, beide binnen de sf namespace. Een bal kan je creëren en tekenen. Het privé deel van de klasse bevat twee attributen: de plek (position) en de grootte (size) van de bal.

|  |  |
| --- | --- |
| #ifndef \_BALL\_HPP  #define\_BALL\_HPP  #include<SFML/Graphics.hpp>  classball {  private:  sf::Vector2f position;  float size;  public:  ball( sf::Vector2f position, float size = 30.0 );  void draw( sf::RenderWindow& window ) const;  };  #endif | |
| header voor een bal object | 01-02 ball.hpp |

De file ball.cpp bevat de implementaties van de beide methoden. In de ball.cpp file wordt gebruik gemaakt van SFML declaraties, dus de <SFML/Graphics.hpp> file moet ook worden ge-include. In dit geval is dat strikt genomen niet nodig omdat dit in ball.hpp ook al gebeurt, maar het is netter om daar niet van afhankelijk te zijn.

Merk op dat bij de constructor ball::ball() de default van de size parameter niet wordt herhaald. Die default heeft de compiler alleen nodig om een aanroep van de constructor te compileren, en we zijn hier juist bezig met de andere kant, de implementatie van de methode.

Bij het initializeren van de variabelen wordt de initiele waarde tussen { } haken gezet in plaats van tussen ( ) haken. Dit heeft de voorkeur omdat er dan geen impliciete (automatische en dus in de source onzichtbare) conversies kunnen plaatsvinden.[[2]](#footnote-2)

De draw methode tekent een (gevulde) cirkel op het scherm. Hiertoe wordt eerst een lokaal object van het type sf::CircleShape aangemaakt. Vervolgens worden twee methoden van dit object aangeroepen om de straal en de plek van de cirkel te zetten, en vervolgens wordt een methode van de window parameter aangeroepen om de cirkel te tekenen.[[3]](#footnote-3) De draw methode hoeft niets te veranderen aan de bal variabelen, dus deze functie is als const decelareerd zodat het mogelijk is het aan te reoepen met een const bal object.

|  |  |
| --- | --- |
| #include<SFML/Graphics.hpp>  #include"ball.hpp"  ball::ball( sf::Vector2fposition, floatsize ):  position{ position },  size{ size }  {}  voidball::draw( sf::RenderWindow&window ) const {  sf::CircleShape circle;  circle.setRadius( size );  circle.setPosition( position );  window.draw( circle );  } | |
| implementatie van een bal object | 01-02 ball.cpp |

## Een simpele SFML main

De main() functie begint met het declareren en initialiseren van een window en een bal. De volgende drie regels maken eerst de window buffer leeg, vervolgens wordt de bal in die buffer getekend, en ten slotte wordt de buffer op het scherm getoond.

De while lus die volgt is nodig om de window zich als een nette windows applicatie te laten gedragen. Zolang de window nog open is wordt een event dat voor de window is gegenereerd opgevraagd. Een event is bv. een muisklik, een muisbeweging, of het indrukken van een toets. Het enige event dat we willen afhandelen is het ‘sluit de window’ event. Als dat komt sluiten we de window. Als er geen events voor ons zijn geven we via de sf::sleep() call de processor de tijd om andere zaken af te handelen (we zijn ten slotte niet het enige programma dat actief is). In een serieus programma moeten in de while lus natuurlijk dingen gebeuren, maar voor dit eerste programma beelden we alleen een witte bal af.

|  |  |
| --- | --- |
| #include<SFML/Graphics.hpp>  #include"ball.hpp"  int main(){  sf::RenderWindow window{ sf::VideoMode{ 640, 480 }, "SFML window" };  ball my\_ball{ sf::Vector2f{ 320.0, 240.0 } };  window.clear();  my\_ball.draw();  window.display();    while( window.isOpen() ){  sf::Event event;  while( window.pollEvent( event) ){  if( event.type == sf::Event::Closed ){  window.close();  }  }  sf::sleep( sf::milliseconds( 20 ) );  }  return 0;  } | |
| stilstaande ball main | 01-02 main.cpp |

Merk op dat de bal niet precies in het midden staat, maar wat meer naar rechtsonder. Dit komt omdat in SFML de plek van een scherm object niet het midden van het object is maar de linker bovenhoek.

Standaard bouwen we het programma in ‘tekst’ modus, dus er verschijnt nog een tweede window waarin tekstuele uitvoer verschijnt.[[4]](#footnote-4) Als we op het X’je van de grafische windows klikt krijgt de while lus een ‘sluit de window’ event en het programma beëindigt zichzelf. In de tekst window moet je dan nog een toets invoeren om ook die window te sluiten.

## Reageren op toetsenbord en muis

In de eventloop kunnen we reageren op input van de gebruiker. Als dat het beeld kan beinvloeden moeten dan ook binnen de eventloop het beeld opnieuwe genereren en op de window zichtbaar maken. Hiervoor voegen we aan de bal twee functies toe om de bal te kunnen bewegen.

|  |  |
| --- | --- |
| class ball {  . . .  public:  void move( sf::Vector2f delta );  void jump( sf::Vector2f target );  . . .  }; | |
| void ball::move( sf::Vector2f delta ) {  position += delta;  }  void ball::jump( sf::Vector2f target ) {  position = target;  } | |
| voeg move en jump toe aan de bal | 01-03 ball.hpp & ball.cpp |

Vervolgens kunnen we in de main loop op andere gebeurtenissen gaan reageren dan alleen het closed event. We kunnen opvragen of een bepaalde toets van het toestenbord of van de muis is ingedrukt. Als een cursortoets is ingedrukt dan verschuiven we de bal een stukje in de bijbehorende richting, en als de linker muisknop is ingedrult dan verplaatsen we de bal in een keer naar de huidige muispositie. Een klein detail is dat we de muispositie terugkrijgen als een sf::Vector2i in plaats van een sf::Vector2f, maar dat is snel opgelost met een functie die een Vector2i omzet naar een vector2f.

|  |  |
| --- | --- |
| sf::Vector2f Vector2f\_from\_Vector2i( sf::Vector2i rhs ){  return sf::Vector2f(  static\_cast< float >( target.x ),  static\_cast< float >( target.y )  );  } | |
| conversie van Vector2i naar Vector2f | 01-03 main.cpp |

In de main() wordt voor de 4 toetsen opgevraagd of ze ingedrukt zijn, en zo ja dan wordt de bal een stapje in de juiste richting bewogen. Als de linker muisknop is ingedrukt dan wordt de positie van de muiscursor opgevraagd, en de bal springt naar die positie. De main handelt de invoer af, tekent het beeld, en wacht natuurlijk een paar milliseconden om andere programma’s een kans te geven.

|  |  |
| --- | --- |
| int main(){  . . .  while( window.isOpen() ){  if( sf::Keyboard::isKeyPressed( sf::Keyboard::Left )){  my\_ball.move( sf::Vector2f{ -1, 0 } );  }  if( sf::Keyboard::isKeyPressed( sf::Keyboard::Right )){  my\_ball.move( sf::Vector2f{ +1, 0 } );  }  if( sf::Keyboard::isKeyPressed( sf::Keyboard::Up )){  my\_ball.move( sf::Vector2f{ 0, -1 } );  }  if( sf::Keyboard::isKeyPressed( sf::Keyboard::Down )){  my\_ball.move( sf::Vector2f{ 0, +1 } );  }  if( sf::Mouse::isButtonPressed( sf::Mouse::Left )){  my\_ball.jump( Vector2f\_from\_Vector2i( sf::Mouse::getPosition( window )));  }  . . .  window.clear();  b.draw( window );  window.display();  sf::sleep( sf::milliseconds( 20 ));  }  . . .  } | |
| bewegende bal main | 01-03 main.cpp |

# Naamgeving



|  |
| --- |
| behandelde begrippen:   * Belang van naamgeving * Afbakening van ‘sub’ woorden * Juiste ‘type’ naam (zelfst. naamwoord, werkwoord, voltooid deelwoord) * Kleinere scope 🡺 kortere naam * Volgorde van woorden in een naam |

Heel vroeger (zeg maar toen ik nog jong was) was het typen en printen van programma’s een enorme opgave. Een veel gebruikt apparaat was de Teletype: een combinatie van toetsenbord, paper tape lezer/schrijver, en printer.



De printer had de fantastische snelheid van 10 tekens per seconde (maar naar een nieuwe regel gaan duurde veel langer) en maakte een oorverdovend lawaai. De paper tape lezer/schrijver (zelfde snelheid, werd vaak gebruikt om een definitieve versie van een programma op te slaan of weer in te lezen) had de zelfde snelheid. Het was dus niet erg praktisch om in je code lange namen te gebruiken. Daarnaast waren er vroeger linkers die alleen naar de eerste 8 .. 12 tekens van een naam keken. Je kon dus niet twee variabelen in je programma hebben die pas na de 12e letter verschilden. Dat waren redenen dat het gebruik van korte en daardoor vaal cryptische namen voor functies en variabelen in zwang is gekomen. Tegenwoordig hebben we full-screen editors waarmee we snel door code kunnen browsen, linkers die namen van een duizenden tekens aan kunnen, en de omvang van source code is al helemaal geen probleem meer. Maar helaas is de gewoonte om dingen in een programma korte, cryptische namen te geven blijven bestaan.

Naamgeving is belangrijk voor de leesbaarheid van code. Aangezien er doorgaans meer tijd zit in het debuggen en onderhouden van code dan in het schrijven van het origineel is het economisch om goede namen te kiezen.



Bekijk de volgende functie. Snap je wat het doel van de functie is, wat voor parameters er mee geven moeten worden, en waarom hij zo geïmplementeerd is? Hoogst waarschijnlijk niet.

|  |
| --- |
| void f( int a, int b ){  g( a );  while( b-- ){  h();  }  } |

Vergelijk dit met het volgende versie van de zelfde functie.

|  |
| --- |
| void ps2\_protocol\_command\_send\_replies\_ignored(  int command,  int n\_replies  ){  ps2\_byte\_send( command );  while( n\_replies-- ){  (void) ps2\_byte\_receive();  }  } |

Zelfs als je niet weet wat ps2 in deze context betekent (het is het keyboard en muis protocol dat PCs gebruikten voor de opkomst van USB) kan je toch zien dat:

* De functie een ps2 commando verzendt en de antwoorden negeert;
* De eerste parameter een commando is, en de tweede het aantal antwoorden.

De implementatie doet wat de naam suggereert: eerste het commando verzenden, en dan n\_replies keer een byte ontvangen en weggooien (de (void) cast maakt duidelijk dat de auteur zich er van bewust was dat er wel een waarde terugkwam, maar daar niets mee wilde doen).

De genereerde code is in beide gevallen het zelfde, maar het verschil in leesbaarheid is enorm. Je zou aan de eerste versie natuurlijk commentaar kunnen toevoegen om het duidelijker te maken, maar je moet toch al namen geven aan functies, variabelen etc., dus is het wel zo handig om dat meteen te gebruiken om de code leesbaar te maken. ‘Het beste commentaar is commentaar dat er niet hoeft te staan.’

Hoe kies je een goede naam voor je klassen, attributen, methoden, functies, variabelen, constanten, parameters, etc.? Gezond verstand is erg belangrijk, maar er zijn een aantal goede vuistregels.

De hoofdregel is dat een naam het de lezer makkelijk moet maken je programma te lezen en te begrijpen. Voor iets dat hij toch wel begrijpt, bv. de loop variabele van een for() die alle elementen van een array op 0 zet, hoef je geen lange naam te kiezen, dat is zelfs onhandig omdat het meer leeswerk vergt en niet nodig is voor het begrijpen.

|  |
| --- |
| int counters[ number\_of\_counters ];  for( int i = 0; i < number\_of\_counters; ++i ){  counters[ i ] = 0;  } |
| int counters[ number\_of\_counters ];  for( int counter\_index = 0; counter\_index < number\_of\_counters; ++counter\_index ){  counters[ counter\_index ] = 0;  } |

Een lange naam is hier niet nodig en zelfs ongewenst omdat de scope (de regels waarin de i gebruikt wordt) heel klein is (2 regels) en het idoom (een index gebruiken om door een array heen te lopen) zo standaard is dat iedere programmeur dit in 1 blik herkent.

Namen en commentaar zijn bedoeld voor de lezer van een programma. Om voor een zo groot mogelijk publiek leesbaar te zijn is onvermijdelijk dat ze in het Engels geschreven worden. Dat hoeft geen Shakespeare-kwaliteit Engels te zijn, maar het moet wel begrijpelijk en niet mis te verstaan zijn.

Naarmate een ding (functie, variabele, etc.) meer verspreid door een programma, en/of op een minder direct herkenbare manier wordt gebruikt is het belangrijker dat de naam van dat ding aangeeft wat hij is of doet.

De naam van een ding moet het juiste soort woord zijn: een type, klasse, variabele of object is een ding, en moet dus een zelfstandig naamwoord zijn (of bevatten). Een functie of methode die iets doet (een zgn. side-effect veroorzaakt) is een actie, en moet dus een werkwoord zijn of bevatten. Een functie of methode die iets test en afhankelijk van de uitkpomst een boolean waarde teruggeeft (een zgn. predicaat), moet een voltooid deelwoord of iets dergelijks zijn of bevatten.

Neem als voorbeeld hiervan een functie lowerchar(c) met 1 char argument. Wat doet die functie?

* een argument converteren naar een kleine letter, of
* de geconverteerde waarde teruggeven, of
* checken of het argument al of niet een kleine letter is?

De naam maakt hier geen onderscheid. Betere namen zouden zijn convert\_to\_lowerchar(), converted\_to\_lowerchar(), is\_lowerchar().

De C library functie strcmp is een goed voorbeeld van hoe het niet (meer) moet. Deze functie vergelijkt twee strings. Wat denk je dat deze functie teruggeeft, waar als ze gelijk zijn, of juist waar als ze niet gelijk zijn? Het antwoord is geen van beiden: 1 als de eerste string eerder zou komen bij alfabetische sortering, 0 als ze gelijk zijn, en -1 als de tweede string eerst zou komen.

Een betekenisvolle naam zal al snel uit meerdere woorden bestaan. Helaas is het in de meeste programmeertalen niet mogelijk spaties in een naam op te nemen. Als je de letters van die woorden zomaar achter elkaar plakt levert dat geen erg leesbaar geheel op. Je kan de eerste letter van ieder woord een hoofdletter maken, dat geeft redelijk aan waar de woorden beginnen. Een underscore tussen de woorden maakt het nog makkelijker, en dan kan je die hoofdletters eigenlijk ook wel weglaten.

|  |
| --- |
| Most programming languages do not allow spaces in identifiers. ☹  Justputitalltogetherwithoutseparatorsisnotagoodidea  INTHEBEGINNINGPRINTERSHADNOLOWERCASE  camelCaseIsNamedAfterTheHumpsOfACamel  PascalCaseAlsoCapitalizesTheFirstLetter  Adding\_Underscores\_Makes\_Reading\_Easier  but\_with\_underscores\_who\_needs\_the\_captials  Avoid capitals: tcp\_checksum\_calculate() |

Over de juiste manier om de woorden af te bakenen worden bijna oorlogen gevoerd. Als een taal een grote standaard bibliotheek heeft is het bijna vanzelfsprekend om de stijl van die bibliotheek over te nemen. In Java bv. wordt meestal camelCase of PascalCase gebruikt. De C bibliotheek is er al oud en draagt daardoor de sporen van het ‘iedere letter is duur’ tijdperk: veel afkortingen, en vaak geen afscheiding van de woorden (bv. strcpy). In de C++ bibliotheken worden meestal kleine letters en underscores gebruikt(maar helaas niet altijd).

Als een naam uit meerdere woorden bestaan dan moet je kiezen in welke volgorde je de woorden zet. Vaak worden de woorden in zinsvolgorde gezet, bv start\_airplaine\_engine voor het starten van de motor van een vliegtuig. Een nadeel hiervan is dat ‘start’ niet bij het ding staat dat gestart wordt, nl. de ‘engine’. Een betere volgorde is te doen alsof je met OO objecten te maken hebt, de operatie zou dan zijn airplane.engine.start(). De volgorde is links het grootste (meestomvattende) object, dan in volgorde de steeds kleinere deelobjecten, en dan als laatste de actie. Vervang alle . tekens door \_ en je hebt airplaine\_engine\_start(). Een bijkomend voordeel is dat als je een lijst van dergelijk namen op alfabetische volgorde zet, de operaties op hetzelfde ‘object’ (airplaine\_engine\_...) bij elkaar komen, wat doorgaans handiger is dan de verwante operaties (start\_...) bij elkaar.

Let er bij het kiezen van een naam op dat je noemt wat het ding doet (functie, methode) of wat voor functie het heeft (variabele, parameter, attribuut), niet hoe dat gebeurt. Alleen als de lezer dat moet weten, bv. als je meerdere implementaties biedt die functioneel hetzelfde doen (bv. een ‘gewoon’ array en een array dat zijn elementen in een file opslaat) is het zinning de implementatie in de naam naar voren te laten komen.

In C++ kan je functies en methoden overloaden, maar in C kan dat niet. Als je dus in C dezelfde functie biedt maar met verschillende soorten parameters, bv. print(string), print(int) en print(float), dan is het handig om dat als achtervoegsel aan de naam toe te voegen, dus achter het werkwoord: print\_string(), print\_int(), print\_float().

Een functie of methode die geen veranderingen aanbrengt (in jargon wordt die ‘pure’ genoemd) kan als naam het ding of waarde hebben dat hij teruggeeft, bv. average() of first(). Let wel op dat je dit niet doet bij bv. een methode die het eerste element van een lijst teruggeeft en dat element ook verwijderd, die zou bv. get\_and\_remove\_first moeten heten. Dat is een lange en lelijke naam. Meestal duid zo’n naam er op dat er iets mis is met de functionliteit die je in je functie wil stoppen, maar in dit geval is er een algemeen bekende namen voor: pop, dequeue, of get\_from.

Bij functies die hun argument converteren naar een andere vorm heb je twee keuzes: bv. ascii\_to\_int(), of int\_from\_ascii(). Het eerste lijkt meer op gewoon taalgebruik, maar het tweede heeft toch de voorkeur omdat dan ascii dichter bij het (ascii) argument staat, en int dichter bij de variabele waar het resultaat aan wordt toegekend. Ook komt dit meer overeen met de ‘van rechts naar links’ werking van een assignment statement.

|  |
| --- |
| int n, char \*s;  . . .  n = ascii\_to\_integer( s );  n = integer\_from\_ascii( s ); |

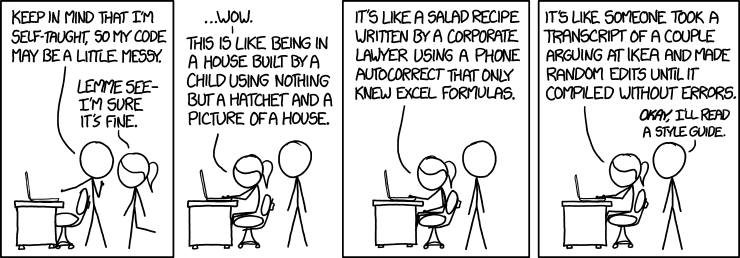
Er zijn talen (bv. Ada en Python) waar de namen van de parameters van een functie of methode in de aanroep genoemd (kunnen) worden. Vergelijk dit met de C++ sitiuatie waar je in de aanroep alleen de waarden van de parameters ziet.

|  |
| --- |
| // Ada  create\_account(  name => ”Dennis Richie”  initial\_amount => 0,  maximum\_overdraft => 1000  );  // C++  create\_account( ”Dennis Richie”, 0, 1000 ); |

In de C++ versie kan je wel raden wat de functie “Dennis Richie” parameter is, maar zelfs als je weet dat de functie nog een begin slado en een maximaal roodstaan bedrag nodig heeft weet je niet in welke volgorde die moeten staan. Probeer dit probleem bij het kiezen van parameters te vermijden. Als je toch parameters hebt waarvan de volgorde verswisseld zou kunnen worden, vermeld dan de volgorde achteraan de naam van je functie.

|  |
| --- |
| create\_account\_name\_initial\_overdraft( ”Dennis Richie”, 0, 1000 ); |

# Layout



Code in een modern programmeertalen heeft een hierargische opbouw. Het is erg bevorderlijk voor de leesbaarheid als die hierargie ook in de layout tot uitrukking komt. Idereen is het er wel over eens dat de code van een ‘dieper’ deel verder naar rechts moet inspringen. Maar dat laat nog een paar details open, met name waar de open { en sluit } haakjes geplaatst moeten worden, en dat is het onderwerp van verhitte discussies. Er zijn een aantal bekende stijlen, die ieder binnen bepaalde programmeertalen en/of projecten zo hun volgelingen hebben.

De Withesmith en Pascal stijlen plaatsen de { } haakjes op het zelfde niveau, maar bij Pascal staan ze op het nieveau van de omringende code, en bij Whitesmith op het niveau van de geneste code.

|  |  |  |
| --- | --- | --- |
| // Whitesmith  if( a > b )  {  max = a;  n++;  }  else  {  max = b;  } |  | // Pascal - ANSI  if( a > b )  {  max = a;  }  else  {  max = b;  } |

Een nadeel van deze stijlen is dat ze aparte regels gebruiken voor zowel de open { als de sluit } haak. Aangezien het aantal regels op een scherm beperkt is, wordt zo de hoeveelheid code die op een scherm getoond kan worden kleiner. Uit onderzoek is gebleken dat code die zonder scrollen bekeken kan worden beter wordt overzien, en dat doordoor minder fouten worden gemaakt die met de samenhang in die code te maken hebben. Andere stijlen plaatsen 1 of beide haken op een code regel. De Pico (compacte) stijl gaat hier in het verst, maar dit heeft als nadeel dat je bij het invoegen van een nieuwe regel aan het einde van een genest blok goed moet oppassen dat je de sluithaak } ‘meeneemt’ naar die nieuwe regel.

|  |  |  |
| --- | --- | --- |
| **// Pico (compact)**  **if( a > b ){**  **max = a; }**  **else {**  **max = b; }** |  | **// K&R - 1TBS**  **if( a > b ){**  **max = a;**  **} else {**  **max = b;**  **}** |

Mijn favoriet is de Kernigan & Richie stijl (naar de twee ‘uitvinders’ van C en Unix), ook wel one-true-brace-style (1TBS) genoemd. Deze stijl is net iets minder compact dan Pico, maar de haken staan niet ‘in de weg’ als je nieuwe regels wil invoegen. Tegenstanders van deze stijl vinden het onhandig dat de open en sluit haken niet boven/onder elkaar staan.

In sommige projecten zullen alle betrokkenen de zelfde editor gebruiken (bv Microsoft Visual Studio), maar vaak is dit niet het geval. Dan is het van belang om geen tabs in je code te gebruiken, want verschillende editors (en printers) gaan daar verschillend mee om, zodat jouw code die in jouw editor er perfect uit ziet in een andere editor volkomen chaotisch overkomt.

De inspring stijl en de manier waarop woorden in een naam worden afgescheiden zijn beide belangrijk voor de leesbaarheid van een programma, maar het is veel belangrijker dat er 1 consequente stijl wordt gebruikt dan welke dat precies is. Als je voor een bedrijf gaat werken zal die meestal een expliciete standaard hebben, of anders wel een impliciete standaard in de al aanwezige code. Is dat niet het geval, dan kan je zelf iets kiezen. Pas die stijl daarna wel consequent toe!

Als een genest blok maar 1 statement bevat is het vanuit de taal niet nodig om daar { } haken omheen te zetten. Het is echter verstandig dit wel te doen, om te voorkomen dat als er later een statement wordt toegevoegd, dit statement onbedoeld buiten het blok komt te staan. Om deze reden stellen de meeste code standaarden is het gebruiken van de { } haken in zo’n situatie plicht.

|  |  |  |
| --- | --- | --- |
| if(( i >= 0 ) && ( i < max ))  array[ i ] = 0; |  | if(( i >= 0 ) && ( i < max ))  array[ i ] = 0;  ++n\_valid; // oeps! |

Het C/C++ switch statement zit wat ongelukkig in elkaar. De case labels zijn echte labels waar heen gesprongen wordt, en daarna gaat de uitvoering van het programma gewoon verder op die plek, dwars door een eventueel volgend case label heen. Gewoonlijk wordt iedere case afgesloten met een break statement, dat er voor zorgt dat de uitvoering van het programma verder gaat na het switch statement. Als zo’n break echter niet aanwezig is dan is het voor de lezer de vraag of de programmeur dit echt zo bedoeld heeft (‘fallthrough’ naar het volgende case block) of dat het een vergissing is. Alle code standaarden die ik ken stellen daarom dat ieder case block moet worden afgesloten met een break.[[5]](#footnote-5)

|  |
| --- |
| switch( c ){  case ’\n’ :  position.x = 0;  position.y= minimum( position.y + 1, 23 );  break;  case ’\v’ :  position.x = 0;  position.y = 0;  break;  default :  display[ position.x, position.y ] = c;  position.x = minimum( position.x + 1, 79 );  break;  } |

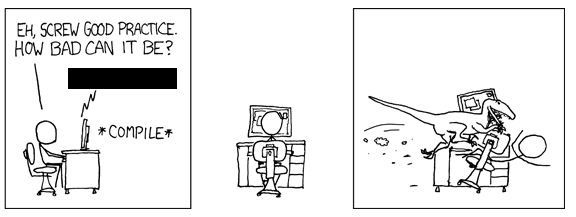
Als illustratie van wat je met een switch statement kan doen: deze rare constructie wordt Duff’s device genoemd. In het switch statement zit een do..while loop verstopt die over een aantal cases heen loopt. Het doel is het copieren van data. Dat kan natuurlijk in een loop die telken 1 int copieert, maar een compiler kan vaak sneller code maken door zo’n loop te ‘unrollen’, en dan hoeft er ook maar eens in de (in dit geval) 8 bytes gecheckt te worden of we al aan het einde zijn. Maar om dat ook te laten werken als het aantal niet deelbaar is door 8 blijven er 0..7 waarden ‘over’. De truuk van deze code is om dat aantal eerst te copieren door in de switch naar het juiste aantal te springen. Aan het einde van de reeks cases kom je dan in de while( --n > 0 ) terecht die er voor zorgt dat alle 8 cases nog ( count + 7 ) / 8 keer herhaald worden.

|  |
| --- |
| send( int \* to, int \* from, int count){  int n = (count + 7) / 8;  switch(count % 8) {  case 0: do { \*to++ = \*from++;  case 7: \*to++ = \*from++;  case 6: \*to++ = \*from++;  case 5: \*to++ = \*from++;  case 4: \*to++ = \*from++;  case 3: \*to++ = \*from++;  case 2: \*to++ = \*from++;  case 1: \*to++ = \*from++;  } while(--n > 0);  }  } |
| Duff’s device |

Duff zelf zegt over deze vondst van hem:

“I feel a combination of pride and revulsion at this discovery. … Many people have said that the worst feature of C is that switches don't break automatically before each case label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against.”

# Undefined behaviour



|  |
| --- |
| behandelde begrippen en technieken:   * Implementation defined behaviour * Undefined behaviour |

Een aantal eigenschappen van C++ zijn door de standaard niet strak gedefinieerd, maar overgelaten aan de compiler schrijver. Een voorbeeld is de omvang van de verschillende soorten integers. Voor iedere specifieke compiler zijn zulke aspecten wel gedefinieerd, en ze moeten gedocumenteerd zijn. Maar als je een programma wil schrijven dat bruikbaar is met verschillende compilers moet je rekening houden met zulke verschillen.

Er zijn ook eigenschappen van C++ die bewust niet gedefinieerd zijn, dit wordt ‘undefined behaviour’ genoemd. Bekende voorbeelden zijn

* het gebruiken (lezen of schrijven) van een stuk geheugen dat niet bij een object hoort, bv. via een ‘wilde’ pointer;
* het gebruiken van een stuk geheugen dat niet *meer* bij het betreffende object hoort;
* het gebruiken van de waarde van een variabele die nooit een waarde heeft gekregen (bv een lokale int variabele)
* overflow bij signed integers (bij unsigned is het gedrag bij overflow juist wel gedefinieerd)

Als je programma iets doet dat ‘undefined behaviour‘ is, dan kan moet je in principe rekening houden met \*ieder\* gedrag, inclusief het formatteren van je harde schrijf of het omtoveren van jou tot een vleermuis.[[6]](#footnote-6) In de praktijk weet je soms wel wat de compiler voor code zal genereren, en er zijn dan ook veel programma’s die (bewust of onbewust) gebruik maken van wat volgens de C++ standaard undefined behaviour is. Hierin schuilt wel een gevaar voor de toekomst: met een volgende versie van de compiler (of een compiler van een andere fambrikant) kan het gedrag opeens heel anders zijn. Dit wordt dan vaak geweten aan een bug in de compiler, maar het is een fout van de programmeur.

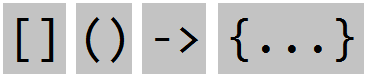
De trend van de laaste jaren is dat de drie bekende C++ compilers (GCC, Clang, Microsoft) steeds meer undefined behaviour gaan herkennen en vervolgende gebruiken voor optimalizatie. Wat denk je dat het volgende programma zal printen?

|  |
| --- |
| signed char c = 127;  c = c + 1;  if (c > 0) {  std::cout << "Larger ";  } else {  std::cout << "Smaller ";  } |

1. Larger
2. Smaller
3. Larger Smaller
4. Geen idee

Het juiste antwoord is “geen idee”, want het programma vertoond undefined behaviour omdat de variabele c zal overflowen. Na undefined behviour is de compiler vrij te doen wat hij wil, dus sommige compilers optimalizeren het hele if statement gewoon weg en printen dus niets. En als ze “42” printen dan voldoen ze ook aan de standaard.

# Lambda’s



|  |
| --- |
| behandelde begrippen en technieken:   * Data-driven programming * Lamda’s * Capture * function-wraps-class truuk * std::function |

## Data-driven programming

In een vorig programma bevatte de main() een reeks if statements die reacties op de verschillende toetsen afhandelden.

|  |
| --- |
| int main(){  . . .  while( window.isOpen() ){  if( sf::Keyboard::isKeyPressed( sf::Keyboard::Left )){  my\_ball.move( sf::Vector2f{ -1, 0 } );  }  if( sf::Keyboard::isKeyPressed( sf::Keyboard::Right )){  my\_ball.move( sf::Vector2f{ +1, 0 } );  }  if( sf::Keyboard::isKeyPressed( sf::Keyboard::Up )){  my\_ball.move( sf::Vector2f{ 0, -1 } );  }  if( sf::Keyboard::isKeyPressed( sf::Keyboard::Down )){  my\_ball.move( sf::Vector2f{ 0, +1 } );  }  if( sf::Mouse::isButtonPressed( sf::Mouse::Left )){  my\_ball.jump( Vector2f\_from\_Vector2i( sf::Mouse::getPosition( window )));  }  . . .  } |

Een reeks van vier vrijwel gelijke if statements is nog wel leesbaar, maar als het er bv. 10 waren geweest dan kan je beter gebruik maken van wat ‘data driven programming’ wordt genoemd: breng de variatie die er is onder in data en ‘interpreteer’ die in bv. een for(:) lus. Als het aantal acties toeneemt, wordt deze versie snel veel leesbaarder dan de versie met losse if’s, ook al omdat de lijst met acties buiten de main loop geplaatst kan worden.

|  |  |
| --- | --- |
| const struct {  sf::Keyboard::Key key;  float x; float y;  } actions[] = {  { sf::Keyboard::Left, -1.0, 0.0 },  { sf::Keyboard::Right, +1.0, 0.0 },  { sf::Keyboard::Up, 0.0, -1.0 },  { sf::Keyboard::Down, 0.0, +1.0 }  };  . . .  for( auto& action : actions ){  if( sf::Keyboard::isKeyPressed( action.key )){  b.move( sf::Vector2f{ action.x, action.y } );  }  } | |
| aparte lijst met acties, buiten de main | 03-01 main.cpp |

Merk op dat we nergens hoeven aan te geven hoeveel acties er zijn, dus daar kunnen we ook geen fout in maken. De omvang van het array wordt bepaald door het aantal initializatie elementen, en de het aantal iteraties van het for( : ) statement wordt weer bepaald door het aantal elementen in het array.

Met een derglijke lijst van acties zijn we nogal beperkt in wat we kunnen opgeven als trigger (een toets die is ingedrukt) en als actie (het verplaatsen van de bal met een bepaalde waarde voor de x en y richtingen).

## Lambda’s

Voor wat meer flexibiliteit kunnen we gebruik maken van lambda’s. Een lambda in C++ is een functie die op de plaats van een expressie geschreven wordt, en als functie parameter kan worden doorgegeven en of in een variabele kan worden opgeslagen. Een lambda begint met een capture clause (de [ ] haakjes en wat daar tussen staat), gevolgd door een (optionele!) parameter lijst, en als laatste de lambda body. De parameter lijst en de body zijn hetzelfde als bij een functie. Het return type hoef je niet aan te geven, de compiler bepaalt dit dan aan de hand van de return statements in de body.[[7]](#footnote-7) Je kan ook een expliciet return type aangeven, maar dan moet dit met de ‘trailing return type’ syntax.[[8]](#footnote-8) De capture clause kan je niet weglaten, zelfs als die leeg is, want die [ ] haakjes geven aan dat wat volgt een lambda is.

Het type van een lambda is ‘verborgen’: iedere lambda heeft een ander type, zelfs als de parameter lijst, return type en body precies het zelfde zijn. Als je een lambda wil opslaan in een variabele dan kan dit door auto te gebruiken, maar je kan aan die variabele noot een andere lambda toekennen, want iedere lambda is een ander type, ook al zien ze er precies hetzelfde uit.

|  |  |
| --- | --- |
| // lambda's without parameters, returning void  auto empty\_void\_lambda\_1 = [](){};  auto empty\_void\_lambda\_2 = []()->void {};  auto empty\_void\_lambda\_3 = []{};  // error  empty\_void\_lambda\_1 = [](){};  // lambda’s incrementing their parameter, returning int  auto increment\_1 = []( int & n ) { return ++n; };  auto increment\_2 = []( int & n )->int { return ++n; };  // idem, returning double  auto increment\_3 = []( int & n )->double { return ++n; };  auto increment\_4 = []( int & n ) { return 0.0 + ++n; }; | |
| lambda examples | 03-02 |

Als je een lambda wilt meegeven aan een functie dan kan dat door door het type van de lambda als (impliciete) template parameter door te geven. Je hoeft de lambda dan niet eerst op te slaan in een variabele (maar dat mag wel): lambda’s worden meestal aangemaakt in een functie aanroep (anders had je bijna net zo goed een losse functie kunnen schrijven). Je moet natuurlijk wel zorgen dat de echte lambda die je meegeeft past bij hoe de lambda parameter in de functie wordt aangeroepen. Doe je dit fout, dan krijg je een foutmelding op de plaats waar de lambda in de template (op een verkeerde manier) wordt aangeroepen.

|  |  |
| --- | --- |
| template< typename T >  void pass\_a\_lambda( T lambda ){  lambda();  }  empty\_void\_lambda\_1();  pass\_a\_lambda( empty\_void\_lambda\_1 );  pass\_a\_lambda( []{} );  // error at the lambda() call in the pass\_a\_lambda function  pass\_a\_lambda( []( int x ){} ); | |
|  | 03-02 |

## function-wraps-constructor

Als je op deze manier een lambda wil meegeven aan de constructor van een klasse dan is dit wat lastig omdat je dan eerst het type van de klasse precies moet aangeven (inclusief de template parameter). Dit kan als je een lambda variabele hebt door met decltype() het type van die variabele op te vragen. Maar dan moet je de lambda dus eerst in een variabele opslaan, en die variabele twee keer noemen in de constructor aanroep.

|  |  |
| --- | --- |
| template< typename T >  class store\_a\_lambda\_class {  private:  T lambda;  public:  store\_a\_lambda\_class( T lambda ):  lambda( lambda )  {}  void run(){  lambda();  }  };  // this is OK, but it is ugly  auto empty = []{};  auto store1 = store\_a\_lambda\_class< decltype( empty ) >( empty );  store1.run();  // this does not work because the two lambda's have a different type  auto store2 = store\_a\_lambda\_class< decltype(empty) >( []{} ); | |
| Een constructor kan geen impliciete klasse template parameters krijgen | 03-03 |

Een truuk om hier omheen te werken is om de constructor aanroep in een functie template onder te brengen. Een functie template kan je aanroepen zonder de template parameter expliciet te noemen, maar kan die template parameter wel gebruiken, bv. om het return type an de functie te specificeren. Dit werkt dus ook met een lambda die je in de aanroep aanmaakt.

|  |  |
| --- | --- |
| template< typename T >  store\_a\_lambda\_class< T > store\_a\_lambda( T lambda ){  return store\_a\_lambda\_class< T >( lambda );  }  auto store3 = store\_a\_lambda( empty\_void\_lambda\_1 );  store3.run();  auto store4 = store\_a\_lambda( []{} );  store4.run(); | |
| function – wraps - constructor | 03-03 |

## std::function

Idere lambda is een uniek type, dus je kan niet zomaar lamda’s opslaan in een variabele die je creeert met het type van een lambda. Dit kan wel in std::function. Dit is een (template) class waarin je lambda’s van een specifieke signatuur[[9]](#footnote-9) kan opslaan. Je moet als template parameter de signatuur opgeven. Die ziet er uit als een functie header, maar zonder de functie naam en zonder parameter namen. Aan zo’n variabele kan je dus een lambda (van die signatuur) toekennen.

|  |  |
| --- | --- |
| #include <functional>  std::function< int(int) > f;  f = []( int x ){ return x; };  std::cout << "identy " << f( 5 ) << "\n";  f = []( int x ){ return 1 + x; };  std::cout << "next " << f( 5 ) << "\n";  f = []( int x ){ return 2 \* x; };  std::cout << "double " << f( 5 ) << "\n"; | |
| std::function | 03-04 |

De capture clause van een lambda bepaalt hoe de lambda omgaat met variabelen die gedefinieerd zijn in de functie waar de lambda is gecreerd, maar binnen de lambda body worden gebruikt. [[10]](#footnote-10) De keuze is, net als bij functie parameters, by value of by reference. By value betekent in dit geval automatisch by *const* value. Als je niets aangeeft, dan kan je binnen de lambda geen variabelen van zijn ‘omringende’ functie gebruiken.[[11]](#footnote-11) Je kan per variabele specificeren of je die by value of by reference wilt gebruiken met respectievelijk de naam, of & gevolgd door de naam, met een komma tussen de verschillende specificaties. Dat ziet er dus ongeveer uit als een parameter lijst, maar zonder de parameter typen ( de compiler weet de naam, dus hij kan het type opzoeken). Met een losse & als eerste kan je aangeven dat de default (voor namen die je niet expliciet noemt) by reference is. Met = kan je aangeven dat de default by (const) value is.

|  |  |
| --- | --- |
| // std::cout is global, so you can use it without capturing  auto here = [](){ std::cout << \_\_FILE\_\_ << ":" << \_\_LINE\_\_ << "\n"; };  here();  // line is in the enclosing function, so you must capture it,  // either by copy or by reference, either individually or collectively  int line = \_\_LINE\_\_;  [ & ]() { std::cout << \_\_FILE\_\_ << ":" << line << "\n"; }();  [ &line ] { std::cout << \_\_FILE\_\_ << ":" << line << "\n"; }();  [ = ]() { std::cout << \_\_FILE\_\_ << ":" << line << "\n"; }();  [ line ] { std::cout << \_\_FILE\_\_ << ":" << line << "\n"; }();  // error: line is not accessible in the lambda  []{ std::cout << \_\_FILE\_\_ << ":" << line; }(); | |
| capture | 03-05 |

Als je een lambda creeert in een klasse functie, dan wil je wellicht in de lambda gebruik maken van de eigenschappen van het object (de klasse functies en de klasse variabelen). Dit kan je doen door de this pointer (de impliciete pointer die naar het object wijst) te capturen, en de in de lambda expliciet te grbuiken.

|  |  |
| --- | --- |
| class capture\_this {  private:  int n;  public:  capture\_this():  n( \_\_LINE\_\_)  {  auto lambda = [ this ](){  this->print( this->n );  };  lambda();  }  void print( int x ){  std::cout << x << "\n";  }  }; | |
| capture this pointer | 05-06 |

Bij een capture by reference moet je goed opletten dat je niet een variabele by reference gebruikt die niet meer bestaat. Met een lambda kan dit voorkomen als je de lamda opslaat in een variabele die langer leeft dan de gereferencede variabele.

|  |  |
| --- | --- |
| using void\_void = std::function< void(void) >;  void assign\_lambda( void\_void & f ){  int n = \_\_LINE\_\_;  std::cout << \_\_LINE\_\_ << ":" << n << "\n";  // n is captured by reference by a lambda that will live longer  // than n. trouble on the way!  f = [ &n ](){  std::cout << \_\_LINE\_\_ << ":" << n << "\n";  };  }  void\_void f;  assign\_lambda( f );  // this function stores a lambda that uses the local variable n,  // which is no longer valid. Undefined behaviour!  f(); | |
| Lambda undefined behaviour | 03-07 |

## Lambda-driven programming

Met behulp van lambda’s die we opslaan in std::function<> objecten kunnen we de acties van het bewegende-bal programma vatten in objecten die we kunnen opslaan in een lijst. In de main loop lopen we dan de items van die lijst door, zonder ons er om te bekommeren wat daar precies in staat. Dit kan bij een grotere applicatie heel handig zijn, want we kunnen dan (met een iets andere opzet) die acties definieren bij de objecten waar ze betrekking op hebben, in plaats van in de main. De main ziet er dan (afgezien van de acties en de het scherm object) prettig kaal uit: er is eigenlijk niets meer over dat specifiek is voor de applicatie.

|  |  |
| --- | --- |
| int main( int argc, char \*argv[] ){  sf::RenderWindow window{ sf::VideoMode{ 640, 480 }, "SFML window" };  ball my\_ball{ sf::Vector2f{ 320.0, 240.0 } };  action actions[] = { . . . };  while (window.isOpen()) {  for( auto & action : actions ){  action();  }  window.clear();  my\_ball.draw( window );  window.display();  sf::sleep( sf::milliseconds( 20 ));  sf::Event event;  while( window.pollEvent(event) ){  if( event.type == sf::Event::Closed ){  window.close();  }  }  }  std::cout << "Terminating application\n";  return 0;  } | |
| Lambda-driven main | 03-08 |

De acties bestaan telkens uit twee delen: een conditie (bv. de cursor-links knop is ingedrukt) en wat er moet gebeuren als die conditie waar is. Beiden kunnen we als lambda schrijven, en een action object kan die twee lambda’s opslaan als std::function<> objecten. De constructor krijgt die twee lambda’s mee, en slaat ze op. De operator() check of aan de conditie voldaan is, zo ja dan wordt het werk uitgevoerd.

|  |  |
| --- | --- |
| class action {  private:  std::function< bool() > condition;  std::function< void() > work;  public:  action(  std::function< bool() > condition,  std::function< void() > work  ) :  condition( condition ),  work( work )  {}  void operator()(){  if( condition() ){  work();  }  }  }; | |
| Condition & work class | 03-08 |

De constructor die we nu hebben vraagt als conditie een lambda die een bool terug geeft. Dat is heel algemeen, maar het zou makkelijk zijn als we als conditie kunnen opgeven ‘deze knop is ingedrukt’, waarbij knop dan een toetsenbord toets kan zijn of een muis knop. Dat is eenvoudig te realizeren door twee constructors toe te voegen, die als eerste parameter een toets van het betreffende type meekrijgt. Die constructor maakt dan zelf de lambda aan die de test uitvoert door de betreffende SFML functie aan te roepen.

|  |  |
| --- | --- |
| action(  sf::Keyboard::Key key,  std::function< void() > work  ) :  condition(  [ key ]()->bool { return sf::Keyboard::isKeyPressed( key ); }  ),  work(work)  {}  action(  sf::Mouse::Button button,  std::function< void() > work  ) :  condition(  [ button ]()->bool { return sf::Mouse::isButtonPressed( button );  ),  work(work)  {} | |
| Extra constructors | 03-08 |

Nu kunnen we de lijst met acties schrijven.

|  |  |
| --- | --- |
| action actions[] = {  action( sf::Keyboard::Left, [&]{ my\_ball.move( sf::Vector2f( -1.0, 0.0 )); }),  action( sf::Keyboard::Right, [&]{ my\_ball.move( sf::Vector2f( +1.0, 0.0 )); }),  action( sf::Keyboard::Up, [&]{ my\_ball.move( sf::Vector2f( 0.0, -1.0 )); }),  action( sf::Keyboard::Down, [&]{ my\_ball.move( sf::Vector2f( 0.0, +1.0 )); }),  action( sf::Mouse::Left, [&]{ my\_ball.jump( sf::Mouse::getPosition( window )); })  }; | |
| actions | 03-08 |

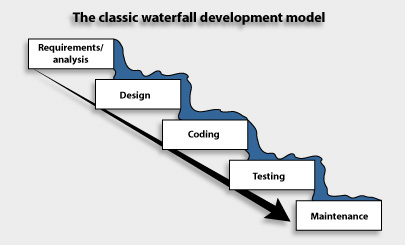
# Agile versus Waterfall



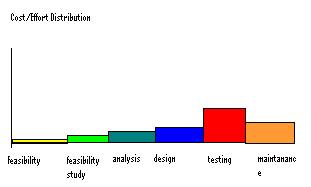
De ontwikkeling van een product doorloopt altijd een aantal stappen. De namen die aan zulke stappen worden gegeven variëren (en de naam duid soms op de activiteit, en soms op het eindproduct), maar een voorbeeld is

* Plan (het initiële idee)
* Architectuur (hoe gaan we het aanpakken)
* Realisatie (coderen)
* Verificatie (testen)
* Uitrollen (produceren, verkopen, gebruiken)

Iedere stap levert iets op. Het resultaat van de eerste stap kan bv. een memo zijn over een nieuw product waar beslist veel winst mee gemaakt kan worden. Een naïeve voorstelling van het proces is dat na het opleveren van zo’n product de stap is afgesloten, het eindresultaat ‘over de muur gegooid’ wordt, en de volgende stap aan het werk gaat. Dit wordt uitgebeeld in het ’waterval’ model: het water is het resultaat van iedere stap, en zoals het water betaamd stroomt dit van boven naar beneden.



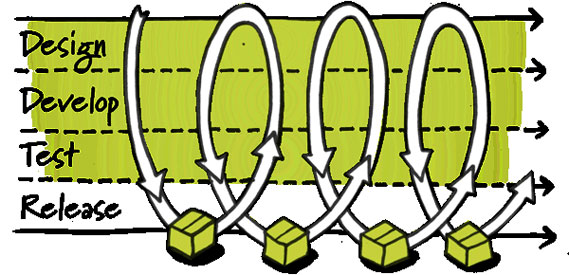
Dit model van een ontwikkelproces veronderstelt dat het resultaat van iedere stap perfect is (dus niet gewijzigd hoeft te worden). Dit is nogal naïef, zeker als je bedenkt dat iedere stap onder tijdsdruk staat om zijn resultaat af te leveren, en dat problemen in dat resultaat waarschijnlijk pas in volgende stappen naar boven komen. Een gevolg daarvan is dat er in latere stappen altijd meer problemen en vertragingen optreden dan er vooraf zijn ingeschat. Vooral de test fase is berucht.



Een eenvoudige aanpassing van het waterval model is het toevoegen van feedback. In iedere stap kunnen er inzichten ontstaan die consequenties (zouden moeten) hebben voor het werk in de voorgaande stap. Sterker, die inzichten kunnen consequenties hebben voor alle voorgaande stappen!

|  |  |
| --- | --- |
|  |  |

Het is mooi dat we kennis van later in het proces terugvoeren naar eerdere stappen, maar dit is wel erg kostbaar, omdat er dan al (veel) werk is gedaan dat afhankelijk was van die eerdere stap. In een ideale wereld komt dit natuurlijk maar zelden voor, maar helaas worden producten door mensen gemaakt, en mensen maken nu eenmaal fouten. Dat geldt niet alleen voor de mensen die het product maken, maar ook voor de mensen die bedenken welk product er gemaakt moet worden. Om dit probleem aan te pakken zijn er twee heel verschillende manieren van werken ontstaan.

[](https://www.google.nl/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0CAcQjRxqFQoTCKb-rPG_9scCFYM3FAodF9kMtQ&url=https://www.linkedin.com/pulse/essential-resources-services-technologies-your-startup-jason-oh&psig=AFQjCNEuE4CzX3oZ41F81kAxbgRUqHxppQ&ust=1442319168631758)

De ‘agile’ methoden splitsen een project op in heel veel deelprojectjes, die ieder een concreet, nuttig en bovenal aantoonbaar correct (deel) resultaat opleveren. Een voorbeeld hiervan is scrum, waarbij wordt gewerkt in sprints van hooguit een paar weken (maar liever 1 week). Iedere sprint begint met de beslissing wat er gebouwd gaat worden, en wordt afgesloten met het aan (of liever: door) de gebruiker demonsteren van het (geteste!) eindresultaat. Hierdoor wordt het gevaar van voortbouwen op een fout vermeden: een fout komt snel aan het licht, tijdens het testen (wat een verplicht onderdeel is van een sprint) of (als het een misverstand was tussen de gebruiker en de ontwikkelaars) bij het demonstreren. Zo’n ‘agile’ methode is vooral geschikt als

* Het project is opsplitsbaar in (onafhankelijke) delen.
* Die delen onafhankelijk ontwikkeld, getest en gedemonstreerd kunnen worden.
* Een demonstratie een goede test is
* Er een gebruiker aanwezig is die actief mee kan werken.
* De resultaten van een sprint nuttige input kunnen leveren voor volgende sprints (bv. doordat de eisen van de gebruiker bijgesteld zijn).
* Er eisen zijn die pas aan de hand van prototypes of testen kunnen worden geformuleerd
* Het ontwikkelteam en de gebruikers een sterk gemeenschappelijk belang hebben



Scrum en andere ‘agile’ methoden proberen de gevolgen van een fout te beperken door zo min mogelijk werk te doen dat afhankelijk is van een nog niet correct (door de klant geaccepteerd) stuk werk. De andere (bijna tegenovergestelde) aanpak van het probleem is te proberen fouten in een vroeg stadium op te sporen en te corrigeren. Daarbij is het belangrijk dat de mensen die die volgende stappen moeten uitvoeren (en dus de ‘shit’ van een eerdere stap over zich heen krijgen) betrokken worden bij zo’n eerdere stap. Ze kunnen natuurlijk niet al het werk van zo’n eerdere stap doen, maar ze kunnen wel meebeslissen over het eindresultaat van zo’n stap.

Zo’n ‘heavyweight’ ontwikkelmethode, waarbij er veel moeite wordt gedaan om het resultaat van iedere stap correct te krijgen, is op zijn plaats als

* Een project niet goed gesplitst kan worden in ‘features’ voor de eindgebruiker, maar wel in lagen die op elkaar bouwen.
* Het lastig is om het (eind) product (vaak) te testen.
* Er geen gebruiker beschikbaar is die kan en wil deelnemen in het project.
* Een demonstratie niet een geschikte manier is om het product te testen en de features te tonen.
* De eisen vanaf het begin vast staan.
* Er bij de ontwikkeling meerdere partijen betrokken zijn die verschillende belangen hebben.

Een applicatie die geschikt is om met een ‘agile’proces ontwikkeld te worden heeft vaak een overwegend verticaal gelaagde structuur: er zijn een aantal onafhankelijke ‘features’ die ieder iets nuttigs doen voor de eindgebruiker, bv. de verschillende soorten objecten die in een teken programma getekend kunnen worden:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Plaats een cirkel | Plaats een rechthoek | Plaats tekst | Teken met een pen | Teken met een kwast | Teken met een spuitbus |
| Scherm afhandeling | | | | | |

In een applicatie met een overwegend horizontale zijn er een aantal lagen die ieder gebruik maken van de lagere lagen, waarbij de functionaliteit voor de gebruiker wordt geleverd door de bovenste laag. Zo’n systeem leent zich beter voor een ‘heavyweight’ ontwikkel methode.

|  |  |  |
| --- | --- | --- |
| Dir | Copy | Delete |
| File system | | |
| Device driver(s) | | |
| Geheugen beheer | | |

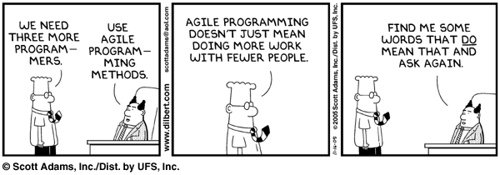
Applicaties die iets doen voor een mens (editor, compiler, tekenprogramma, browser, …) zijn vaak van nature vertikaal. Applicaties die hardware besturen (chemische fabriek, rijdende auto, autopiloot van een vliegtuig) zijn vaker horizontaal van aard.

Een andere manier om er tegen aan te kijken is:

* Als je een onduidelijke specificatie hebt en er is een klant is die je kan helpen bij die scherper te krijgen dan is een agile metode geschikt.
* Als je een in steen gebeitelde specificatie hebt en de klant kan of wil daar niets aan toevoegen dan is een waterval-achtige method geschikt.
* Als je een duidelijke specificatie hebt en een klant die wil meewerken: wees gelukkig, dit wordt een fijn project, en bijna alle methoden zijn geschikt.
* Als je geen duidelijke specificatie hebt en ook geen klant die je kan helpen: het maakt niet uit welke method je kiest, je project is gedoemd.



Overigen hoeft een project niet voor 1 ontwikkelmethode te kiezen: vaak leent het ene onderdeel van een project zich voor een waterval-achtige methode (requirements liggen keihard vast, de gebruiker stelt zich formeel op) en een ander deel zich meer voor een agile methode (requirements kunnen nog varieren, de gebruiker heeft behoefte aan en is bereid tot overleg).



# De heap



|  |
| --- |
| behandelde begrippen en technieken:   * Heap, new * new, new[], delete, delete[] * Garbage collection * Regel van drie * Eigenaarschap * Smart pointers: unique\_ptr<>, shared\_ptr<> |

## New en delete

Ieder object in een C++ applicatie heeft een bepaalde levensduur:

* Een globale variabele ontstaat voordat de main() begint, en wordt opgeruimd nadat de main() eindigt.
* Een lokale static variabele ontstaat op het moment dat zijn functie voor het eerst uitgevoerd wordt, en wordt eveneens opgeruimd nadat de main() eindigt.
* Parameters ontstaan net voordat de functie wordt uitgevoerd, en worden opgeruimd nadat de functie is uitgevoerd.
* Lokale variabelen ontstaan op de plek van hun definitie, en worden opgeruimd als het blok eindigt.
* Een tijdelijke object (het resultaat van een functie aanroep) ontstaat op het moment dat het nodig is en wordt opgeruimd bij het eerstvolgende ‘sequence point’, dat is meestal het einde van het statement waarin het tijdelijke object ontstond.

In al deze gevallen heeft de programmeur geen omkijken naar de levensduur van de objecten, en er ook geen invloed er op. Wel moet de programmeur opletten om geen objecten te gebruiken nadat ze opgeruimd zijn (dat is undefined behaviour).

Op het moment dat een object wordt aangemaakt moet je aangeven wat het exacte type is, en dat betekent voor bv. een string ook hoe groot die string (maximaal) kan zijn. Dit is vervelend als je een object wil aanmaken dat een hoeveelheid gegevens moet opbergen die op dat moment nog niet bekend is, bv. omdat het van externe dingen afhangt (invoer van de gebruiker, inhoud van een file, etc). Een oplossing hiervoor is om de gegevens niet in je object zelf op te slaan maar in een ander object, dat je op de heap allokeert. In je eigenlijke object kan je dan een pointer naar dit heap object opslaan. Om de hoeveelheid gegevens die je op deze wijze kan opslaan flexibel te houden kan je twee technieken gebruiken:

* Pointer naar array: je kan op de heap in plaats van een enkel object een array van objecten allokeren, met een tijdens het runnen bepaald aantal elementen. Mocht dit aantal niet genoeg zijn, dan kan je een nieuw array op de heap allokeren dan groter is, en alle oude elementen daarheen copieren. Dit lijkt erg (tijd-) inefficiënt, maar als je zodra het oude array niet groot genoeg is het nieuwe array bv. 2x zo groot maakt, dan komt dit over kopiëren gemiddeld niet vaak voor.[[12]](#footnote-12)
* Recursieve datastructuur: je kan in ieder object een of meer pointers zetten die naar volgende objecten wijzen (of nullptr zijn als er geen volgende objecten zijn). Zo kan je net zo veel objecten alloceren en met pointers aan elkaar verbinden als je nodig hebt.

De recursieve datastructuur vermijdt het kopiëren dat bij de array versie nodig is, maar allokeert wel veel meer (kleinere) objecten, die in de meeste gevallen willekeurig over het geheugen verspreid zijn. Dit vergt meer CPU tijd voor het ‘managen’ van de heap, en is ongunstig voor de ‘lokaliteit’ van je data en dus voor de performance van je cache. Daardoor is het op een systeem met genoeg geheugen en een hiërarchie van caches in de meeste gevallen beter om een array-gebaseerde opslag te gebruiken dan een die gebaseerd is op een recursieve datastructuur. Voor een systeem met weinig geheugen en geen caches geld eeerder het omgekeerde.

In C is er voor het allokeren van heap geheugen de functie malloc(). Die geef je het aantal bytes mee dat je wilt hebben, en je krijgt een void pointer terug naar een stuk geheugen (of NULL als er op de heap geen ruimte meer is). Als je het geheugen niet meer nodig hebt roep je free() aan met de pointer.

|  |
| --- |
| struct person {  ...  };  person \*p = malloc(sizeof(person));  // gebruik \*p  free( p ); |
| Allocate and free memory in C |

Het geheugen gebruiken nadat je het hebt vrijgegeven, of meer gebruiken dan je geallokeerd hebt, of het twee keer vrijgeven, zijn allemaal ‘undefined behaviour’. In C ben je er zelf voor verantwoordelijk dat je het juiste aantal bytes reserveert, de compiler zal je daar niet bij helpen, maar je kan wel sizeof() gebruiken om de kans op fouten te verminderen. De malloc aanroep geeft een void\* terug, en die kan je (in C) assignen aan ieder type pointer. (Dus helaas ook aan een verkeerd type.)

In C++ is er new voor het allokeren van een enkel object en new[] voor het allokeren van een array van objecten. Als je een enkel object allokeert geeft je meteen ook aan wat het type is en wat de constructor parameters voor dat object zijn. De compiler bepaalt voor je hoeveel geheugen er nodig is (dat kan je dus niet fout doen), en zorgt ervoor dat de constructor wordt aangeroepen. Je krijgt een pointer terug naar de juiste klasse, die hoef je dus niet te casten en je kan alleen assignen aan het juiste soort pointer. Net als bij C krijg je een nullptr (dat is de C++ naam voor NULL) terug als er niet genoeg heap geheugen meer beschikbaar is. Bij het allokeren van een array geef je aan hoeveel array elementen je wilt hebben en van wat voor type, en de compiler zorgt ervoor dat die elementen ieder geinitializeerd worden door aanroepen van de default constructor. Als je het geheugen niet meer nodig hebt dan roep je delete (voor een enkel object) of delete[] (voor en array van objecten) aan om het geheugen weer vrij te geven. De compiler zorgt er voor dat, voordat het geheugen weer aan de heap wordt toegevoegd, de destructor(s) worden aangeroepen.

|  |
| --- |
| person \*p2 = new person();  // gebruik \*p2  delete p2;  person \*p3 = new person[ 20 ];  // gebruik \*p3  delete[] p3; |
| Allocate and free memory in C++ |

Net als bij C is het undefined behaviour om meer array elementen te gebruiken dan je gealloceerd hebt, of het te gebruiken nadat je het hebt vrij gegeven. Daarnaast is het ook undefined behaviour als je allokeert met new en vrijgeeft met delete[] of andersom, en als je allokeert met malloc en vrijgeeft met delete (of delete[]) of andersom. Maar op veel systemen maakt het in de praktijk helaas[[13]](#footnote-13) niet uit omdat zowel new als new[] gewoon malloc aanroepen.

## Garbage collection

Als je het geheugen niet meer nodig hebt maar ook niet vrij geeft dan zal het systeem op den duur door z’n geheugen heen raken: het heeft dan een ‘memory leak’. Hoe erg dit is hangt af van hoe je de applicatie gebruikt: bij een applicatie die runt en snel klaar is zoals een compiler hoeft dit geen probleem te zijn, maar voor bv. een Operating System of een web server betekent het dat het systeem op den duur vast loopt en opnieuw opgestart moet worden. Dit heeft er toe geleid dat talen als Java en Python het managen van geheugen helemaal op zich nemen: het run-time systeem dat bij de taal hoort bevat een “garbage collector” die geheugen dat niet meer wordt gebruikt automatisch vrijgeeft voor hergebruik. Het voordeel hiervan is dat de programmeur er geen omkijken naar heeft en er ook geen fouten mee kan maken. Nadelen zijn

* Garbage Collection kost tijd, en maakt daardoor de run-time performance van het systeem op korte termijn minder voorspelbaar;[[14]](#footnote-14)
* Het moment waarop een object wordt vrijgegeven wordt bepaald door de garbage collector, niet door de programmeur. In dergelijke talen heeft een object doorgaans geen destructor, wat het lastiger maakt om andere resources dan geheugen effectief te managen.

Voor een gemiddelde desktop applicatie wegen de voordelen meestal zwaarder dan de nadelen. Voor een real-time applicatie wegen de nadelen meestal zwaarder. In sommige gevallen worden de sterke punten van beide benaderingen gecombineerd: het is niet ongewoon om voor een spel de performance-kritische delen in C of C++ (soms zelfs in assembler) te schrijven, en het (veel grotere) deel dat niet tijd-kritisch is in een gemangde taal (bv. Java of Python).

Bij het allokeren en/of vrijgeven van een stuk geheugen moet de library code die de heap beheert de administratie van het vrije geheugen bijwerken. Als er al vrije blokken zijn (die gescheiden worden door blokken die wel in gebruik zijn) en er wordt een blok vrijgegeven, dan wordt (meestal tijdens het vrijgeven) de lijst langsgelopen om te kijken of het nieuwe blok met oudere blokken gecombineerd kan worden tot een groter blok. Hoeveel tijd dit kost is lastig te voorspellen omdat het af hangt van de toestand van de administratie, met andere woorden: van het verleden. Om deze reden wordt het gebruik van de heap in real-time systemen vaak vermeden, in ieder geval tijdens het runnen: soms wordt nog wel heap geheugen geallokeerd tijdens het opstarten.

## De regel van drie

Een manier om met geheugen allocatie om te gaan is dat een klasse expliciet zorgt dat alle dingen die hij allokeert ook door hem worden vrijgegeven. Neem als voorbeeld een klasse die de naam van een persoon opslaat in een geallokeerd char array. Het array wordt in de constructor geallokeerd, en in de destructor weer vrij gegeven. Het name object is de eigenaar van het geallokeerde geheugen, en is dus verantwoordelijk voor het allokeren en deallokeren. Als je voor alle geallokeerde dingen op deze wijze een eigenaar kan aanwijzen is het geheugen management probleem (geheugen lekken, gebruik van al vrijgegeven geheugen) in principe oplgelost.

|  |
| --- |
| class name {  private:  char \* p;  public:  name( const char \* s ){  p = new char[ strlen( s ) + 1 ];  strcpy\_s( p, strlen( s ), s );  }  ~name(){  delete[] p;  }  }; |

Er zijn nog wel wat losse eindjes. De name klasse heeft geen copy constructor. In dat geval zal de compiler een copy constructor aanmaken. Die copieert domweg alle attributen (gebruikmakend van hun copy constructors). Voor een pointer houdt dit in dat er een copie wordt gemaakt van de pointer. In de volgende code is die copy constructor voor de duidelijkheid explciet uitgeschreven. Als we deze copy constructor gebruiken dan krijgen we twee variabelen, wiens pointers naar het zelfde op de heap geallokeerde namen wijzen. Dat wordt interessant als we de naam van de ene varanderen: de andere veranderd dan mee! En het wordt helemaal chaos als de destructors van de beide variabelen worden aangeroepen: hetzelfde heap object wordt dan twee keer vrij gegeven, wat resulteert in het gevreesde Undefined Behaviour, in dit geval (gelukkig) een foutmelding.

|  |
| --- |
| class name {  public:  char \* p;  name( const char \* s ){  p = new char[ strlen( s ) + 1];  strcpy( p, s );  }  // would be generated by the compiler  name( const name & rhs ):  p{ rhs.p }  {}  ~name(){  delete[] p;  }  };  int main() {  name w( "Wouter" );  name q( w );  std::cout << w.p << " " << q.p << "\n";  w.p[ 0 ] = 'J';  std::cout << w.p << " " << q.p << "\n";  return 0;  } |

Een correcte copy constructor moet in zijn eigen object ruimte reserveren voor de naam, en dan de letters van de naam van de parameter daarheen copieren.

|  |
| --- |
| name( const name & rhs ){  p = new char[ strlen( rhs.p ) + 1 ];  strcpy( p, rhs.p );  } |

Dat is eigenlijk het zelfde werk dat de gewone constructor al doet. Om in een constructor een andere constructor (een deel van) het werk te laten doen kan je gebruik maken van constructor ‘delegation’: je roept dan in de inlitializatie lijst van een constructor een andere constructor aan. Daarmee is de copy constructor teruggebracht tot een minimum.

|  |
| --- |
| name( const name & rhs ): name( rhs.p ){} |

Behalve de copy constructor maakt de compiler, als je die zelf niet schrijft, ook automatisch een assignment operator aan. Deze assignment operator assignt (copieert) de attributen, gebruik makend van hun assignment operatoren. Voor een pointer betekent dit (net als bij de constructor) dat de pointer wordt gecopieerd, zoals in de volgende code expliciet is uitgeschreven. Dit levert voor de name klasse een dubbel probleem op: na een assignment zijn er, net als na een copy, twee variabelen wiens pointers naar de zelfde naam wijzen (met alle problemen die bij de copy constructor zijn genoemd), en het object dat door de bestemming van de assignment was geallokeerd is voorgoed verloren.

|  |
| --- |
| // would be generated by the compiler  name & operator=( const name & rhs ){  p = rhs.p;  return \*this;  } |

Een naieve oplossing is dat we in de assignment operator eerst ons eigen geallokeerde geheugen vrij geven, en daarna (net als in de constructors) voldoende geheugen allokeren voor de nieuwe naam, en die copieren.

|  |
| --- |
| // naieve assignment  name & operator=(const name & rhs) {  delete[] p;  p = new char[strlen(rhs.p) + 1];  strcpy(p, rhs.p);  return \*this;  } |

Dit gaat perfect, behalve in het geval dat we een dergelijke variabele aan zichzelf toekennen: dan zijn p en rhs.p hetzelfde, dus na de ”delete[] p” regel kan je ook rhs.p niet meer gebruiken. Dit kan je op twee manieren oplossen:

* Bewaar een copie van de p pointer, en geef het geheugen waar die naar wijst pas op het einde vrij;
* Check of de rhs en het object dezelfde zijn. Als dat zo is dan hoeft er niets te gebeuren.

Beide methoden voldoen.

|  |
| --- |
| // save-old-pointer assignemnt  name & operator=(const name & rhs) {  auto old\_p = p;  auto rhs\_p = rhs.p;  p = new char[strlen(rhs\_p) + 1];  strcpy(p, rhs\_p);  delete[] old\_p;  return \*this;  } |
| // do-nothing assignment in case of self-assignment  name & operator=(const name & rhs) {  if (&rhs != this) {  delete[] p;  p = new char[strlen(rhs.p) + 1];  strcpy(p, rhs.p);  }  return \*this;  } |

Deze manier om met het allokeren van heap geheugen om te gaan (door het te verbergen in een klasse, en die klasse het werk te laten doen) wordt wel de regel van 3 genoemd: Als je een klasse schrijft met een pointer er in die moet wijzen naar heap geheugen, dan moet je vrijwel altijd 3 functies schrijven:

* een of meer constructors waarin het geheugen wordt geallokeerd (denk er aan dat de compiler een default constructor maakt als er helemaal geen constructors zijn, en een coipy constructor als er geen copy constructor is);
* een assignment operator waarin het oude geheugen wordt vrijgegeven en het nieuwe geheugen wordt geallokeerd (denk er aan dat de compiler een assignment operator maakt als er geen geschreven is);
* een destructor waarin het geheugen wordt vrijgegeven.

## Eigenaarschap

Het werken met pointers en heap objecten lijkt misschien niet zo moeilijk: als je datgene wat je allokeert zelf weer vrij geeft komt alles goed. Dat is zo, maar het venijn zit in de situaties waar degene die het object heeft geallokeert niet degene is die het ook weer vrijgeeft, bv. als je een functie hebt die objecten aanmaakt naar aanleiding van een invoer file. Die functie levert (een pointer naar) het object op. Het is aan zijn aanroeper om dat object ooit weer vrij te geven. Als het object wordt opgeslagen in een datastructuur die door meedere delen van de applicatie wordt gebruikt dan wordt het lastig helder te houden wie er verantwoordelijk is voor het vrijgeven.

De kern van het problem is eignaarschap: wie is de eigenaar van het geallokeerde geheugen, en dus verantwoordelijk voor het vrijgeven. Bij code die op deze manier werkt (bv. een functie die een pointer naar geallokeerd geheugen teruggeeft) moet heel duidelijk in de documetatie staan dat het eigenaarschap van dat geheugen wordt overgedragen naar de aanroeper van de functie.

Nog beter is het een ‘smart pointer’ te gebruiken.

## Smart pointers

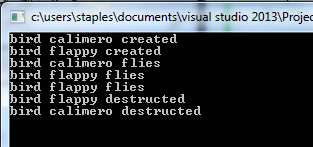
Een variabele wordt aan het einde van zijn levensduur opgeruimd (het object wordt gedestruct, en de ruimte wordt vrijgeggeven). Maar dit geldt alleen voor de variabele zelf, als het een pointer is geldt dit niet voor hetgene waar de pointer naar wijst. In de volgende code wordt aan het einde van de functie test() calimero wel opgeruimd, want dat is een variabele, maar flappy niet, want die is op de heap aangemaakt, en er staat nergens een ‘delete p;’ statement.

|  |
| --- |
| class bird{  public:  bird( const char \*name ): name( name ){  std::cout << "bird " << name << " created\n";  }  ~bird(){  std::cout << "bird " << name << " destructed\n";  }  void fly(){  std::cout << "bird " << name << " flies\n";  }    private:  std::string name;  };  void test(){  bird c( "calimero" );  bird \* p = new bird( "flappy" );  c.fly();  p->fly();  } |

Als wij de enige ‘eigenaar’ zijn van het object waar de pointer naar wijst zou het heel handig zijn als het geregeld kon worden dat dit wel automatisch wel gebeurt. Dat kan door in plaats van een pointer een pointer-achtig ding te maken: een object dat zich gedraagt als een pointer, maar in zijn destructor ook dat waar hij naar wijst delete.

Wat voor operaties hebben we nodig van zo’n ‘smart pointer’? Het moet een template zijn, want we willen zulke pointers voor ieder basistype kunnen maken. We moeten er een kunnen aanmaken, dus construeren, op grond van een gewone pointer. In zijn destructor moet hij die pointer deleten. Verder is de belangrijkste operatie van een pointer het dereferencen, en dat levert het object op waar naar wordt gewezen. Dat moet de operator\* dus ook doen. In C++ is de operator-> (de combinatie van dereference en het benaderen van een onderdeel) een aparte operator, die gek genoeg niet het object maar een pointer naar het object moet teruggegeven. Die operator moeten we dus ook hebben.

|  |
| --- |
| template<typenameT>  classremover {  public:  remover( T\* p ):  p( p ){}  ~remover(){  delete p;  }  T & operator\*() const {  return \*p;  }  T \* operator->() const {  return p;  }  private:  T \*p;  }; |

Hiermee hebben we een werkende ‘smart pointer’ die het object waar hij naar wijst weggooit op het moment dat hij zelf wordt weggegooid.

|  |
| --- |
| void test(){  bird c( "calimero" );  remover<bird> p( newbird( "flappy" ) );  c.fly();  (\*p).fly();  p->fly();  } |

Er zijn nog wel wat losse eindjes: bv. de assignment en de copy constructor moeten zorgen dat er maar 1 exemplaar van de pointer bestaat. Een dergelijke smart pointer is beschikbaar in de library als std::unique\_ptr<>. De naam verwijst er naar dat er altijd maar zo’n 1 pointer naar het heap object zal zijn.[[15]](#footnote-15) Als die pointer ge-delete wordt dan wordt dus ook het heap object ge-delete.

|  |
| --- |
| #include<memory>  void test(){  bird c( "calimero" );  std::unique\_ptr<bird> p( newbird( "flappy" ) );  c.fly();  (\*p).fly();  p->fly();  } |

In de vorige code wordt bij het allokeren het type (‘bird’) twee keer genoemd. Als dat een ingewikkelde type expressie is dan is het handiger om de (template) functie std::make\_unique te gebruiken. Een andere reden is dat je dan het expliciet gebruik van new in je code kan vermijden.

|  |
| --- |
| #include<memory>  auto p = std::make\_unique<bird>( "flappy" ); |

Een andere smart pointer die vaak handig is heet std::shared\_ptr<>. Een shard\_ptr kan je assignen, en er kunnen meerdere van deze pointers naar het zelfde object wijzen. De shared\_ptr’s die naar het zelfde object wijzen houden bij met z’n hoevelen ze zijn, en als de laatste ophoudt met naar dat object te wijzen (omdat hij zelf gedelete wordt, of omdat hij naar een ander object gaat wijzen, of omdat hij gereset wordt en dus naar niets gaat wijzen), dan delete hij het object.

|  |
| --- |
| #include<memory>  void test(){  std::shared\_ptr<bird> p1( newbird( "flappy" ));  std::shared\_ptr<bird> p2;  p2 = p1;  p1.reset();  p2.reset();  } |

Met een unique\_ptr voor kunnen we de name klasse iets eenvoudiger maken. De destructor is overbodig, want de compiler-genenereerde destructor roept de destructor van de unqiue pointer aan, en die zorgt er voor dat het geheugen wordt vrijgegeven. De gewone constructor, copy constructor en assignment operator zijn nog wel nodig, maar in de assignment hoef je het oude geeugen niet meer vrij te geen, want dit gebeurt automatisch als je de unique pointer naar eenm niew stuk geheugen laat wijzen.

|  |
| --- |
| class name {  private:  std::unique\_ptr< char[] > p;  public:  name(const char \* s) {  p = std::make\_unique< char[] >( strlen(s) + 1 );  strcpy( p.get(), s );  }  name(const name & rhs) : name( p.get() ) {}  name & operator=(const name & rhs) {  auto temp = new char[ strlen( rhs.p.get() ) + 1 ];  strcpy( temp, rhs.p.get() );  p.reset( temp );  return \*this;  }  // no destructor needed  }; |

# Exceptions



|  |
| --- |
| behandelde begrippen:   * Wat te doen bij een fout * Exceptions in C++ |

Een functie of methode heeft een doel. In verreweg de meeste gevallen zal hij dat doel bereiken en het resultaat daarvan aan zijn aanroeper ter beschikking stellen (of in het geval van een void fucntie: hij heeft iets gedaan, en dat was het doel). Er zijn echter gevallen waarin dat niet lukt. Dit komt meestal doordat een invoerwaarde (parameter, of soms iets globaals) een waarde heeft die niet geldig is (bv. delen door 0), of dat iets dat de functie nodig heeft om zijn werk te doen niet beschikbaar is (bv. de heap is vol).[[16]](#footnote-16) Als zoiets gebeurt, moet de code na de aanroep de mogelijkheid hebben iets anders te doen dan normaal, want de situatie is niet normaal.

Om dit mogelijk te maken de functie zelf iets anders doen dan hij normaal zou doen. Hij kan bv.

* Zo goed mogelijk doorgaan, en misschien een speciale waarde teruggeven. bv.
* Bij delen door 0: de waarde +oneindig, dat is een speciale floating-point waarde, een van de zogenaamde NaN’s, Not A Number;
* Bij een file die er niet is: een default file openen;
* Bij geheugen dat er niet meer is: nullptr teruggeven.
* Het programma stoppen met een foutmelding.
* Via een globale variabele laten weten dat er iets aan de hand is.
* Via een extra parameter aan zijn aanroeper laten weten wat er aan de hand is, zodat die kan beslissen wat er moet gebeuren.
* Niets doen: laat de zaak gewoon in het honderd lopen. Dit lijkt een belachelijke keuze (en dat is het natuurlijk ook), maar dit is maar al te vaak wat er feitelijk gebeurt.

De eerste drie alternatieven zijn het aantrekkelijk, omdat ze geen extra parameters en geen extra code in de aanroeper vereisen als die geen speciale actie wil ondernemen. Maar welke er gekozen moet worden (zo goed mogelijk doorgaan, of afbreken met een duidelijke foutmelding) is afhankelijk van de context. De pinger in de zwarte doos van een vliegtuig moet zo goed mogelijk doorgaan. Een programma dat gedebugd wordt of op een andere wijze een duidelijke user interface heeft (en niet te veel schade aanricht door te stoppen) kan beter een foutmelding geven. Maar de functie zelf weet niets van deze afwegingen, dus wat moet hij doen?

Een extra parameter gebruuiken is in zekere zin het beste, want het legt de beslissing wat er moet gebeuren waar het hoort, nl. buiten de functie. Het nadeel is echter dat nu alle functie aanroepen moeten worden uitgebreid met een parameter die aangeeft of de operatie geslaagd is (niet alleen de functies waarin ‘direct’ iets mis kan gaan, echt alle functies in de keten van aanroepen). En na de functie aanroep moet je eerst controleren of de functie wel geslaagd is, en zo niet dan moet je dit aan jouw aanroeper melden. Dat levert een explosie van extra code op, en hoe moet je dit doen als een functie aanroep ‘vermomd’ is als operator?

De globale errno variabele is een poging om fouiten te melden via een globale variabele: deze variabele is initieel 0, en krijgt een andere waarde als een functie (die dit mechanisme gebruikt) een fout ontdekt. Een dergelijke functie zal errno nooit terugzetten naar 0, dus de gebruiker kan na een reeks functie calls in een keer checken of alles goed is gegaan door errno met 0 te vergelijken. Dit mechanisme heeft een aantal nadelen (bv. dat na de eerste fout de acties gewoon verder gaan), en de manier waarop functies het gebruiken is inconsistent (bv. moet een functie een niet-nul waarde overschrijven met een nieuwe waarde of juist niet?).

Het C++ exception mechanisme probeert het beste van alle oplossingen te combineren. Stel je voor dat iedere functie aanroep kan naast de reguliere manier ook nog op een andere manier kan eindigen, en dan via een parameter laat weten wat er aan de hand was, maar zonder dat de gebruiker dit allemaal hoeft uit te schrijven. Op de plek waar geconstateerd wordt dat er een probleem is (bv. de te openen file bestaat niet) wordt via een throw statement een exception genenereerd. Dit beeindigt de executie van het huidge blok (dat komt ongeveer overeen met een { } paar en alles daartussen), en daarna het blok daarom heen, net zo lang tot er een blok gevonden wordt dat aangeeft dat de exceptie moet worden afgevangen dmv. een except claused die past bij de waarde die aan de throw werd meegegeven.

In het volgende voorbeeld moet de functie divide() een integer deling uitvoeren. Alvorens dit te doen wordt er eerst gecheckt of het deeltal 0 is, en zo ja dan wordt de exceptie divide\_by\_zero genenereerd door het throw statement. De functie calculate roept twee keer divide() aan, binnen een try{ } blok. Aan dit blok hangt een catch die de divide\_by\_zero exception opvangt en afhandeld. In dit geval door het resultaat 0x77, hier moet in de praktijk natuurlijk iets zinnigers staan.

|  |
| --- |
| class divide\_by\_zero : public std::exception { . . . };  int divide( int c, int d ){  if( d == 0 ){  throw divide\_by\_zero;  }  return c / d;  }  void calculate(){  try {  p =  divide( 3, q )  – divide( 9, ( b +5 ));  } catch ( const divide\_by\_zero &e ) {  p = 0x77;  }  . . .  } |

Een throw statement vermeldt de exception die generereerd wordt. Dit kan letterlijk iedere waarde zijn, maar de conventie is dat alle exceptie waarden afgeleid zijn van het type std::exception. Dit type heeft een virtuele what() methode die een std::string teruggeeft die aangeeft wat er fout is gegaan. Het throw statement roept de constructor van het aangegeven type aan, en kan daar parameters aan meegeven om te verduidelijken wat er aan de hand is. De macro’s \_\_FILE\_\_ en \_\_LINE\_\_ kunnen daarbij handig zijn: die worden door de compiler vertaald naar de huidige source file naam (een string) en regelnummer (een integer).

|  |
| --- |
| class divide\_by\_zero : public std::exception {  private:  std::string msg;  public:  divide\_by\_zero( const char \*file, int line ){  msg = ”divide by zero”  + ”at ” + std::string( file )  + ”:” + std::string( line );  }  const char \* what() override {  return msg;  }  };  int divide( int c, int d ){  . . .  throw divide\_by\_zero( \_\_FILE\_\_, \_\_LINE\_\_ );  }  void calculate(){  try {  . . . divide( . . . )  } catch ( const std::exception& e ) {  std::cerr << e.what();  }  } |

Achter een try { } blok komen de catch blokken. Ieder catch blok geeft aan wat voor type exceptie dat blok afhandelt. De match tussen de exceptie waarde en de verschillende catch blocken wordt gemaakt op de zelfde manier als kiezen tussen een reeks functies met de zelfde naam: de parameter moet passen, en een waarde van een afgeleide klasse (de exceptie) past op een catch parameter die daar een superklasse van is.

Het is ook mogelijk in 1 keer alle excepties op te vangen door als parameter ... op te geven, maar dan krijgt het catch blok geen toegang tot de exceptie waarde (dat zou niet kunnen, want het type is niet bekend).

|  |
| --- |
| try {  . . .  } catch( ... ){  std::cerr<< ”I have no idea what happened\n”;  exit();  } |

Een try { } blok kan een reeks catch blokken hebben. Deze worden een voor een geprobeerd, en de eerste die de exceptie als parameter accepteert wordt aangeroepen. De volgende catch blokken van dat try { } blok hebben dan geen functie meer in het opvangen van deze exceptie. Als geen van de catch blokken ‘past’ dan wordt de exceptie geescaleerd naar het volgende ‘omvattende’ try { } blok. Dit kan in een aanroepende functie zijn. Als er tot in de main geen passende catch gevonden wordt dan wordt het programma afgebroken met een foutmelding.

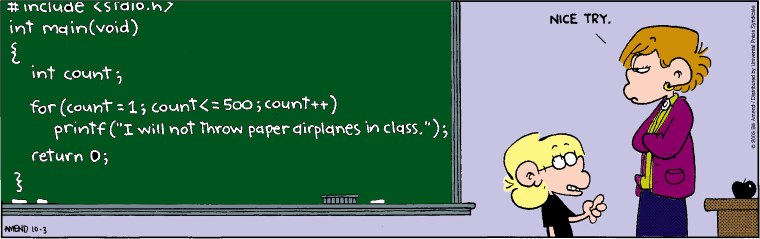
Als een blok code wordt veraten dan worden de destructors aangeroepen voor de lokale variabelen. Dit gebeurt ook als een blok wordt beeindigt doordat er een exception is geneneerd.

Het is duidelijk dat excepties gebruikt kunnen worden om situaties af te handelen die de normale werking van het programma onmogelijk maken. Volgens sommigen moet het gebruik van excepties beprekt blijven tot zulke situaties. De huidige implementatie van exceptions door de compilers sluit hier bij aan: er wordt geoptimalizeerd voor het geval dat er geen exceptions worden gegeneerd, ten koste van het geval dat er wel een exception wordt gegenereerd.

Het lastige aan deze opvatting is dat de code die de exceptie genereert niet kan weten of de situatie die hij voor zijn kiezen krijgt fataal is voor de aplicatie of dat het door een ‘hogere’ functie in de keten van aanroepen prima afgehandeld kan worden. Een heel andere opvatting is dat een exception gewoon een ander resultaat is van een functie, niet meer of minder ‘nromaal’ dan andere resultaten.

Voor het schrijven van high-perfromance en/of real-time code wordt het gebruik van exceptions vaak vermeden.

# For loop



|  |
| --- |
| behandelde begrippen:   * for( : ) loop * een eenvoudige pointer container |

Van C heeft C++ de standaard for loop overgenomen. In C++ 0x11 is daar de zgn. range-based for loop bij gekomen. De eenvoudigste manier om die te gebruiken is met een array.

|  |
| --- |
| const int array\_size = 12;  int array[ array\_size ];  for( auto x : array ){  som += x;  } |

De loop variabele x krijgt een voor een de waarden uit het array. Als je deze for loop wil gebruiken om de waarden in het array te wijzigen, dan moet je loop variabele geen integer maar een integer reference zijn. Die reference wijst dan een voor een naar alle array elementen.

|  |
| --- |
| const int array\_size = 12;  int array[ array\_size ];  for( auto & x : array ){  x = 0;  } |

Als de expressie na de : geen array is, dan probeert de compiler een vertaling die ongeveer hier op neer komt:

|  |
| --- |
| // for( x : object ){ . . . }  auto xp = object.begin()  while( xp != object.end() ){  x = \* xp;  . . .  ++xp;  } |

Om dit te laten slagen moet het object de methoden begin() en end() hebben, en die moeten iets teruggeven dat als een pointer gebruikt kan worden.[[17]](#footnote-17)

Stel dat we een reeks SFML objecten moeten aanmaken en op het scherm tonen. Dit kan natuurlijk door ze een voor een aan te maken, op te slaan in lokale variabelen, en later voor ieder van die variablelen de draw(screen) methode aan te roepen.

|  |
| --- |
| int main(){  . . .  wall w1{ . . . };  wall w2{ . . . };  wall w3{ . . . };  wall w4{ . . . };  . . .  w1.draw( window );  w2.draw( window );  w3.draw( window );  w4.draw( window ); |

Als we behalve tekenen nog iets anders met deze objecten moeten doen, bv. bewegen, dan moeten we iedere keer als we een object toevoegen, dit op 3 plekken doen: het aanmaken, het tekenen, en het bewegen. Dat wordt snel onhandelbaar.

Een alternatief is dat we de objecten op de heap aanmaken, en de pointers[[18]](#footnote-18) naar die objecten bewaren in een array. Als we array onderbrengen in een klasse die bijhoudt hoeveel array elementen al gevuld zijn, en de juiste begin() en end() methoden biedt, dan kunnen we door al onze objecten heen lopen met een for loop.

|  |
| --- |
| int main(){  . . .  wall\_pointer\_storage walls;  walls.add( new wall{ . . . } );  walls.add( new wall{ . . . } );  walls.add( new wall{ . . . } );  . . .  b.draw( window );  for( auto p : walls ){  p->draw( window );  } |

Dit kan vrij eenvoudig: begin() geeft een pointer naar het eerste array element terug, en end() een pointer naar het eerste nog niet gevulde array element. Aangezien de array elementen zelf al pointers zijn, moeten deze methoden dus pointers naar pointers teruggeven.

|  |
| --- |
| class wall\_pointer\_storage {  public:  void add( wall \* p ){  if( count < 100 ){  store[ count ] = p;  ++count;  }  }  wall \*\* begin(){ return store; }  wall \*\* end(){ return store + count; }  private:  wall \* store[ 100 ];  int count = 0;  }; |

Deze opslag is nu voor 100 wall pointers. We kunnen er een template van maken, waardoor zowel het opgeslagen type als het aantal door de gebruiker opgegeven kunnen worden.

|  |
| --- |
| template< typename T, int N >  class storage {  public:  void add( const T & p ){  if( count < N ){  store[ count ] = p;  ++count;  }  }  T \* begin(){ return store; }  T \* end(){ return store + count; }  const T \* begin() const { return store; }  const T \* end() const { return store + count; }  private:  T store[ N ];  int count = 0;  }; |

In deze versie is zijn nog begin() en end() methoden toegevoegd die kunnen werken met een const object. Wel geven die natuurlijk een const T pointer terug, zodat een for loop die van deze methoden gebruik maakt geen verandereingen in de array elementen kan aanbrengen.

Een dergelijke opslag wordt een container genoemd. De STL, een onderdeel van de stndaard C++ library, levert een groot aantal containers.

# Factory pattern



|  |
| --- |
| behandelde begrippen:   * Factory pattern * Virtual destructor |

Stel dat we een aantal objecten willen aanmaken op grond van een configuratie file. De voorbeeld file hieronder bevat voor ieder object 1 regel. De regel begint met het coördinaat van het object, dan komt het soort object, en wat daarna komt is afhankelijk van het soort object: voor een cirkel en rechthoek eerst de kleur, dan voor een cirkel de straal en voor een rechthoek de omvang, en voor een plaatje alleen de file.

|  |
| --- |
| (50,60) CIRCLE blue 10  (10,10) RECTANGLE red (30,40)  (110, 50) PICTURE bird.jpg |

Iedere regel van de configuratie file resulteert in het aanmaken van een object, maar iedere regel kan een ander type object aanmaken. Om een regel te lezen en om te zetten naar een object moeten we dus een functie schrijven die verschillende soorten objecten terug kan geven. In C++ kan dit niet direct, want het type van het resultaat van een functie ligt vast in de functie declaratie. De typen die onze functie moet kunnen teruggeven zijn weliswaar allemaal afgeleid van het basistype screen\_object, maar als we dat type zouden teruggeven dan krijgen we slicing: er is in dat type geen ruimte zijn voor de specifieke attributen (bv. place en size) van een van de echte typen (bv. ball bevat een place en size, waar geen ruimte voor is in screen\_object). Een reference teruggeven is ook niet practisch, want waar is dan het echte object? Het kan geen lokale variabele van de functie zijn, want die bestaat niet meer nadat de functie is geeindigt. Wat echter wel kan is dat onze functie een pointer naar het (op de heap) aangemaakte object teruggeeft. Dit kan in de declaratie van die functie een screen\_object pointer zijn, want we willen dat object toch alleen via de screen\_object interface gebruiken. Zo’n functie die, bv. op grond van zijn parameters, of in dit geval op grond van tekstuele input, verschillende soorten objecten kan aanmaken en teruggeven, wordt een ‘factory’ genoemd.

Als we op deze manier een reeks objecten hebben aangemaakt moeten we ze later ook weer kunnen vrijgeven. Daarbij is het van belang dat we ieder object door zijn eigen destructor laten opruimen, want alleen die destructor weet welke zaken er vrijgegeven moeten worden. Maar wij hebben van alle objecten alleen een pointer naar het object, en we weten van ieder object alleen dat het afgeleid is van screen\_object. De enige manier om te zorgen dat de juiste destructor wordt aangeroepen is dus te zorgen dat die destructor virtual is. Aangezien een object zelf nooit kan weten of het ooit op een dergelijk manier gebruikt gaat worden is het de regel dat iedere klasse die ooit op deze wijze als superclass gebruikt zal worden een virtual destructor moet hebben. Het alternatief is dat je de klasse final maaket, dan kan hij nooit als superklasse gebruikt worden.

Een factory functie moet op grond van de gegevens die hij binnen krijgt beslissen welk type object hij gaat aanmaken. In ons geval is dat op grond van het woord dat hij (na de positie) uit de invoer file leest. Als dit woord niet een van de geldige waarden heeft dan wordt er een passende exception gegenereerd die aangeeft wat de waarde van het woord is. (Als het woord leeg was dan waren we blijkbaar aan het einde van de file gekomen.) Als het woord wel herkend wordt dan worden de overige gegevens uit de file gelezen, en vervolgends wordt het creeeren van het object (natuurlijk) overgelaten aan de constructor van de desbetreffende klasse.

|  |
| --- |
| screen\_object \* screen\_object\_read( std::ifstream & input ){  sf::Vector2f position;  std::string name;  input >> position >> name;  if( name == "CIRCLE" ){  . . .  return new circle( . . . );  } else if( name == "RECTANGLE" ){  . . .  return new rectangle( . . . );  } else if( name == "PICTURE" ){  . . .  return new picture( . . . );  } else if( name == "" ){  throw end\_of\_file();  }  throw unknown\_shape( name );  } |

Voor het inlezen van een kleur kunnen we een operator>> definieren. In die operator kunnen we natuurlijk met een reeks if statements testen op de verschillende kleuren, maar een tabel met de kleur namen en waarden die we met een for loop doorlopen levert veel compactere code op.

|  |
| --- |
| std::ifstream & operator>>( std::ifstream & input, sf::Color & rhs ){  std::string s;  input >> s;  const struct { const char \* name; sf::Color color; } colors[]{  { "yellow", sf::Color::Yellow },  { "red", sf::Color::Red },  . . .  };  for( auto const & color : colors ){  if( color.name == s ){  rhs = color.color;  return input;  }  }  if( name == "" ){  throw end\_of\_file();  }  throw unknown\_color( s );  } |

Als de ingelezen kleur naam niet overeenkomt met een kleur in de lijst dan wordt wederom een passende excetptie gegenereerd. Deze is natuurlijk afgeleid van std::exception en het resultaat van de what() methode vermeldt het gevonden woord dat geen geldige kleur bleek te zijn.

|  |
| --- |
| class unknown\_color : public std::exception {  public:  unknown\_color( const std::string & name ):  s{ std::string{ "unknown colcor [" } + name + "]" }  {}  const char \* what() const override {  return s.c\_str();  }  private:  std::string s;  }; |

Het inlezen van een positie is iets ingewikkelder dan het lezen van het soort object of een kleur, omdat een positie uit 5 onderdelen bestaat. Na het inlezen wordt de input stream in een if statement gebruikt als boolean. De operator>> van de ingebouwde typen veroorzaken zelft geen exceptie als het inlezen mislukt, maar ze zetten de input stream in een toestand waarin de stream (als boolean gebruikt) false is. Hiermee kunnen we in zo’n geval dus zelf een exceptie genereren.

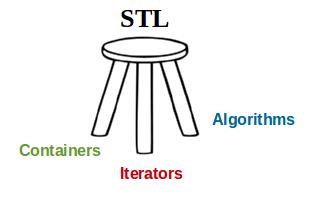
|  |
| --- |
| std::ifstream & operator>>( std::ifstream & input, sf::Vector2f & rhs ){  char c;  if( ! ( input >> c )){ throw end\_of\_file(); }  if( c != '(' ){ throw invalid\_position( c ); }  if( ! ( input >> rhs.x )){ . . . }  if( ! ( input >> c )){ . . . }  . . .  if( ! ( input >> rhs.y )){ . . . }  if( ! ( input >> c )){ . . . }  if( c != ')' ){ throw invalid\_position( c ); }  return input;  } |

De aanroeper van de factory function opent de juiste invoer file, en roept dan, binnen een try-catch blok, herhaaldelijk de factory function aan met die file totdat er een exception wordt opgevangen. Het eerste catch blok handelt een end\_of\_file exception af. Hiervoor is geen nadere actie nodig: de end\_of\_file exception is de manier waarop de for() lus eindigt. Executie gaat verder na de catch bloks, en komt dus meteen bij de laatste }. Merk op de de file niet expliciet wordt gesloten: dit gebeurt impliciet in de destructor van het std::ifstream object, die wordt aangeroepen omdat dit object aan het einde van zijn leven is gekomen als de laatste } wordt bereikt.

Het andere catch blok handelt alle overige (van std::exception afgeleide) exceptions op. Hier wordt een melding geprint, en dan wordt exit(0) aangeroepen dat het programma beëindigt.

|  |
| --- |
| {  std::ifstream input( "objects.txt" );  try {  for(;;){  screen\_objects.add( screen\_object\_read( input ));  }  } catch ( end\_of\_file ){  // do nothing  } catch ( std::exception & problem ){  std::cout << problem.what();  }  } |

# Standaard Template Library



|  |
| --- |
| behandelde begrippen:   * container, iterator, algorithme * for\_each() * lambda * std::array, std::vector, std::set, std::map |

Een for( auto x : a ) loop heeft van het object a twee waarden nodig om alle elementen van a af te lopen: een begin en een eind. Die waarden moeten zich zo’n beetje als een pointer gedragen. Zo’n waarde wordt een **iterator** genoemd, en het object a een **containe**r. De minimale eisen aan zo’n iterator zijn (in dit geval):

* Een iterator moet een operator\* hebben die iets terug geeft. Dat ‘iets’ is het type van de waarden die in de container zijn opgeslagen.
* Een iterator moet een operator++ hebben waarmee de iterator verplaatst kan worden naar het volgende element.
* Iterators moeten met elkaar vergeleken kunnen worden (operator==) om te weten wanneer we voorbij het laatste element zijn.
* Om een iterator (by value) door te kunnen geven aan een functie moet er natuurlijk een copy constructor beschikbaar zijn.

Met deze eigenschappen van een iterator kunnen we de elementen van de containter op de gewone volgorde (van de eerste tot de laatste) benaderen. We kunnen bv. een functie schrijven die als iterator een int \* accepteert en alle elementen van de container een gegeven waarde geeft, en een functie die ze uitprint.

|  |
| --- |
| void range\_fill\_int( int \* begin, int \* end, int n ){  for( int \* p = begin; p != end; p++ ){  \*p = n;  }  } |

|  |
| --- |
| void range\_print\_int( int \* begin, int \* end ){  for( int \* p = begin; p != end; p++ ){  std::cout << \*p <<" ";  }  std::cout <<"\n";  } |

|  |
| --- |
| int main(){  int a[ 10 ];  range\_fill\_int( a, a + 10, 12 );  range\_print\_int( a, a + 10 );  } |

Deze functies kunnen alleen met integer pointers werken. Het is eenvoudig om ze om te zetten in templates, zodat ze met ieder passend type kunnen werken.Dan kunnen we ook overschakelen op het type std::array, dat wel de methoden begin() en end() heeft, maar doe zijn niet noodzakelijk van het type int \*.

|  |
| --- |
| #include<array>  template< typename T, typename E>  void range\_fill( T begin, T end, E n ){  for( T p = begin; p != end; p++ ){  \*p = n;  }  }  template<typenam e T>  void range\_print( T begin, T end ){  for( T p = begin; p != end; p++ ){  std::cout << \*p <<" ";  }  std::cout <<"\n";  }  void main(){  std::array< int, 10 > a;  std::array< double, 8 > b;  range\_fill( a.begin(), a.end(), 12 );  range\_fill( b.begin(), b.end(), 3.14 );  range\_print( a.begin(), a.end() );  range\_print( b.begin(), b.end() );  . . .  } |

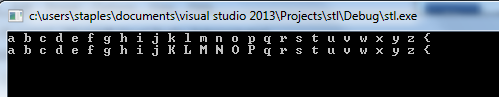
De voorgaande functies deden ieder iets bepaalds met de array elementen. We kunnen ook een functie maken die als parameter een functie krijgt en die aanroept (engels:applies) voor ieder array element.

We kunnen een template functie schrijven die een gegeven functie aanroept voor ieder element van de container. In dit geval geeft die functie de elementen opeenvolgende characters uit het alfabet. Daarmee hebben we **wat** we doen losgemaakt van **hoe we de elementen vinden**.

|  |
| --- |
| template<typename T, typename E >  void range\_apply( T begin, T end, void f( E & ) ){  for( T p = begin; p != end; p++ ){  f( \*p );  }  }    char letter = 'a';  void alphabet( char & x ){  x = letter++;  }  void main(){  std::array<char, 10 > a;  range\_apply( a.begin(), a.end(), alphabet );  range\_print( a.begin(), a.end() );  . . .  } |

De laatste code heft wel een lelijk aspect: we moeten een globale variabele gebruiken om de alphabet functie te laten onthouden waar hij was. Als een ander stuk code (in een andere thread) het zelfde aan het doen is loopt dit natuurijk fout, en globale variabelen zijn zowiezo niet prettig. Een manier om dit te vermijden is een lambda te gebruiken: dan kunnen we een lokale variabele gebruiklen, en die via de capture lijst door de lambda laten gebruiken. Onze range\_apply vereist een functie pointer en zal dus geen lambda accepteren, maar dit kunnen we oplossen door de range\_apply een std::function te laten accepteren, of van het type een template parameter te maken. In dit geval is er een nog makkelijkere oplossing: er is in de library (#include <algorithm>) al een for\_each die zo’n lamba kan accepteren.

|  |
| --- |
| #include<algorithm>    char next = 'A';  for\_each(  a.begin(),  a.end(),  [ &next ]( char&c ){ c = next++; }  );  range\_print( a.begin(), a.end() ); |

Tot nu toe hebben we als range (van begin totaan end) steeds de hele container meegegeven. Dit hoeft niet. Stel dat we van het alphabet alleen de letters k .. p willen omzetten in hoofdletters, dan kunnen we de k en de p opzoeken, die plekken meegeven als begin en end, en zo de omzetting tot die range beperken.

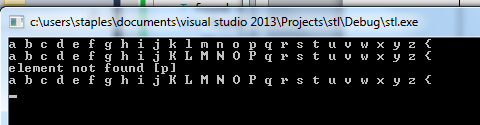
|  |
| --- |
| std::array<char, 27> a;  char next = 'a';  for\_each( a.begin(), a.end(), [ &next ]( char&c ){ c = next++; } );  range\_print( a.begin(), a.end() );  for\_each(  find( a.begin(), a.end(), 'k' ),  find( a.begin(), a.end(), 'p' ) + 1,  []( char&c ){ c = ( c - 'a' ) + 'A'; }  );  range\_print( a.begin(), a.end() ); |

Deze code bevat wel een probleem: als de letter ‘p’ niet gevonden wordt in de range, dan geeft de tweede find zijn end iterator terug. Dat is op zich prima, maar als we daar dan 1 bij optellen krijgen we een iterator waarde die 1 te groot is voor de container, en de for\_each gaat dus ook het end element (dat is 1 voorbij de geldige waarden) benaderen. Dat is undefined behaviour, dus in de prakrijk moeten we hier voorzorgen voor nemen: we moeten zoeken naar ‘q’, of controleren dat we een geldige iterator hebben teruggekregen (!= en) voordat we er 1 bij optellen. Een andere oplossing is onze eigen find te maken, die een exception genereert als het element niet gevonden wordt. Het is makkelijk (voor de find functie) als die exceptie een template is, want dan kan hij gebruikt worden voor het ‘niet vinden’ van alle soorten waarden (mits die opgeteld kunnen worden bij een string).

|  |
| --- |
| template<typenameT>  class element\_not\_found : public std::exception {  public:  element\_not\_found( const T & element ):  s{ std::string{ "element not found [" } + element + "]" }  {}  constchar \* what() const override {  return s.c\_str();  }  private:  std::string s;  }; |

De find\_or\_throw is zelf natuurlijk ook een template functie, met als template parameters de iterator, de gezochte waarde, en de exceptie. De exceptie zelf is een template class, dus we moeten aangeven voor welk type we die willen hebben. De functie zelf is verder simpel: gebruik de standaard find om het element te vinden, als dat niet lukt dan wordt de exceptie gegenereerd.

|  |
| --- |
| template<typename T, typename V, typename E = element\_not\_found<V>>  T find\_or\_throw( T begin, T end, const & value ){  T found = find( begin, end, value );  if( found == end ){  throw( value );  }  return found;  } |

De gebruiker van find\_or\_throw moet beslissen wat er gebeurt als het element niet gevonden wordt. In dit voorbeeld wordt een melding geprint, maar het is goed denkbaar dat het catch blok een lege body heeft: als de gezochte begin en/of eindpunten niet worden gevonden dan moet er niets gebeuren.

|  |
| --- |
| . . .  for\_each(  find\_or\_throw( a.begin(), a.end(), 'k' ),  find\_or\_throw( a.begin(), a.end(), 'p' ) + 1,  []( char&c ){ c = ( c - 'a' ) + 'A'; }  );  range\_print( a.begin(), a.end() );  try {  for\_each(  find\_or\_throw( a.begin(), a.end(), 'a' ),  find\_or\_throw( a.begin(), a.end(), 'p' ) + 1,  []( char&c ){ c = '#'; }  );  } catch( constelement\_not\_found<char>& e ){  std::cout << e.what() <<"\n";  }  range\_print( a.begin(), a.end() ); |

Een iterator die een operator++() heeft wordt een **forward** iterator genoemd, omdat je ermee voorwaarts door zijn container kan lopen. Een iterator die een operator—() heeft wordt een **backward** iterator genoemd. Met zo’n iterator kan je dus achteruit (van het laatste element naar het eerste element) door de container lopen. Een iterator die beide operaties heeft wordt een **bidirectional** iterator genoemd. Daarmee kan je dus heen-en-weer door zijn container lopen. De ’krachtigste’ vorm van iterator heeft een operator+(int): daarmee kan je niet alleen voorwaarts en terug door de container lopen, maar ook in 1 keer naar een willekeurig element springen. Dit wordt een **random access** iterator genoemd. De array<> die we tot nu toe alleen maar hebben gebruikt als forward iterator is zo’n random access iterator.

Tot nu toe hebben we een container altijd in ‘voorwaardse’ richting doorlopen: begin, begin+1, begin+2, etc. Soms is het nodig van achteren naar voren te werken. In de for\_each aanroep begin en end omdraaien werkt niet, dan worden de elementen end, end+1, etc. benaderd. De juiste range is end-1 …. begin - 1, en dan in plaats van vooruit stappen achteruit stappen. Het is eenvoudig om een variant van range\_apply te maken die achterstevoren werk. We moeten dan wel voorkomen dat de iterator ooit naar begin - 1 wijst, want zou een ongeldige iterator waarde zijn.

|  |
| --- |
| template<typename T, typename E>  void range\_apply\_reverse( T begin, T end, void f( E & ) ){  for( T p = end; p != begin; p-- ){  f( \*( p - 1 ) );  }  }  . . .  std::array<char, 10 > a;  range\_apply\_reverse( a.begin(), a.end(), alphabet );  range\_print( a.begin(), a.end() ); |

Met deze oplossing voor het omkeren moeten we dus voor iedere functie die op een range werkt een variant maken die in omgekeerde richting werkt. Een andere benadering is om wat er in de container zit om te draaien. Dit kost wel wat CPU cycles, maar het is heel eenvoudig op te schrijven omdat er in de library al een functie template zit die precies dat doet.

|  |
| --- |
| std::array<char, 27> a;  char next = 'a';  for\_each(  a.begin(),  a.end(),  [ &next ]( char&c ){ c = next++; }  );    range\_print( a.begin(), a.end() );  std::reverse( a.begin(), a.end() );  range\_print( a.begin(), a.end() ); |

Stel dat je een container wil doorlopen en alle elementen die aan een bepaalde voorwaarde voldoen wil verwijderen. De vector heeft een operatie erase, dus het lijkt eenvoudig: loop door de elementen met bv. for\_each, en roep erase aan als je een candidaat hebt gevonden die weg moet. Maar er zijn twee problemen: erase vereist als parameter een iterator, en binnen de loop heb je het data element ter beschikking, niet de iterator. Een tweede probleem is dat je door het verwijderen van een element uit een vector alle elementen 1 positie opschuift, waardoor je iterator nu naar het volgende element wijst. De increment aan het einde van de loop zorgt er nu voor de je een element overslaat. De oplossing voor beide problemen is het gebruik van de std::remove\_if functie template, die verplaatst de elementen die wel aan de voorwaarde voldoen naar de ‘voorkant’ van de container, en returnt een iterator naar het eerste element dat niet voldeed. De std::remove\_if kan zelf niet die laatste elementen verwijderen (die operatie vereist toegang to de container zelf), dus we moeten nog die elementen verwijderen (vanaf het resultaat van std::remove\_if totaan end()) door een a.erase() call.

|  |
| --- |
| std::vector<int> a;  for( int i = 0; i < 20; i ++ ){  a.push\_back( i );  }  range\_print( a.begin(), a.end() );  a.erase(  std::remove\_if(  a.begin(),  a.end(),  []( intx ){ return ( x % 3 ) == 0; }  ),  a.end()  );  range\_print( a.begin(), a.end() ); |

De Standard Template Library (STL) biedt een aantal container klassen, die je kan gebruiken om data op te slaan. De opgeslagen data kan je benaderen via de iterator die iedere container implementeert. De volgende tabel vemeldt de belangrijkste vier containers.

|  |  |
| --- | --- |
| **STL container** | **Kenmerken** |
| array | Array: vaste omvang, op de stack, geen range checking (wel in at()) |
| vector | Dynamisch array: op de heap, start leeg, groeit en krimpt naar behoefte |
| unordered\_set | Verzameling (geen duplicated, geen ordening) |
| map | Associatief array |

Als je gewoon data elementen wilt opslaan en je weet hoeveel elementen dan is de std::array< T, N > de juiste container. Dit is eigenlijk gewoon een array, maar zonder de impliciete conversie naar een pointer. Ook heeft het een at() operator, die voor een geldige index het zelfde doet als de operator[ ], maar een exception genereert als de index ongeldig is. Een array wordt op de stack (of globaal) opgeslagen, de andere containers maken gebruik van de heap.

Als je niet van te voren weet hoeveel elementen je wilt opslaan dat is std::vector< T > een goede keuze. Zo’n vector is initieel leeg, en je kan er met de push\_back() of push\_front() methoden elementen aan de achterkant resp. aan de voorkant aan toe voegen.

Als je een vezameling (set) van elementen wilt opbouwen gebruik dan de undordered\_set< T >. Je kan er elementen aan toevoegen met insert() en verwijderen met erase(). In een verzameling kan ieder element maar 1 keer voorkomen, dus als een element al voorkomt in de set en je voegt het nog een keer toe dan blijft het element er 1 keer in voorkomen. In een unordered\_set() zitten de elementen op een onvoorspelbare volgorde. In de ‘gewone’ set< T > zijn ze gesorteerd op grootte. Daarnaast is er nog de unordered\_multiset< T > en de multiset< T >, waarin ieder element meerdere keren kan voorkomen.

In het volgende voorbeeld wordt vier keer een element in de (ongeordende) set gestopt, maar aangezien daar een ducplicaat bij zit bevat de set maar 3 elementen. Merk op dat de volgorde waarin ze worden afgedrukt niet de volgorde is waarin ze er in gestopt zijn. Met erase kan je een element verwijderen.

|  |
| --- |
| std::unordered\_set< std::string > present;  for( auto name : { "piet", "marie", "noortje", "klaas", "piet" } ){  present.insert( name );  }  print ( present );  present.erase( "noortje" );  print( present ); |

Als je elementen wil opslaan en terugvinden op basis van een key (sleutel) dan gebruik je de map< K, T >. Het K type is de key, T is het type van de opgeslagen elementen. Een map wordt ook wel een associatief array genoemd. De elementen kan je benaderen met de array notatie. Als je dan een key gebruikt die nog niet in de map voorkomt dan wordt dat element aangemaakt (door zijn default constructor).

In het volgende voorbeeld wordt een map met drie ‘studenten’ aangemaakt, geindexeerd door kun stundent nummer. Met erase() wodrt 1 student uit de map verwijderd. Als een niet bestaand element met de array notatie wordt opgevraagd dan wordt het aangemaakt. De default constructor van string maakt een lege string aan, dus dat is wat we terugkrijgen. Als we een niet bestaande student benaderen met de at() methode dan wordt een exceptie genenereerd.

|  |
| --- |
| std::map< int, std::string > students;  students[ 1234567 ] = "Joost Prinssen";  students[ 9101112 ] = "Aart Staartjes";  students[ 1001001 ] = "Wieteke van Dort";  for( const auto & student : students ){  std::cout << student.first << " '" << student.second << "'\n";  }  std::cout << "\n";  students.erase( 1234567 );  for( const auto & student : students ){  std::cout << student.first << " '" << student.second << "'\n";  }  students.erase( 1234567 );  std::cout << "\n";  std::cout << "[ 1001001 ] = " << students[ 1001001 ] << "\n";  std::cout << "[ 3333333 ] = " << students[ 3333333 ] << "\n";  try {  std::cout << "at( 44444444 ) = " << students.at( 444444444 ) << "\n";  } catch( const std::exception & e ){  std::cout << e.what();  } |

De volgende tabel geeft de methoden die door de meeste containers worden geimplementeerd.

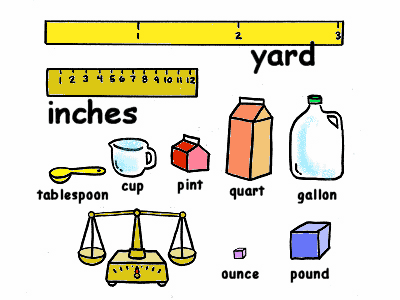
|  |  |
| --- | --- |
| **operatie** | **effect** |
| push\_front | Inserts element before the first (not for vector) |
| pop\_front | Removes the first element (not for vector) |
| push\_back | Inserts element after the last |
| pop\_back | Removes the last element |
| empty | Boolean indicating if the container is empty |
| size | Returns the number of elements |
| insert | Inserts an element at a particular position |
| erase | Removes an element at a particular position |
| clear | Removes all the elements |
| resize | Resizes the container |
| front | Returns a reference to the first element |
| back | Returns a reference to the last element |
| [] | Subscripting access without bounds checking (vector, array, dequeue) |
| at | Subscripting access with bounds checking (vector, array, dequeue) |

Als je je gegevens in een STL container hebt opgeslagen dan moet je er waarschijnlijk ook iets mee gaan doen. Als wat je moet doen al beschikbaar is als STL algorithme gebruik dat dan. Een goed overzicht vind je op <http://www.cplusplus.com/reference/algorithm/>. Het is de moeite waard dat eens door te nemen.

Als je door alle elementen van de container heen moet lopen, dan is de for( : ) loop het gemakkelijkst. Denk er aan dat je het loop element als reference declareert, anders krijg je (wellicht zonder het te beseffen) een *copie* van het element.

Als je door een subrange heen moet lopen, gebruik dan de STL for\_each(). Je moet dan wel (anders dan bij de for( : ) loop) de actie die je op de elementen uitvoert aanleveren in de vorm van een functie of een lambda. Gebruik een lambda als de actie kort is en maar 1 keer in je programma voorkomt, gebruik een losse functie als de actie langer is en/of meer dan 1 keer in je programma voorkomt. Als de functie context nodig heeft (bv. een variabele om opeenvolgden waarden aan te maken) gebruik dan een lambda die een functie aanroept, en geef die context mee aan de lambda in zijn caputure, die ze weer doorgeeft aan de functie als reference parameters.

# Waarden



|  |
| --- |
| behandelde begrippen:   * Grootheden * Meetschalen * Eenheden * User-defined literals * Stelsels * Boost::units |

## Grootheden

Een grootheid (quantity) is iets waarvan we de waarde kunnen meten of tellen. Het komt vaak voor dat je een dergelijke waarde in je programma moet opslaan en er mee moet rekenen. Je kan al deze waarden in je programma representeren als integers of floats, maar dan kan je in je code veel dingen doen die volkomen onzinnig zijn, bv. blauw met 3 vermenigvuldigen of een afstand optellen bij een temperatuur.

Je moet zulke dingen natuurlijk niet doen in je programma, en tot op zekere hoogte kan de compiler je daar bij helpen. Hiervoor moet je bewust zijn van de soorten schalen waarin grootheden kunnen worden uitgedrukt, en van de relaties tussen verschillende grootheden (uitgedrukt in hun eenheden).

## Schalen

Het is daarbij van belang je te realiseren dat er verschillende soorten schalen die uitrukken hoe waarden zich onderling verhouden. Er zijn 4 bekende soorten schalen: nominaal, ordinaal, interval, en ratio. In deze volgorde neemt wat je (zinvol) met waarden in zo’n schaal kan doen toe.

### Nominaal

Het eenvoudigste soort schaal is de nominale schaal. De waarden in zo’n schaal zijn verschillend van elkaar (anders waren het geen aparte waarden), maar dat is alles dat je kan zeggen over die waarden. Je kan ze met name niet op een (zinvolle) volgorde leggen. Denk bv. aan landen of kleuren. Soms wordt aan een nominale opsomming om practische redenen wel een volgorde toegekend, bv. om te kunnen sorteren en opzoeken. Kleuren zou je alfabetisch kunnen sorteren.

### Ordinaal

Als de waarden van een schaal wel een duidelijke volgorde hebben dan spreken we van een ordinale (geordende) schaal. Een set waarden uit een ordinale schaal kan je dus (zinnig) sorteren. Denk bij een ordinale schaal aan de manier waarop enquetes meestal om je mening vragen: je mag kiezen uit een reeks termen, bv. zeer goed, goed, matig, slecht, zeer slecht. Daar zit een duidelijke volgorde in.[[19]](#footnote-19)

### Interval

Bij een interval schaal hebben de waarden uit de schaal niet alleen een volgorde, maar daarbovenop zijn de ondelinge afstanden gelijk[[20]](#footnote-20). Je kan dus verschillen tussen waardne met ellkaarr vergelijken, en het gemiddelde bepalen. Denk aan de temperatuur, uitgedrukt in graden Celius of Farenheid, of een jaartal.

### Ratio

Bij een ratio schaal zitten de waarden op gelijke afstanden, en er is een (zinnig) nulpunt. Dit betekent dat je waarden (zinnig) kunt vermenigvuldigden met een getal. Ratio schalen zijn bv. de tempertauur in graden Kelvin, je leeftijd, of het slado op je bankrekening.

## Enum

Een enumerate is een type dat een reeks waarden kan aannemen, die je ieder met een naam aangeeft. Een voorbeeld is de vier windrichtingen.

|  |
| --- |
| enum { north, east, south, west } wind;  wind x = north; |

Bij deze vorm van enumerate is de enumerate zelf en soort integer, en de waarden zijn een soort integer constanten. Dit betekent dat je aan de enumerate een integer kan toekenen, je er mee kan rekenen, en dat je een probleem krijgt als je de zelfde naam wil gebruiken voor waarden van verschillende enumerates.

|  |
| --- |
| enum { north, east, south, west } wind;  wind x = 2;  x++;  x = north + east;  enum { north, south } pole; // error: duplicate enumerate value |

In C++ is er naast de bovenstaande C-style enum een nieuwe versie, de enum class. Iedere enum class is een apart type, los van integer, en de benoemde waarden zijn alleen zichtbaar ‘binnen’ de enum type naam. Hierdoor is er geen probleem met het gebruik van de zelfde naam binnen meerdere enumerates, en je kan niet per ongeluk met de waarden gaan rekenen. In nieuwe code is het sterk aangeraden enum class te gerbuiken in plaats van de gewone enum.

|  |
| --- |
| enum class { north, east, south, west } wind;  wind x = 2; // error  x++; // error  x = wind::north + wind::east; // error  enum class { north, south } pole; // OK  pole y = pole::south; |

Enumerates zijn geschik om nominale en ordinale schalen te implementeren, maar de waarden zijn geordend (je kan ze vergelijken met <, >, <=, >=) dus voor een nominale schaal zijn ze eigenlijk minder handig. Je zou zelf een soort enumerate kunnen maken die als vergelijkingen alleen == en != ondersteund, maar dat is veel werk en wordt dus niet vaak gedaan.

|  |
| --- |
| class language {  private:  int x;  constexpr language( int x ): x{ x }{}  static constexpr const char \* const names[ 3 ]{  "dutch", "english", "french"  };    public:  constexpr static language dutch\_() { return language{0}; }  constexpr static language english\_(){ return language{1}; }  constexpr static language french\_() { return language{2}; }    constexpr language( const language & rhs ): x{ rhs.x }{}    friend hwcpp::io::ostream & operator<<(  hwcpp::io::ostream & lhs,  language x );    bool operator==( const language &rhs ){  return x == rhs.x;  }  bool operator!=( const language &rhs ){  return \*this != rhs;  }  };  constexpr const char \* const language::names[ 3 ];  constexpr const language dutch { language::dutch\_() };  constexpr const language english { language::english\_() };  constexpr const language french { language::french\_() }; |

## Integers en floats

Voor interval en ratio schalen kan je gebruik maken van de diverse integer en float typen, afhankelijk van je gewenste bereik (hoogste en laagste waarde) en nauwkeurigheid. Wees je wel bevust van de beperkingen van deze typen.

Integers hebben een beperkt bereik. Een berekening die een waarde buiten het bereik zou opleveren is een overflow (of underflow). Voor signed integers is een overflow undefined behaviour. Dat betekent niet alleen dat je niet weet wat er aan waarde uitkomt, maar het kan betekenen dat je programma iets volkomen anders doet dan je verwacht had. Voor unsigned integers worden alle berekeningen uitgevoerd modulo het bereik van het type. Hierdoor zijn signed integer berekeningen vaak sneller, maar unsigned berekeningen zijn veiliger.

Floating point typen (float en double) hebben ook een bereik, maar dat is zo groot (zeker voor double: 1.7E-308 to 1.7E+308) dat je daar meestal geen problem mee zal hebben. Wat wel een rpobleem kan zijn is dat floating point getallen altijd een benadering zijn. Hierdoor zijn ze eigenlijk noot geschikt voor code waarin je wil kijken of twee (floationg point) getallen gelijk zijn. Wat mij betreft had het floating point type geen operatoren == en != moeten hebben (we ook geen =< en >=). Een ander probleem van floating point getallen in een real-time systeem is dat de snelheid van de meeste floating point hardware berekingen onafhankelijk is van de waarden (dat is een heel wenselijke eigenschap in een real-time systeem), behalve voor zogenaamde ‘de-normalized’ waarden. Dat zijn waarden net boven (of onder) nul, die qua formaat afwijken van de ‘normale’ waarden waar de hardware voor is ontworpen. Het resultaat is dat berekeingen op zulke waarden 10 .. 100 keer meer tijd kosten dan berekeningen op normale waarden. Veel systemen hebben de mogelijkheid dit te vermijden door de-normalized waarden als 0 te behandelen, maar hoe je dit moet bereik verschilt per systeem.

Voor interval schalen moet je bedenken dat vermenigvuldigen en delen van waarden niet zinnig is. Wat is twee keer 10°C? Niet 20°C![[21]](#footnote-21) Voor zulke schalen zouden we dus eigenlijk integer en floating point typen moeten gebruiken die geen \* en / operatoren hebben. (En + en – zijn ook verdacht..)

## Eenheden

Als we een waarde meten dan is het resultaat niet alleen een getal. Is mijn lengte 185, 1.85, of 1.955466109E-16? (Om nog maar te zwijgen van de benadering 6”1). Zo’n getal heeft pas betekenis als je weet in welke eenheid (cm, m, C) het is uitgedrukt. Als je een kale (integer of floating point) variabele gebruikt om een dergelijk waarde op te slaan dan is het sterk aan te raden de eenheid in de naam van de variabele te verweken, bv length\_cm.

In 1999 ging de Mars Climate Orbiter satelliet verloren doordat de een onderdeel van besturingssoftware waarden doorgaf uitgedrukt in het Amerikaanse-Britse systeem (in pond-seconden), aan software die deze waarde interpreteerde volgens het SI systeem (in Newton-seconden). Hierdoor remde de satelliet te weinig, kwam in een te lage baan om Mars terecht, en verbrandde. (De atmosfeer van Mars is heel ijl maar toch genoeg om een dergelijke satelliet af te remmen.)

Het is niet al te lastig om een eenvoudige template klasse te maken waarin je waarden kan uitdrukken ten opzichte van een gegeven eenheid, waarmee je eenheid problemen zoals die van de Mars orbiter kan voorkomen.

Het onderstaande klasse template unit wordt geinstatieerd met twee getallen Num en Denum. De klasse slaat een double waarde op, geschaald ten opzichte van Num/Denum. Het zou makkelijker zijn om in plaats van Num en Denum een double waarde mee te geven, maar floats en doubles zijn niet toegestaan als template parameters. In plaats daarvan heeft de klasse een static constante base die dezelfde rol vervult. Bij het opslaan van de waarde wordt de waarde die je opgeeft gedeeld door base, bij het ophalen van de waarde wordt die eerst vermenigvuldigd met base. Deze functies \_\_set en \_\_get zouden eigenlijk private moeten zijn, maar dat zou het voorbeeld een stuk ingewikkelder maken.

De unit klasse heeft een default constructor zodat er variabelen kunnen worden aangemaakt zonder een initiele waarde. Er is een assignment operator, die de onstachaalde waarde van de rhs ophaalt, en die (geschaald) opslaat. Een unit kan worden vermenigvuldigd met en gedeeld door een double, en het resultaat is weer een unit. Deze operatoren hoeven niet te ontschalen en weer te schalen, want die twee operaties zouden elkaar opheffen.

Tot slot moeten er nog manieren zijn om een unit waarde te creeeren, en uit een unit waarde weer een getal te halen. Een unit waarde kan je maken door een (double) waarde te vermenigvuldigen met de constante ‘one’, en je kan van een unit waarde weer terug gaan naar een double door te delen door een unit waarde.

|  |
| --- |
| template < int Num, int Denum >  class unit {  private:  double value;  static double constexpr base = static\_cast<double>(Num) / Denum;  unit(double value) : value{ value } {}  public:  void \_\_set(double x) { value = x / base; }  double \_\_get() const { return value \* base; }  unit() : value(0) {}  template< typename T\_rhs >  unit & operator=(const T\_rhs & rhs) {  \_\_set(rhs.\_\_get());  return \*this;  }  unit operator\*( double rhs ) const {  return unit{ value \* rhs };  }  unit operator/( double rhs ) const {  return unit{ value / rhs };  }  static unit one() { return unit(1); }  double operator/( const unit & rhs ) const {  return value / rhs.value;  }  }; |

De onderstaande demonstratie laat zien dat we met de unit klasse een paar concrete typen kunnen maken (lengte in cm en lengte in inch). Een waarde die is uitgedruk in het ene type kunnen we toekennen aan een variabele van het andere type, en daarbij wordt de waarde automatisch mee-geschaald. Twee operator<< functies drukken de typen af met een passende suffic (cm of inch).

|  |
| --- |
| using length\_in\_cm = unit< 1, 1 >;  using length\_in\_inch = unit< 254, 100 >;  template< typename T >  T & operator<<(T &lhs, const length\_in\_cm & x) {  return lhs << x / length\_in\_cm::one() << " cm";  }  template< typename T >  T & operator<<(T &lhs, const length\_in\_inch & x) {  return lhs << x / length\_in\_inch::one() << " inch";  }  int main(int argc, char \*argv[]) {  length\_in\_cm my\_length;  length\_in\_inch also\_my\_length;  my\_length = length\_in\_cm::one() \* 185.0;  also\_my\_length = my\_length;  std::cout << my\_length << "\n";  std::cout << also\_my\_length << "\n";  } |

Als je in C en C++ een literal getal opschrijft, dan heeft die literal een bepaald type, meestal int of float. Met een suffix kan je forceren dat de literal een ander type heeft, bv 1L is de waarde 1 van het type long. C++ heeft de mogelijkheid om eigen suffixen te definieren voor je eigen types. Dit het user-defined literals en dat doe je door een operator ”” te definieren met als naam de suffix die je wil gebruiken. Die naam (en dus suffix) moet beginnen met een underscore. De functie moet 1 parameter hebben, en er zijn wat beperkingen (die hier niet ebsproken worden) aan het type.

In het onderstaande voorbeeld is er een user-defined suffix \_inch, met een long (!) double parameter.

|  |
| --- |
| length\_in\_inch operator "" \_inch( long double x ){  return length\_in\_inch::one() \* x;  }  auto length = 70.0\_inch;  my\_length = length;  also\_my\_length = my\_length;  std::cout << length << "\n";  std::cout << my\_length << "\n";  std::cout << also\_my\_length << "\n"; |

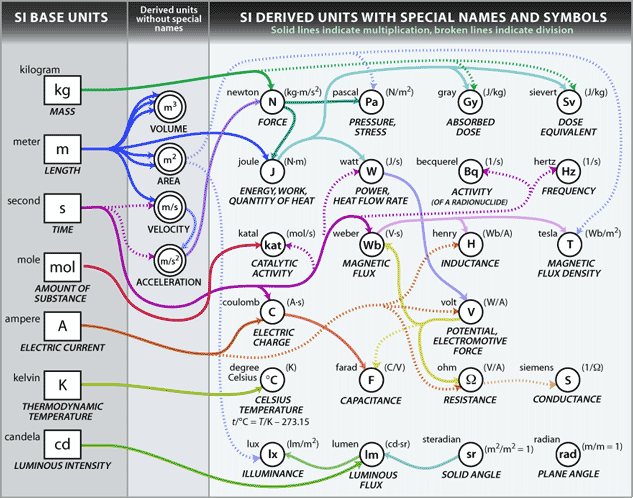
Een (groot?) problem met user-defined literals is dat je een literal die in een namespace is gedefinieerd niet kan gebruiken zonder alle declaraties uit die namespace zichtbaar te maken met ‘using <namespace-name>;’. Er is namelijk geen syntax om bij het schrijven van de literal een namespace op te geven, zoals je wel kan doen bij een variabele, klasse of functie. Maar dan is er dus geen manier meer om onderscheid te maken tussen de user-defined literals met zelfde naam uit twee verschillende namespaces.

## Stelsels

Een unit klasse zoals hiervoor beschreven kan er voor zorgen dat je een bepaalde grootheid (in het voorbeeld lengte) kan uitdrukken in verschillende eenheden. Maar in de praktijk hebben we te maken met veschillende grootheden, bv. naast lengte ook tijd. Lengte en tijd kan je niet bij elkaar optellen. Dit kunnen we bereiken door de unit template een extra template parameter mee ge geven, met (bv.) de waarde 0 voor lengte en 1 voor tijd. Met wat extra inspanning kunnen we er voor zorgen dat die klassen niet met elkaar kunnen werken.

Maar zo simpel zit de echte wereld niet in elkaar: lengte en tijd optellen is onzin, maar lengte delen door tijd is wel zinnig, en dat levert snelheid op. Als de eenheid van afstand m (meter) is en van snelheid s (seconde), dan is de eenheid van snelheid m / s ofwel m \* s-1. Het blijkt dat je alle interessante grootheden kan uitdrukken in het product van een beperkt aantal basis grootheden. Zulke grootheden worden gespecificeerd door de machten van de basis grootheden, en die machten zijn vrijwel altijd kleine positieve of negatieve gehele getallen.

Welke basis grootheden je gebruikt, en zeker welke basis eenheden je voor die grootheden hebruikt, is eigenlijk arbitrair. Zo’n keuze wordt een eenheden stelsel genomen. Et meest gebruikte stelsel is het SI stelsel. Het volgende plaatje toont de SI basis eenheden en de daarvan afgeleide eenheden.



## Boost::units

De boost library is een verzameling vrij beschikbare C++ libraries. De kwaliteit van deze libraries is over het algemeen erg hoog, en Boost wordt algemeen beschouwd als een soort ‘voorportaal’ voor opname in de C++ standaard bibliotheek. Een nadeel van Boost is dat de documentatie niet altijd

even goed en omvangrijk is.

# Overzicht van Design Patterns

|  |
| --- |
|  |

Design Patterns zijn manieren om code te structureren die hun waarde hebben bewezen. De volgende tabel vat de in deze module en in eerdere modules behandele Design Patterns en andere programmeer strategieen samen.

## Adapter

Een adapter is een klasse die een gegeven interface (bv een window met een draw\_pixel methode) implementeert op een andere klasse )of library) die in essentie de zelfde functionalitiet biedt, maar met een andere interface.

## Command

Het command pattern slaat een actie op als object (waarde), zodat die uitgevoerd kan worden op een ander moment dan waarop de actie onstaat. Je gebruikt dit bv. om een commando op te slaan in een event queue, maar je kan het ook gebruiken om alle commando’s die een gebruiker heeft uitgevoerd op te slaan, zodat je ze later kan herhalen. Als je dan het laatste commando negeert heb je een ‘undo’ feature.

## Composite

Een composite is een object dat samengesteld is uit meerdere objecten, en wiens gedrag de som van de objecten is. Een voorbeeld is een scherm object dat is samensgesteld uit een aantal (eenvoudigere) schermobjecten.

## Decorator

Een decorator is een klasse die met behulp van een onderliggend object dat een geven interface implementeert, die zelfde interface biedt, maar met een verandere of toegevoegde functionaliteit. Denk bv aan een window met een draw\_pixel(x,y,p) functie, waarbij de decorator zorgt dat alle schrijfacties in spiegelbeeld op de onderliggende window worden afgebeeld.

## Dirty Flag

Vermijdt (her) berekening van een zelfde waarde door de oude waarde op te slaan. Je hebt dan een Dirty Flag nodig om te weten of je de waarde moet her-berekenen.

## Double Buffer

Als je tijdens het bijwerken van gegevens niet geconfronteerd wil worden met een onvolledige versie van die gegevens pasx je double buffering toe: er zijn twee buffers, waarvan telkens de ene wordt opgebouwd terwijl de andere gebruikt wordt.

## Factory

Een factory is een functie die afhankelijk van run-time gegevens verschillende typen objecten terug kan geven. In C++ betekent dit dat de factory functie een (smart) pointer teruggeeft naar de superklasse van de klassen die hij can creeeren.

## Flyweight

Het flyweight pattern reduceert de omvang van (een groot aantal) objecten door de gemeenschappelijke delen niet in ieder object onder te brengen, maar in ieder object een referentie naar een gedeeld ‘template’ object op te nemen. Het zelfde principe kan toegpast worden om de hoeveelheid code die geproduceerd wordt door verschillende instatiaties van een template te beperken door de gemeenschappelijk functies onder te brengen in een (niet template) superklasse.

## Game Loop

Een game loop is de standaard manier om een spel te realiseren waarin de game-play (en het renderen van het beeld) doorloopt als de gebruiker niets doet, zonder multi-threading te gebruiken. Om dit te bereiken wordt na het afhandelen van input van de gebruiker de game situatie bijgewerkt (physics engine, anutonomous characters, etc). In de eenvoudigste vorm gebeurt dit met een vaste hoeveelheid game time per iteratie van de game loop. Een complexere implementatie past de de hoeveelheid verstreken game-time per loop iteratie aan naargelang de snelheid van het systeem, zonder daarbij de realiteit van de bijwerking aan te tasten.

## Listener (Observer)

Het listerner pattern zorgt dat de code die een event detecteert (bijvoorbeeld de taak die om de 10 ms een knop pollt) niet hoeft te weten welke code dat event afhandelt (dat zou verkeerdom zijn: dan heeft de low-level toepassings-onafhankelijke code weet van de appliactie-specifieke code). In plaats daarvan meldt de verwerkende code zich (als Listener) aan bij de detecterende code. De verwerkende code moet dich daartoe conformeren aan een bepaalde interface, die door de detecterende code is voorgeschreven.

## MVC

(niet behandeld) Het Model View Controller pattern schrijft voor dat de gegevens (Model), de manier waarop die gegevens aan de gebruiker worden getoond (View) en de manier waarop de gebruiker die gegevens kan wijzigen (Controller) gescheiden moeten worden. Dit pattern wordt veel toegepast in web applicaties, vooral als meerdere applicaties of onderdelen van applicaties met de zelfde data moeten werken.

## NVI

(niet behandeld) Het Non-Virtual Interface pattern schrijft voor dat de interfae functies van een klasse die bedoeld is om te subklassen niet virtual moeten zijn. Inplaats daarvan moeten die een projected virtual function aanroepen, en dat is de functie die een subklasse implementeert. Dit maakt het mogelijk dat de publieke interface anders is dan de implementatie interface. Dit kan handig zijn om duplicatie van code in alle implementaties te voorkomen, bv. het checken van parameters.

## Proxy

Een proxy is een object dat optreedt als een plaatsvervanger voor een ander object. Het proxy object biedt de zelfde interface, en sluist alle gebruik van doe interface door naar het eigenlijke object. Een proxy object wordt vaak gebruikt om een object op afstand te gebruiken: het proxy object en het eigenlijke object leven dan op verschillende computers, en een verzoek dat bij de proxy binnenkomt wordt via een communicatie kanaal doorgegeven naar het eigenlijke object (en resultaten worden teruggestuurd naar de proxy, die ze aan de aanroeper doorgeeft).

## RAII

Dit is een C++-specifiek pattern. RAII houdt in dat een resource (bv. een geallokeerd stuk geheugen, of een file handle) die tijdelijk nodig zijn wordt ondergebracht in variabele waarvan de levensduur overeenkomt met de tijd dat de resource nodig is. De destructor van de variabele zorgt voor het vrijgeven van de resource. Hierdoor is het tijdig vrijgeven van de resource verzekeerd, ook als het stuk code verlaten wordt op een bijzondere manier (bv. door een exception).

## Rule-of-3

Dit is een C++-specifiek pattern. Als je een klasse schrijft die een resource (bv. geallokeerd geheugen) beheert, dan moet je deze resource 1) in de constructors reserveren, 2) in de destructor vrijgeven, en 3) in de assignment operator zorgen dat er geen dubbele verwijzingen en/of lekken ontstaan. De rule-of-3 wordt soms zo geformuleerd: Als je de noodzaak voelt 1 van deze drie soorten functies te schrijven, dan is er een grote kans dat je de andere 2 ook moet schrijven. De regel van 3 is in sommige gevallen achterhaald door het gebruik van RAII en smart pointers, die er beiden voor kunnen zorgen dat de default constructor voldoende is.

## SFINAE

Dit is een C++-specifiek principe: bij het bepalen welke van de voorhande template specializaties gebruikt gaat worden voor een ‘aanroep’ van een template, is het geen probleem als er bij het expanderen van de template header van die specializatie iets ‘niet klopt’. Subsitutie Failure Is Not An Error, maar betekent alleen dat de betreffende specializatie dus geen kandidaat is om gebruikt te worden. Dit wordt veelvuldig gebruikt om een voor een specifiek geval passende template specializatie te selecteren (of eigenlijk: de niet passende specializaties uit te sluiten).

## Singleton

Soms is het lastig om een object dat je overal nodig hebt en waarvan er maar 1 kan (en mag) bestaan aan iedere functie die het nodig heeft als parameter door te geven. Bekende voorbeelden zijn de printer, het beeldscherm, en de console. Het singleton pattern biedt een centrale plaats waar (een referentie of pointer naar) het betreffende object kan worden opgevraagd. Dit biedt als voordeel boven een globaal object dat als het object nooit wordt aangevraagd het ook niet aangemaakt hoeft te worden. Het singleton pattern is omstreden, omdat het een verborgen interface is van de code die er gebruik van maakt.

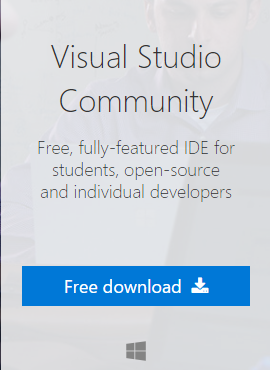
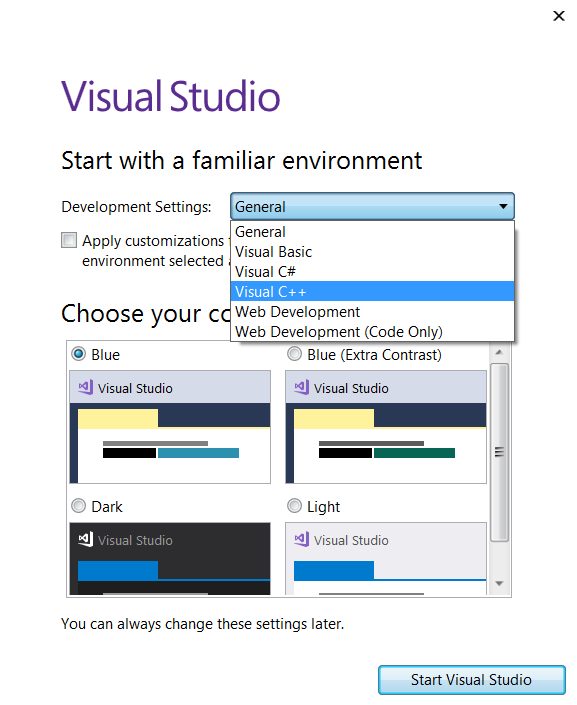
## Update Method / State Machine

Als meerdere onderdelen van je code onfahankleijk van elkaar dingen doen en daartussen wachten op evenets of het verstrijken van tijd dan kan je dieonderdelen coderen als aparte taken. Maar als je geen multitasking becshikbaar hebt dan moet je de activiteiten van die onderdelen onderbregne in een update() functie, die wordt aangeroepen als er een event voor dat ondereel is. De update() functie moet dan het nodig doen, zonder te gaan staan wachten.

# Appendix : Visual Studio, SFML, Boost

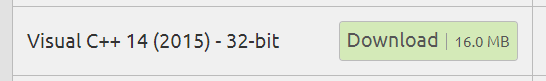
## Installeren

Haal de zip met code van Sharepoint en plaats die in de directory voor dit vak. (Ik neem aan dat je je vakken organizeert met een directory per vak?) Installeer Visual Studio 2017 – Community, te vinden op <https://www.visualstudio.com/downloads/> Mijn laptopje was er een half uur zoet mee. Selecteer bij het installeren de C++ tools. Selecteer bij het opstarten de Visual C++ settings.

Download SFML 2.4.2 for Visual C++ 14 (in de Visual Studio 2017 zit C++ 2015) 32 bits, te vinden op http://www.sfml-dev.org/download/sfml/2.4.0/

Het vervolg neemt aan dat je dit installeert in C:/SFML-2.4.2 dus als je het ergens anders installeert moet allerlei paden overeenkomstig aanpassen. Bespaar je die moeite.



Download de example code uit de repository <http://www.github.com/wovo/v2cpse2-examples>. Start msvc door double-clicken op v2cpse2-msvc.sln. Select the 00-01 … project. Click Project > Export Template > (Project Template ) select the 00-01 … project > next > (automatic import checked) finish.

## Solution aanmaken

Visual Studion werkt met een Solution die een reeks projecten bevat. Om een nieuwe solution aan te maken klink je File 🡺 New 🡺 Project, je kan dan onderin aangeven in welke solution directory het project wordt aangemaakt, en of het binnen een nieuwe solution komt of toegevoegd wordt aan een oude solution. Er wordt voor het project en subdirectory aangemaakt in de solution directory (met als naam de naam van het project).

Je kan een bestaande solution openen door te dubbel-klikken op de .sln file. Ik hou al mijn voorbeeld code in 1 solution, maar dat betekent wel dat het even tijd kost om die solution met al zijn projecten te laden.

## Project aanmaken – via template

Bij het aanmaken van een project geef je aan wat op welk template je het project baseerd. Voor V2CPSE2 is dat het template dat je hebt aangemaakt.

## Project aanmaken – met de hand

Als je om de een of andere reden een project met de hand aanmaakt dan moet je zijn eigenschappen aanpassen: klik rechts op het project, en selecteer properties.

* **Configuration Properties**
  + In **C/C++**
    - In **General** 
      * In **Additional Include Directories** : voeg “C:/SFML-2.4.0/include” toe
      * Voor Warning Level: selecteer W3
      * Voor **Treat Warnings As Errors** : selecteer Yes
    - In Preprocessor:
      * In Preprocessor Definitions: add SFML\_STATIC
    - In Language
      * Voor Disable Language Extensions: selecteer Yes
      * Voor Enforce Type Conversion Rules: Yes
      * Voor Enable Run-Time Type Information: Yes
    - In **Precompiled headers ->**
      * **Voor Precompiled headers** : selecteer “Not Using Precompiled Headers”
  + In **Linker**
    - In **General** 
      * In **Additional Library Directories** : voeg “ C:/SFML-2.4.0/lib” toe
    - In Input
      * In **Additional Dependencies** : voeg “sfml-graphics-s-d.lib;sfml-window-s-d.lib;sfml-audio-s-d.lib;sfml-system-s-d.lib;sfml-network-s-d.lib;opengl32.lib;freetype.lib;jpeg.lib;winmm.lib;gdi32.lib;flac.lib;vorbisenc.lib;vorbisfile.lib;vorbis.lib;ogg.lib;ws2\_32.lib;” toe
    - In Advaced
      * Voor targetMachine: selecteer MachineX86

Zorg dat het platform op win32 staat en je projecten (in de configuration manager) ook op win32.

## Een project bewerken

* Maak eerst het project aan. Hierdoor wordt de project directory aangemaakt.
* Copieer de source files waar je van uit wil gaan naar de project directory
* Voeg de files toe via Source Files 🡺 Add 🡺 Existing item. Het is makkelijk om alle files toe te voegen als sources, en daarna de header files verslepen naar de Header Files
* Om met het project te gaan werken, is het handig om het project te selecteren : klik rechts op het project en selecteer Set as StartUp Project
* Je kan nu onder Build het project builden, cleanen of rebuilden
* Je kan nu onder Debug het project runnen met Start Debugging (F5) of Start Without Debugging (ctrl-F5)

## Een project naam veranderen

Je kan een project binnen MSVC renamen, maar daarmee verander je alleen de naam die MSVC laat zien, niet de directory waar het project in zit. Als je project directories subdirectories zijn van de plek waar je solution file staat (de aanbevolen organisatie), dan kan je het Python script cleanup.py aanroepen om de directory namen in overeenstemming te brengen met de MSVC project namen. Verander dus eerst (in MSVC) de project namen, verlaat dan MSVC, en run cleanup.py. Het script gooit ook alle troep (object files, executables, etc) weg.

# Appendix : week schema

|  |  |  |
| --- | --- | --- |
| Week | Behandele stof | Practicum |
|  | Zelf doen  Appendix: MSVS, SFML, Boost installeren | |
| 1 | H2 t/m H5:   * Inleiding * SFML * Layout * Naamgeving | MSVC / SFML werkend;  Video ‘Global State and Singeltons’ |
| 2 | H6 t/m H8   * Undefined behavior * Lambda’s * Agile, Waterfall | Muren en stuiteren |
| 3 | H9 t/m H12   * Heap * Exceptions * For loops * Factory | Factory |
| 4 | H13   * STL   [Game Programming Patterns](http://gameprogrammingpatterns.com/contents.html) : [H17](http://gameprogrammingpatterns.com/data-locality.html), [H18](http://gameprogrammingpatterns.com/dirty-flag.html), [H3](http://gameprogrammingpatterns.com/flyweight.html)   * Optimization Patterns | STL (King James Bible) |
| 5 | H14   * waarden   [Game Programming Patterns](http://gameprogrammingpatterns.com/contents.html) : [H2](http://gameprogrammingpatterns.com/command.html), [H4](http://gameprogrammingpatterns.com/observer.html), [H6](http://gameprogrammingpatterns.com/singleton.html)   * Command, Observer, Singleton | Tic-Tac-Toe |
| 6 | H15   * Overzicht van Design Patterns   [Game Programming Patterns](http://gameprogrammingpatterns.com/contents.html) : [H8](http://gameprogrammingpatterns.com/double-buffer.html), [H9](http://gameprogrammingpatterns.com/game-loop.html)   * Double buffer, game loop | Geen nieuw practicum |

1. De student mag desgewenst werken met Linux, GCC en SFML, maar is dan zelf verantwoordelijk voor het oplossen van eventuele problemen. [↑](#footnote-ref-1)
2. Helaas zijn er (zeldzame) gevallen waarin de { } notatie het verkeerde doet (een initializer list construeren in plaats van een aantal parameters doorgeven) zodat de voorkeur voor { } niet absoluut kan zij. [↑](#footnote-ref-2)
3. In dit geval wordt het tekenen dus door een functie van de window gedaan. Het omgekeerde is misschien iets logischer: laat het tekenen gebeuren door een functie van het grafische object, en geeft de window als parameter mee. [↑](#footnote-ref-3)
4. Voor de getoonde main() verschijnt er niets, maar die window is handig voor bv. debug informatie. De meegeleverde files bevatten ten minste een melding met de naam van het programma. [↑](#footnote-ref-4)
5. Bijna alle standaarden hebben echter ook een uitzonderingsmogelijkheid om een bepaalde regel in uitzonderlijke gevallen buiten werking te stellen. Mocht je de ‘fallthrough’ dus echt nodig hebben en het gebruik ervan weegt op tegen de foutgevoeligheid en verminderde leesbaarheid, dan kan het. [↑](#footnote-ref-5)
6. Een term die je in dit verband vaak tegenkomt op internet is “nasal demons”: Als je programma Undefined Behaviour vertoont moet je niet verbaasd zijn als er gevleugelde monstertjes uit je neus komen vliegen. [↑](#footnote-ref-6)
7. Het is een compiler error als er meerdere return statements zijn en de gereturnde types niet compatible zijn. Als er geen return statment is (of er zijn return statments zonder waarde) dan is het return type ‘void’. [↑](#footnote-ref-7)
8. Deze syntax kan ook bij een gewone functie gebruikt worden, maar dat is nog niet gebruikelijk. [↑](#footnote-ref-8)
9. De signatuur is de parameter lijst en het return type. [↑](#footnote-ref-9)
10. Dit is iets wat bij gewone functies niet speelt, want je kan geen functie definieren binnen een andere functie. [↑](#footnote-ref-10)
11. Maar je kan wel globale variabelen gebruiken, want daar kan iedereen bij. [↑](#footnote-ref-11)
12. Gemiddeld kost het toevoegen dan weinig tijd: de wordt de ‘amortized’ (uitgesmeerde) cost genoemd. Voor dekstop toepassingen is dit prima, maar voor real-time systemen vaak niet. [↑](#footnote-ref-12)
13. Helaas omdat het over het algemeen beter is als een fout zo snel mogelijk aan het licht komt. De meest verveldende vorm van undefined behaviour is als de code doet wat de programmeur (twen onrechte) verwacht. [↑](#footnote-ref-13)
14. De eenvoudigste vormen van garbage collectuiion gaan aan het werk als er tekort aan geheugen is en doen dan een ‘sweep’ over het geheugen om alle ongebruikte stukken te verzamelen. Dit leidt tot een korte piek in CPU gebruik, wat voor real-time systemen onwenselijk is. Er zijn ook ‘concorruent’ garbage collectors die de CPU last meer over de tijd verspreiden. [↑](#footnote-ref-14)
15. Een unique\_ptr heeft wel een assignment operator en een copy constructor, maar die maken de ‘source’ pointer leeg (zetten ‘m op nullptr). [↑](#footnote-ref-15)
16. De scheiding is niet strict: als je een file naam meegeeft om te openen voor lezen maar die file bestaat niet, dan kan je dit onder beide categorien scharen. [↑](#footnote-ref-16)
17. Het hoeft geen pointer te zijn, als het maar de getoonde operaties (\*, ++, !=) biedt. [↑](#footnote-ref-17)
18. In de praktijk is het vaak handig om hiervoor smart pointers te gebruiken. [↑](#footnote-ref-18)
19. Bij het verwerken van enquete vragen die beantwoord zijn in een ordinale schaal (bv. goed, matig, slecht) kan je deze antwoorden vertalen naar een waarde (bv. 3, 2, 1), en dan het gemiddelde van die waarden nemen. Dit wordt heel vaak gedaan maar het is statisch volkomen onzinnig: Er wordt een *abitraire* vertaling gemaakt van een ordinale schaal naar een interval schaal (waarom niet goed=2, matig=1, slecht=-3000?). [↑](#footnote-ref-19)
20. Strikt genomen hoeven de onderlingen afstanden ze niet gelijk te zijn, als ze maar (zinnig) in een getal uit te drukken zijn. [↑](#footnote-ref-20)
21. Strik genomen: 10°C is 283°K. Kelvin wel een ratio schaal, dus 2 keer is 566°K, en dat is 293°C. [↑](#footnote-ref-21)