

Programmeren in C++ voor beginners

Richèl Bilderbeek

Copyright (C) 2007 en 2008
Richèl Bilderbeek

0. Dag 0

C++ begint op nul met tellen. Deze handleiding ook.

Deze handleiding begint te tellen vanaf nul en begint met een inhoudsopgave en zaken vooraf.

0.0. Inhoudsopgave

Programmeren in C++ voor beginners.....	1
0. Dag 0.....	2
0.0. Inhoudsopgave.....	3
0.0. Voorwoord.....	5
0.1. Over de cursus.....	6
0.2. Het C++ Panel.....	8
0.3. Thema's per dag.....	9
0.1. Aan te raden literatuur.....	10
0.1.0. Boeken.....	10
0.1.1. On-line boeken.....	10
0.1.2. Snel iets opzoeken.....	10
1. Dag 1.....	11
Een mogelijk eindprogramma van vandaag.....	11
1.0. Over C++.....	12
1.1. Over Kylix, C++ Builder en Turbo C++.....	13
1.2. Overzicht van de belangrijkste vensters.....	14
1.3. Een programma beheren.....	15
1.4. Een nieuw programma beginnen.....	15
1.5. Een programma opslaan.....	15
1.6. Een programma laden.....	15
1.7. Applicatie 0.....	16
1.8. Het Component Palette.....	17
Het Component Palette.....	17
Een Form met alle Standard Components.....	17
1.9. De Object Inspector.....	18
1.10. Events.....	19
1.11. Een Event programmeren.....	20
1.12. Properties run-time wijzigen.....	21
1.13. Properties lezen.....	23
1.14. Het if-statement.....	25
1.15. Compile fouten.....	29
1.16. Linker errors.....	31
1.17. Een echte applicatie.....	32
1.18. Volgende week.....	32
2. Dag 2.....	33
Een mogelijk eindprogramma van vandaag.....	33
2.0. Data types en variabelen.....	34
2.1. Scope van variabelen.....	37
2.2. Conversie van variabelen.....	39
2.3. Rekenen met int en double.....	41
2.4. De for-loop.....	42
2.5. Verkorte rekenschrijfwijze.....	43
2.6. Meer over for-loops.....	44
2.7. Speciale for-loops.....	44
2.8. Een truc.....	45
2.9. Het switch statement.....	45
2.10. Debuggen.....	47
2.11. Volgende week.....	49
3. Dag 3.....	50
Een mogelijk eindprogramma van vandaag.....	50
3.0. TStringGrid.....	51
3.1. Functies.....	51
3.2. De debugger en functies.....	54
3.3. Waar zet ik mijn functies in Kylix? Unit1.h?.....	54
3.4. assert.....	58
3.5. Volgende week.....	62
4. Dag 4.....	63
Een mogelijk eindprogramma van vandaag.....	63
4.0. const.....	64
4.1. Referencing.....	67
4.2. Referencing voor meerdere return-waarden.....	68
4.3. Reference-to-const.....	69

4.4.	References en pointers.....	70
4.5.	Pixel manipulatie.....	70
4.6.	Een plaatje vergroten.....	73
4.7.	Rekenen met unsigned char en int.....	74
4.8.	Volgende week.....	75
4.9.	Code van Skrienseefer.....	75
5.	Dag 5.....	77
5.0.	Een member variable.....	78
5.1.	Een std::vector.....	80
5.2.	Twee-dimensionale std::vector.....	84
5.3.	Access violation.....	85
5.4.	Bitblitting.....	86
5.5.	Volgende week.....	87
5.6.	Code PompebledRainer.....	87
5.6.0.	Unit1.h.....	88
5.6.1.	Unit1.cpp.....	88
6.	Dag 6.....	92
6.0.	Template functies.....	92
6.1.	STL Header files.....	94
6.2.	STL functies.....	96
6.3.	Random numbers.....	97
6.4.	Een eigen data type.....	98
6.5.	Double-buffering.....	99
6.6.	Volgende week.....	99
7.	Dag 7.....	100
7.0.	Een klasse.....	100
7.1.	Constructor.....	103
7.2.	Const methoden.....	105
7.3.	Klasse definitie en declaratie scheiden.....	107
7.4.	Verschil tussen class en struct.....	108
7.5.	De debugger.....	109
7.6.	Het laatste woord over de klasse Klok.....	109
7.7.	Volgende week.....	109
8.	Dag 8.....	110
8.0.	De TForm1 constructor.....	110
8.1.	Een TForm aanmaken.....	111
8.2.	De TForm1 constructor aanpassen.....	113
8.3.	Werken met meerdere verschillende Forms en/of Units.....	115
8.4.	Een console applicatie.....	116
8.5.	std::string.....	118
8.6.	En nu verder.....	120
9.	Appendix A: Een plaatje laten stuiteren.....	120
10.	Appendix B: Maar wat is dat dan?.....	122
11.	Appendix C: Sneltoetsen.....	124
12.	Appendix D: Overzicht.....	125
12.0.	Cast.....	125
12.1.	Code.....	125
12.2.	Compiler.....	125
12.3.	Data type.....	125
12.4.	Declaratie.....	125
12.5.	Definitie.....	126
12.6.	Event.....	126
12.7.	Functie.....	126
12.8.	Functie declaratie.....	126
12.9.	Functie definitie.....	126
12.10.	Functie prototype.....	127
12.11.	Generiek programmeren.....	127
12.12.	Header file.....	127
12.13.	Identifier.....	127
12.14.	Implementatie bestand.....	127
12.15.	Initialiseren.....	128
12.16.	Klasse.....	128
12.17.	Klasse declaratie.....	128
12.18.	Klasse definitie.....	128
12.19.	Lidfunctie.....	129
12.20.	Methode.....	129
12.21.	Methode declaratie.....	129
12.22.	Methode definitie.....	129
12.23.	Procedure.....	130
12.24.	Referentie.....	130
12.25.	STL.....	130

12.26.	Unit.....	130
12.27.	Variabele.....	130
12.28.	Variabele declaratie.....	131
12.29.	Variabele definitie.....	131

0.0. Voorwoord

Er zijn veel manieren om een cursus C++ programmeren op te zetten: begin je eerst met de C syntax, de C++ klassen of een grafische bibliotheek? Is er al materiaal aanwezig dat moet worden hergebruikt? En wordt er een boek gevolgd?

Deze cursus is de eerste cursus die ik helemaal vanuit mijn eigen filosofie heb opgezet.

Ja, we beginnen met grafische dingen, ook al weten we niet precies wat er achter de schermen gebeurt. Dit is ook niet belangrijk: er is een (hopelijk) visueel aantrekkelijk resultaat en ooit worden deze details vanzelf duidelijk.

Nee, er wordt geen boek gevolgd. Het best aansluitende boek (Sam's Teach Yourself C++ Builder in 24 Hours) is niet meer verkrijgbaar. Inplaats daarvan worden er boeken gevolgd. Wel volg ik de wijsheid van de beste C++ boeken op.

Deze cursus heb ik 'alleen' opgezet. En alleen tussen aanhalingstekens, want er waren altijd genoeg mensen die in gedachten over mijn schouders meekeken. Dit waren Bjarne Stroustrup, Herb Sutter, Scott Meyers, Andrei Alexandrescu, Andrew Koenig, Herbert Schildt, Jesse Liberty, Barbera Moo, Bruce Eckel, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Marshall Cline, Greg Lomow, Mike Girou. Als tegengewicht keken veel code monkeys (in de negatieve betekenis) mee, die ik heb samengevoegd in de jonge Bill Gates.

Ook Momo en Spike waren er altijd om me te helpen mijn gedachten te ordenen. En ook Persephone, ook al was dit de hond van Scott Meyers.

Veel plezier met de cursus,

Richèl Bilderbeek

0.1. Over de cursus

Deze cursus is opgezet met de volgende hoofdgedachten:

- * De Joost methode
- * Practisch
- * Voor iedereen te begrijpen
- * Geen huiswerk verplicht
- * Geen boeken verplicht
- * Leuk

De cursus werkt volgens de Joost methode. Deze methode is mijn eigen favoriete methode om iemand C++ te leren en is vernoemd naar de eerste persoon die ik op deze manier heb leren programmeren. Deze methode houdt in dat we eerst leren met de programmeeromgeving te werken, zonder te snappen wat er eigenlijk gebeurt. Naarmate de cursus vordert, vallen de stukjes vanzelf op hun plaats. Hierdoor is deze cursus vergeleken met een doorsnee cursus 'achterstevoren': de laatste dag pas leren we pas een console/DOS applicatie te maken!

De cursus is praktisch opgezet: elke dag werkt toe naar een bepaald eindprogramma. Alles wat hiervoor niet relevant is, wordt later of niet behandeld.






De cursus is voor iedereen te begrijpen. Het tempo is zo laag mogelijk gehouden en de uitleg zo simpel mogelijk. Dit houdt niet in dat als je het begrijpt, dat je dan goed kunt programmeren. Dit vereist immers een bepaalde logische manier van denken, die elke niet-programmeur moet ontwikkelen.

De cursus verplicht geen huiswerk. Het is de bedoeling dat elke dag voldoende is om haar thema's te behandelen. Om snel een goed programmeur te worden, is het wel nodig veel ervaring op te doen. Ook kan er dan meer uit de cursus worden gehaald.

De cursus verplicht geen boeken. Elke dag is er een uitleg van hoogstens drie kwartier, daarna koffie en daarna kan er zelf geprogrammeerd worden. Deze cursusgids bevat de kern van wat er behandeld wordt, maar is bij lange na niet volledig. Dit is ook niet erg: op het Internet is alles te vinden. Ook staan er goede (Engelse) C++ boeken online. Voor degenen die graag van papier lezen, zijn er zowel goede Nederlandse als Engelse boeken. Zie 'Aan te raden literatuur' voor aanbevolen literatuur.

Belangrijkste gedachte achter de cursus is dat 'ie leuk moet zijn. Omdat ik persoonlijk het het leukst vind om 'domme' spelletjes te maken, gebruik ik deze ook als voorbeeldprogramma's. En het programmeren van spelletjes is net zo leerzaam als het programmeren van een 'serieuzere' applicatie. Het gaat immers om de ervaring die wordt opgedaan.

0.2. Het C++ Panel

	<p>Bjarne Stroustrup heeft C++ ontwikkeld. Hij vertelt soms zijn idee achter een bepaald concept.</p>
	<p>Herb Sutter is auteur met Andrei Alexandrescu van 'C++ Coding Standards'. Hij geeft advies welke stijl wel of niet raden is.</p>
	<p>Andrei Alexandrescu heeft niet alleen 'C++ Coding Standards' geschreven met Herb Sutter, maar ook 'Modern C++ Design'. Andrei is gek op templates, een van de latere onderwerpen. Als Herb het te druk heeft, geeft hij advies over programmeerstijl.</p>
	<p>Scott Meyers is auteur van onder andere 'Effective C++'. Hij geeft advies over het ontwerp van een goed programma.</p>
	<p>Andrew Koenig is een praktisch man en samen met Barbara Moo auteur van 'Accelerated C+'. De Koenig-look-up is naar hem vernoemd. Hij geeft praktische adviezen, soms zelfs over Kylix, waar hij in het echt niets van af weet.</p>
	<p>Bill Gates is een praktisch man. Hij geeft advies hoe het <u>niet</u> moet, maar wat je wel soms in code aantreft. Soms geeft 'ie ook zijn mening, die je soms ook terugziet in code. Hij heeft nooit het laatste woord.</p>

0.3. Thema's per dag

Dag	Pure C++	Kylix, CLX	Programma
1	if	IDE, Component Palette, Object Inspector, TForm, TTimer, TImage, ShowMessage	Stuiterende voetbal
2	data typen, conversies, switch, for	String, debuggen	Gorilla
3	Functies I, #include, assert	TStringGrid	Magisch vierkant
4	Functies II, const , references	TImage::SetPixel	Screensaver (met flikkering)
5	std::vector, lid variabelen	TCanvas::Draw	Screensaver (zonder flikkering), Tank
6	template functies, struct , std::rand, std::cos, std::swap	Double buffering	Alles
7	class , constructor, methoden		Alles
8	std::cout, std::string	Project Manager, meerdere Units/Forms	Alles

0.1. Aan te raden literatuur

0.1.0. Boeken

- * Aan de slag met C++,
- * C++ in 24 hours, Jesse Liberty
- * C++ from the ground up, Herbert Schildt

0.1.1. On-line boeken

- * The C++ Annotations, Frank Brokken:
<http://www.icce.rug.nl/documents/>
- * Thinking in C++, Bruce Eckel:
<http://www.ibiblio.org/pub/docs/books/eckel/>
- * Google:
<http://www.google.com>, zoek op 'Free C++ books'

0.1.2. Snel iets opzoeken

- * Mijn site:
<http://www.richelbilderbeek.nl/Cpp.htm>
- * CodePedia:
<http://www.codepedia.com/Cpp>
- * Google:
<http://www.google.com>

1. Dag 1



Een mogelijk eindprogramma van vandaag.

Vandaag gaan we leren werken met Kylix. We leren componenten kennen. Componenten hebben eigenschappen en gebeurtenissen. Eigenschappen kunnen we bij bijvoorbeeld een muisklik (een gebeurtenis) wijzigen. Door creatief te spelen met deze eigenschappen kunnen we een plaatje over het beeldscherm laten stuiteren. Als enige 'pure C++' wordt het **if** statement geleerd.

1.0. Over C++

C++ is een oprogrammeertaal ontwikkeld in 1983 (!) waarin alle denkbare programma's in geprogrammeerd kunnen worden. Bijvoorbeeld Windows XP en Vista zijn grotendeels in C++ geprogrammeerd, maar ook het schietspel Quake.

Ook is C++ een multi-paradigmataal: een programmeur kan elke denkbare (soms 'vieze') taalconstructie doen, omdat C++ geen werkwijze oplegd. In C++ kan 'object georiënteerd' worden geprogrammeerd, maar dit hoeft niet.



Ik laat je helemaal vrij. Wel geef ik je de mogelijkheid schonere code dan in C te schrijven.

Daarnaast is de C++ syntax (zinsbouw) kort met vaak leestekens in plaats van woorden. De layout van C++ is vrij. C++ is case gevoelig.

C++	BASIC
for (int i=0; i!=10; ++i) { //Doe dingen }	for i = 0 to 10 step 1 rem doe dingen next

Vergelijking syntax van C++ met BASIC

Standaard C++ is relatief beperkt: er kan alleen met tekst (één kleur, zonder knippereffecten) gewerkt worden. Om met kleur te kunnen werken zijn er vele bibliotheken ontwikkeld, waaronder de CLX, die door Kylix wordt gebruikt.



C++ is ontwikkeld om platform onafhankelijk te zijn. C++ neemt zelfs niet aan dat de gebruiker een toetsenbord of beeldscherm heeft!

1.1. Over Kylix, C++ Builder en Turbo C++

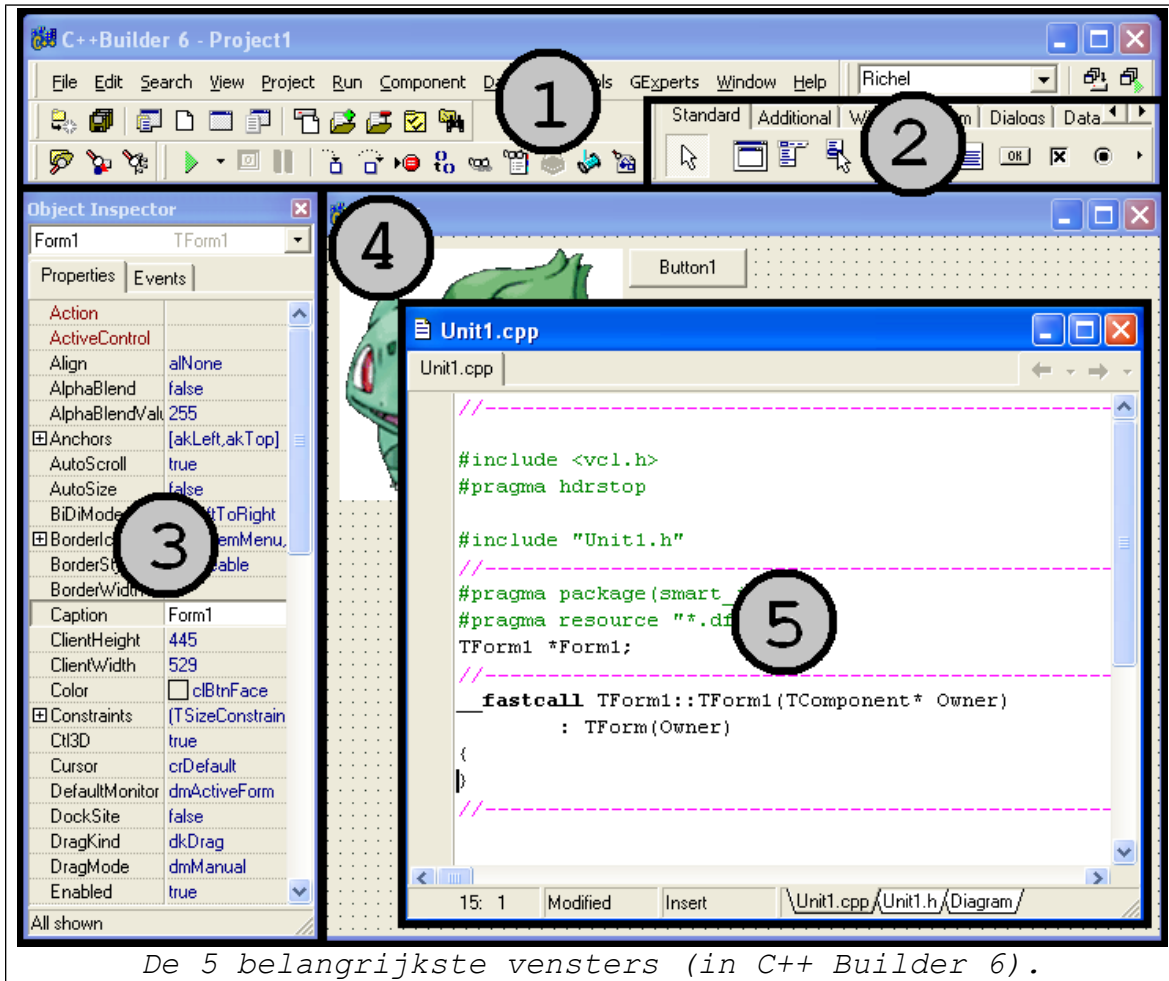
Kylix, C++ Builder en Turbo C++ zijn alledrie door Borland (vroeger Inprise, nu CodeGear) ontwikkeld. Ze zien er hetzelfde uit, hebben allemaal de C++ standaardbibliotheek (STL) meegeleverd en de twee grafische Borland bibliotheken (CLX en VCL) zijn bijna identiek.

	Kylix	C++ Builder	Turbo C++ 2006
Platform	Linux	Windows	Windows
Programmeertalen	C++ en Delphi	C++	C++
Beschikbaarheid	Niet meer	Een maand gratis, dan kopen	Gratis
Mogelijkheden	Meer	Meest	Veel
Bibliotheken	CLX	VCL en CLX	VCL

Een grafische programmeeromgeving helpt de programmeur met programmeren. In deze cursus wordt zwaar van deze hulp gebruik gemaakt.

1.2. Overzicht van de belangrijkste vensters

Bij het starten van Kylix zijn er meteen veel vensters te zien. De 'Object Treeview' en 'Project Manager' hebben we niet nodig, dus die kunnen we weggklikken. Dan blijven er nog 5 vensters over.



De 5 belangrijkste vensters (in C++ Builder 6).

Waar	Wat	Waarvoor	Sneltoets
1	Menubalk	Menu opties	Alt
2	Component Palette	Bevat alle componenten	Geen
3	Object Inspector	Bekijken van een component	F11
4	Form	'Werkblad', design	F12
5	Code Editor	Schrijven code	F12

1.3. Een programma beheren

Er is wat bestandsbeheer en discipline nodig om goed en zonder problemen met je programma's te kunnen werken. Gebruik onderstaande richtlijnen tot je doorhebt wat er gebeurt als je je niet aan deze richtlijnen houdt.

1.4. Een nieuw programma beginnen

Als je een nieuw programma wilt maken, doe **ALTIJD** eerst 'File | Close All'.

1.5. Een programma opslaan

Kies 'File | Save All' of CTRL-SHIFT-S. Er gaan nu twee bestandsnamen gevraagd worden: de naam van het Project en de naam van de Unit. Deze moeten **NIET** dezelfde naam krijgen. Leer je aan de naam van je project met 'Project' en de naam van je unit met 'Unit' te laten beginnen, bijvoorbeeld 'Project1' en 'Unit1'.

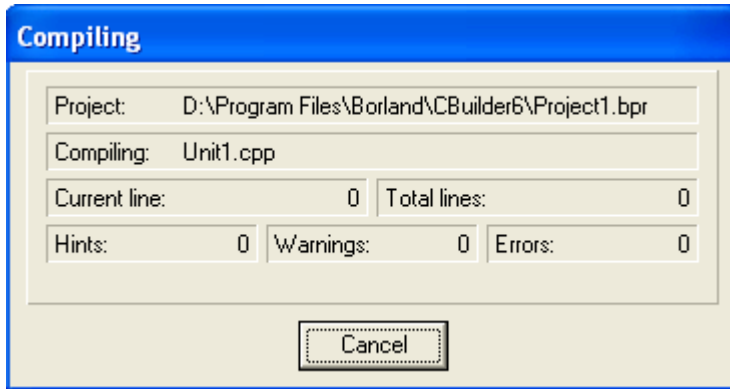
1.6. Een programma laden

Als je een nieuw programma wilt laden, doe **ALTIJD** eerst 'File | Close All'. Kies dan **ALTIJD** 'File | Open Project'.

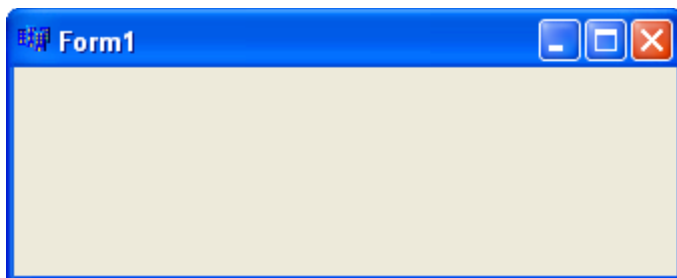
1.7. Applicatie 0

Je eerste applicatie maken is simpel. Begin een nieuw programma. Doe **ALTIJD** eerst 'File | Close All'. Druk op 'Run | Run' of F9.

Kort verschijnt dit schermpje:



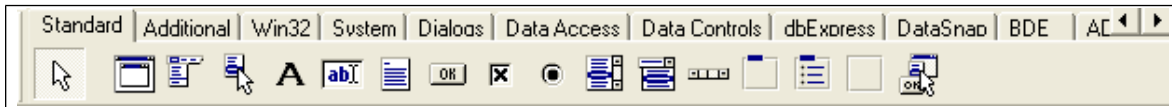
Daarna komt 'Applicatie 0' tevoorschijn:



Hoe weinig spectaculair ook, dit is een volwaardige windowsachtige applicatie.

1.8. Het Component Palette

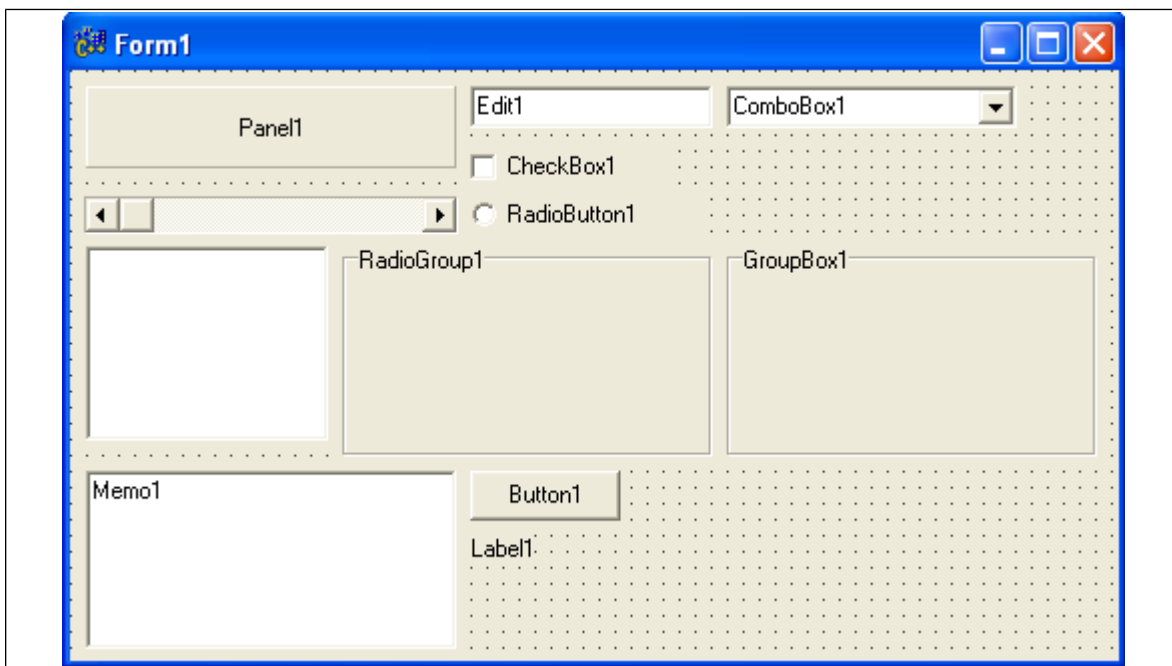
Het Component Palette bevat Componenten. Een Component is 'een ding dat je met de Object Inspector kunt veranderen'. Elk Component hoort bij of op een Form, behalve Form zelf.



Het Component Palette

Klik op een Component, bijvoorbeeld een Button (onder de tab 'Standard' en klik dan op het Form. Er staat nu een knop op het Form. Druk op 'Run | Run' of F9 om deze iets uitgebreidere applicatie te starten. Op de knop kan gedrukt worden, maar hij doet nog niets.

Kijk eens welke Components er allemaal aanwezig zijn en welke nuttig zouden kunnen zijn.



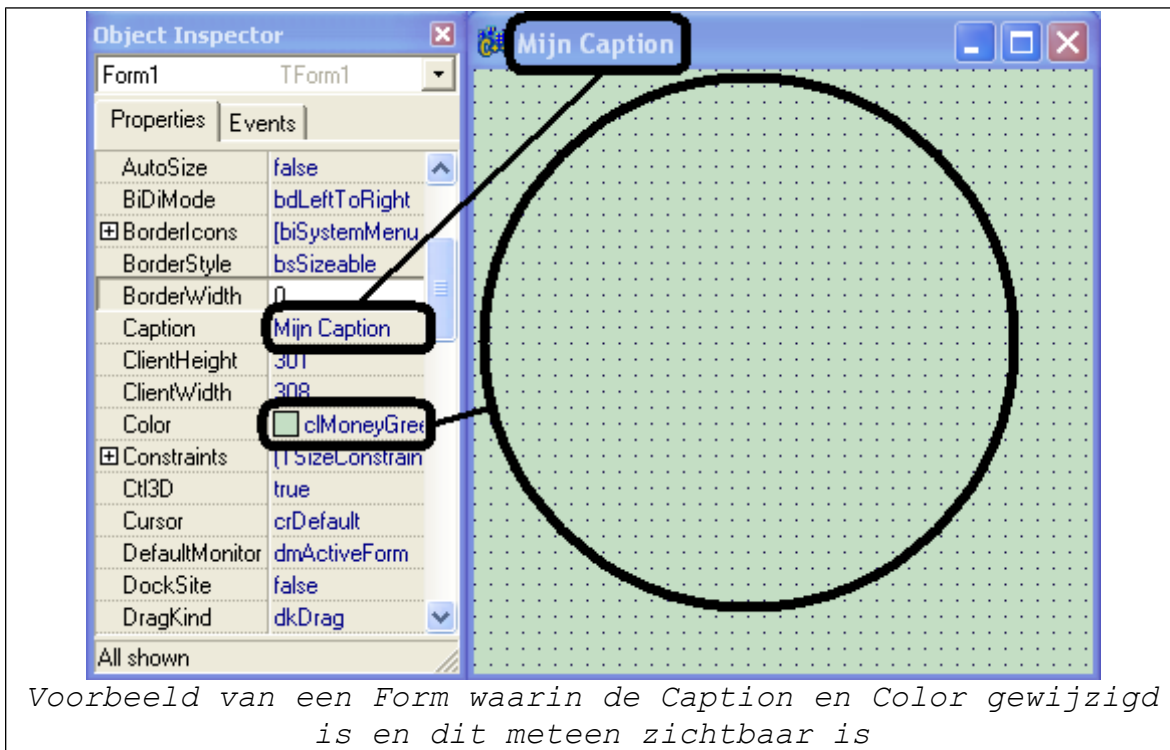
Een Form met alle Standard Components

1.9. De Object Inspector

Met de Object Inspector kan er 'design time' geprogrammeerd worden. Klik op het Form. In de Object Inspector verschijnt dan bovenaan de tekst 'Form1'. In het tabblad 'Properties' staan de eigenschappen van dit Component.

Sommige eigenschappen zijn woorden (bijvoorbeeld Caption), andere getallen (Width), weer andere zijn 'true' of 'false' (Visible) en bij sommigen verschijnt zelfs een pop-up menu (Color).

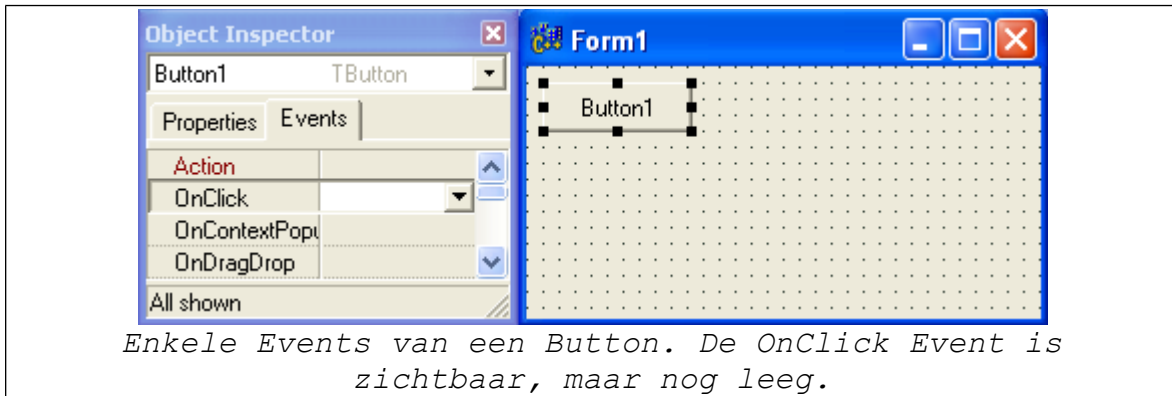
Maar wat je ook kiest, het Form past zich meteen aan de nieuwe waarde aan als dit nuttig/mogelijk is.



Alle Components kunnen per definitie tijdens 'design-time' worden gewijzigd. Omdat Components verschillen, verschillen ook hun Properties.

1.10. Events

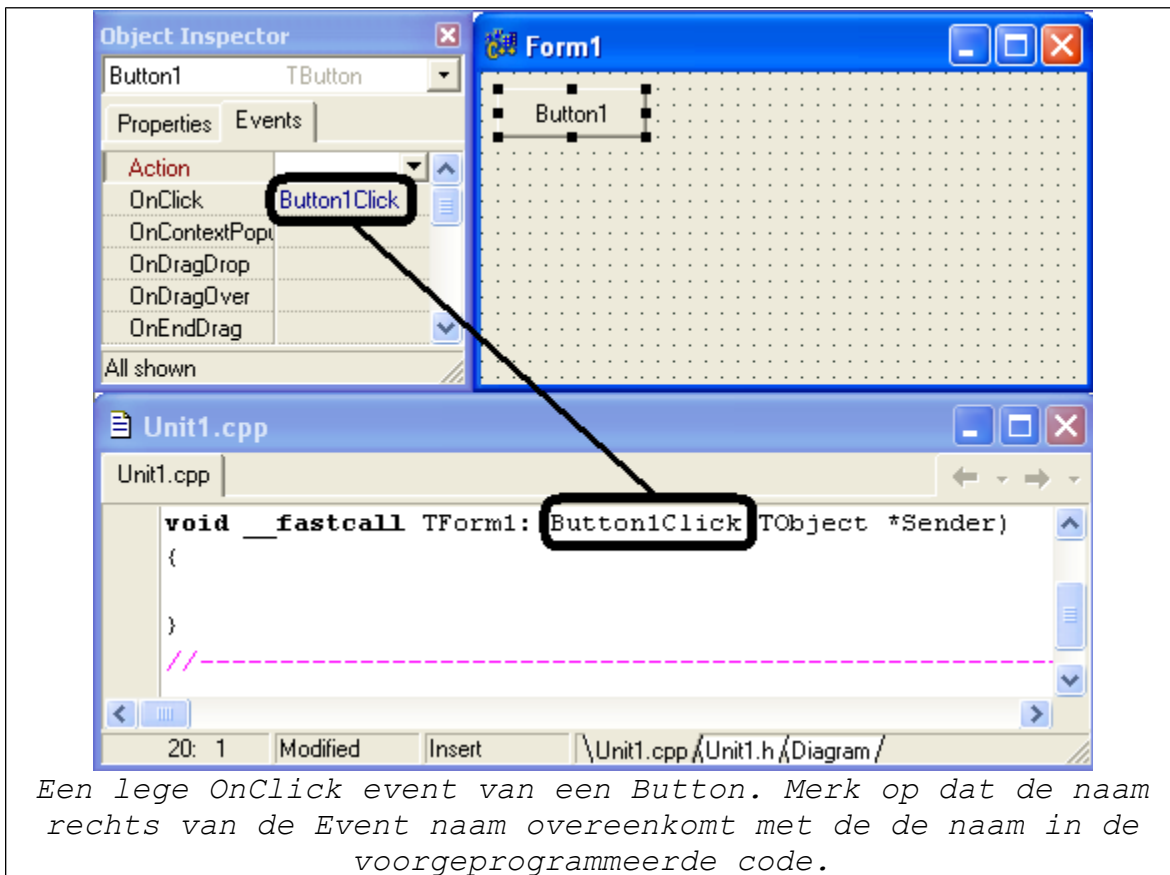
Programma's reageren op gebeurtenissen. Dit worden in Kylix 'Events' genoemd. De bekendste Event is de 'OnClick' Event van een Button.



Onder de Events komt de C++ code te staan van wat er bij een Event moet gebeuren.

1.11. Een Event programmeren

Selecteer in de Object Inspector een Component. Ga naar het tabblad 'Events'. Dubbelklik op het lege hokje rechts van de Event naam, bijvoorbeeld 'OnClick' van een Button. De Code Editor verschijnt met een stuk voorgeprogrammeerde code.



De code tussen de accolades wordt aangeroepen als op de knop geklikt wordt. De rest van de 'moeilijke' regel wordt later duidelijk. De regel die begint met '//' is een commentaar regel.

Nu kunnen Properties run-time worden gewijzigd.

1.12. Properties run-time wijzigen

Het wijzigen van Properties onder run-time is iets moeilijk dan tijdens design-time.

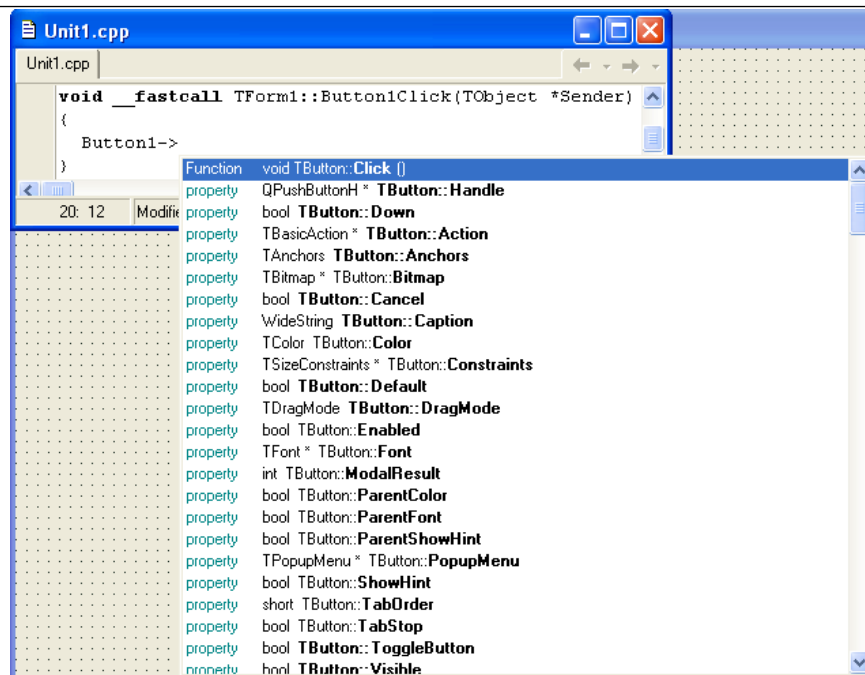
Het duidelijkst is een voorbeeld:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    //Deze regel is een commentaar regel

    //Verander de Caption van Button1 naar de tekst 'Hallo'
    Button1->Caption = "Hallo";

    //Zet Button1 100 pixels links van de rand van het Form
    Button1->Left = 100;

    //Zet de kleur van Form1 op groen, oftewel clLime
    Form1->Color = clLime;
}
```



Als je 'Button1->' toets, verschijnt een pop-up menu met alle mogelijke 'dingen' die je kunt kiezen. Dit wordt 'Class Browsing' genoemd en voorkomt veel typefouten.

Merk de volgende dingen op:

- * Commentaar begint met een dubbele delen-door-streep, oftewel een dubbele slash. De rest van de regel wordt niet gelezen door de computer
- * Elke 'echte' regel eindigt met een puntkomma
- * Eerst wordt de naam van het Component gegeven, dan een pijltje, dan de Property, dan een is-teken, dan de nieuwe waarde van de Property, dan een puntkomma.
- * Is de type van het Property tekst, bijvoorbeeld een Caption, dan moet de tekst tussen dubbele aanhalingstekens.
- * Getallen en 'speciale woorden' hebben geen aanhalingstekens. Deze 'speciale woorden' worden later 'enumeraties' genoemd en zijn eigenlijk getallen in een wat leesbaarder jasje. De naam van deze enumeraties kunnen ook uit de Object Inspector worden gelezen.

Bovenstaande code is al 'echte' C++ code. Er kunnen dan ook al dingen misgaan. Dit wordt een compile error genoemd.

Merk op dat deze code alleen maar Properties toekent aan Components, in plaats van deze te lezen.

1.13. Properties lezen

Properties kunnen ook gelezen worden om vervolgens ergens anders weer geschreven te worden.

Hier onder een voorbeeld met een Form met twee Buttons, elk met een eigen OnClick Event.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    //Button 1 is geklikt
    //Wissel de Captions van Button1 en Button2 om
    //Gebruik de Caption van Form1 als 'bruggetje'
    Form1->Caption = Button1->Caption;
    Button1->Caption = Button2->Caption;
    Button2->Caption = Form1->Caption;
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{
    //Button 2 is geklikt
    //Laat Button2 naar rechts lopen
    Button2->Left = Button2->Left + 10;
}
```

Merk de volgende dingen op:

- * C++ heeft een vrije layout. Daarom moet met een puntkomma aan worden gegeven waar een zin eindigd. Bij Button1Click heb ik de is-tekens mooi onder elkaar gezet. Dit verhoogt (in mijn ogen) de leesbaarheid van de code. Leesbaarheid gaat (meer) een rol spelen bij moeilijkere code.
- * Om Button2 naar rechts te laten lopen, wordt eerst de Property Left gelezen, hier tien bij opgeteld en dit getal wordt weer naar de Left van dezelfde Button weggeschreven. Later leren we een verkorte schrijfwijze.
- * Een Caption van "10" kan niet toegekend worden aan een Left Property. Een Caption is een woord en Left is een getal. Op Dag 2 zien we hoe we deze types naar elkaar kunnen converteren. Soms gaat zo'n conversie toch automatisch, bijvoorbeeld als je een Left aan een Caption wilt toekennen. Dit is dan een keuze van de Borland programmeurs geweest.

Het plus-teken ('+') wordt een 'wiskundige operator' genoemd. C++ kent er meerdere, hier de belangrijkste:

Wat	Waarvoor	Voorbeeld	Uitkomst
+	Optellen	16 + 10	26
-	Aftrekken	16 - 6	10
/	Delen	16 / 5	3
*	Vermenigvuldigen	16 * 4	64
%	Rest bij deling	16 % 5	1
!	Niet	!true	false

Nu kunnen we Components laten bewegen, maar voor een leuk programma is het nog nodig wat-als-dan-statement, oftewel if-statements, te kunnen maken.

1.14. Het if-statement

De syntax van een if-statement is als volgt:

```
if ( /* iets */ )  
{  
  
}
```

Of

```
if ( /* iets */ )  
{  
    //Doe dit als 'iets' waar is  
}  
else  
{  
    //Doe dit als 'iets' niet waar is  
}
```

De '/* iets */' is een multi-line commentaar (hoewel deze hier toch single-line is) en dit is een commentaar dat begint bij '/*' en eindigt bij de '*/'.

Op de plaats van '/* iets */' moet een conditie komen te staan.

```
//Is Button1 te ver naar rechts?  
if ( Button1->Left > 100 ) //Hier GEEN puntkomma's!!!  
{  
    //Zet Button1 weer naar links  
    Button1->Left = 0;  
}  
else  
{  
    //Laat Button1 naar rechts lopen  
    Button1->Left = Button1->Left + 10;  
}
```

Merk het volgende op:

- * Een conditie staat tussen ronde haken.
- * Na een conditie staat geen puntkomma.
- * Na de ronde haken komt geen puntkomma.
- * Een single-line commentaar kan ook op dezelfde regel van 'echte code' staan.
- * Het groter-dan teken in het if-statement ('>') leest als 'groter-dan'. De conditie lees je dan ook als 'als de Left van Button1 groter is dan 110, dan ...'.
- * Tussen de accolades van een if-statement springt de code naar rechts. Dit is om de leesbaarheid te vergroten.



Er zijn meerdere manieren van inspringen. Gebruik een consistente manier van inspringen, zoals Richel. Sommige programmeerteams verbieden het gebruik van tabs om in te springen.

Er zijn meerdere relaties tussen twee dingen dan een groter-dan relatie:

Relatie	Teken
Gelijk	==
Ongelijk	!=
Groter	>
Kleiner	<
Groter of gelijk	>=
Kleiner of gelijk	<=

if-statements kunnen ook aan elkaar gekoppeld worden:

```
if ( Edit1->Text == "Richel"
    && Edit2->Text == "Bilderbeek") //Mooie layout
{
    //Als beide waar zijn
    Form1->Caption = "Welkom Richel!";
    ButtonSuperGeheim->Visible = true;
}
else
{
    Form1->Caption = "Haha! Jij komt niet verder!";
}

if ( Button1->Left < -100
    || Button1->Left > 1000)
{
    //Als een van beide waar is
    Form1->Caption = "Oei, Button1 is het scherm uit!";
}
```

Merk de volgende dingen op:

- * 'en' wordt geschreven als '&&', ofwel een dubbele ampersand.
- * 'of' wordt geschreven als '||', ofwel een dubbele pijp.
- * C++ layout is vrij, dus beide condities kunnen ook op een regel. In dit document paste dit niet.
- * Tussen de accolades van een if-statement springt de code naar rechts. Dit is om de leesbaarheid te vergroten.
- * Een Button kan 'ButtonSuperGeheim' heten als de Name Property zo heet.

Ook kan een **if**-statement weer in een **if**-statement staan:

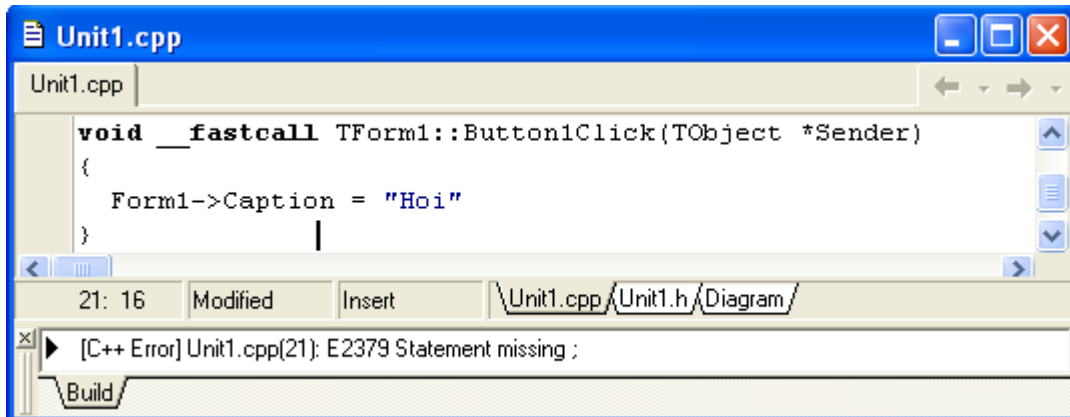
```
if ( Edit2->Text == "Bilderbeek")
{
    //Een familielid
    if (Edit1->Text == "Richel")
    {
        //Ikzelf
        Form1->Caption = "Welkom Richel!";
        ButtonSuperGeheim->Visible = true;
    }
    else
    {
        //Een ander familielid
        Form1->Caption = "Welkom!";
        ButtonFamilieOnly->Visible = true;
    }
}
else
{
    //Geen familielid
    Form1->Caption = "Haha! Jij komt niet verder!";
}
```

Merk het volgende op:

- * Tussen de accolades van een if-statement springt de code naar rechts. Dit is om de leesbaarheid te vergroten.
- * De Property Visible heeft in de Object Inspector alleen de mogelijke waarden 'true' en 'false'. De Code Editor gebruikt een **bold** font voor deze woorden omdat dit officiële C++ sleutelwoorden zijn. Een waarde als bijvoorbeeld 'clLime' is geen C++ sleutelwoord.

1.15. Compile fouten

Bij het programmeren worden er soms typefouten gemaakt, waardoor de computer 'het niet meer snapt'. De mate hoe duidelijk de foutmelding is, hangt af van de fout.

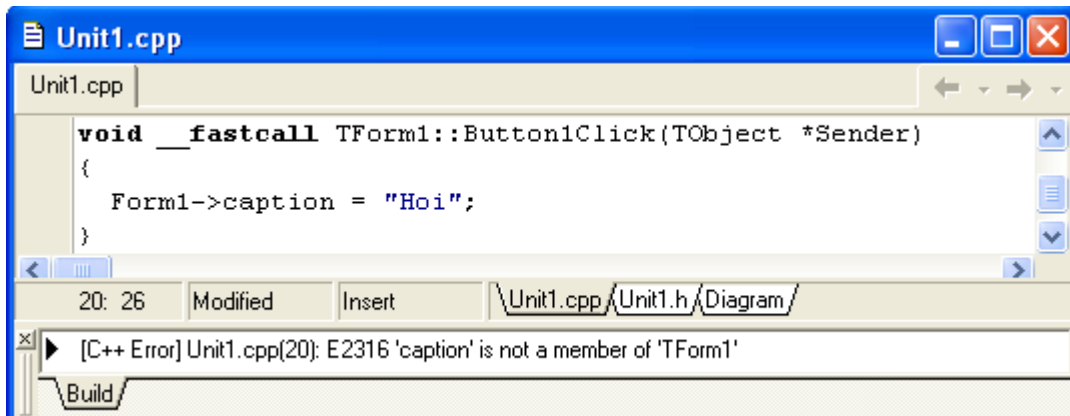


```
Unit1.cpp
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Form1->Caption = "Hoi"
}
```

21: 16 Modified Insert \Unit1.cpp\Unit1.h\Diagram/

[C++ Error] Unit1.cpp(21): E2379 Statement missing ;

Build

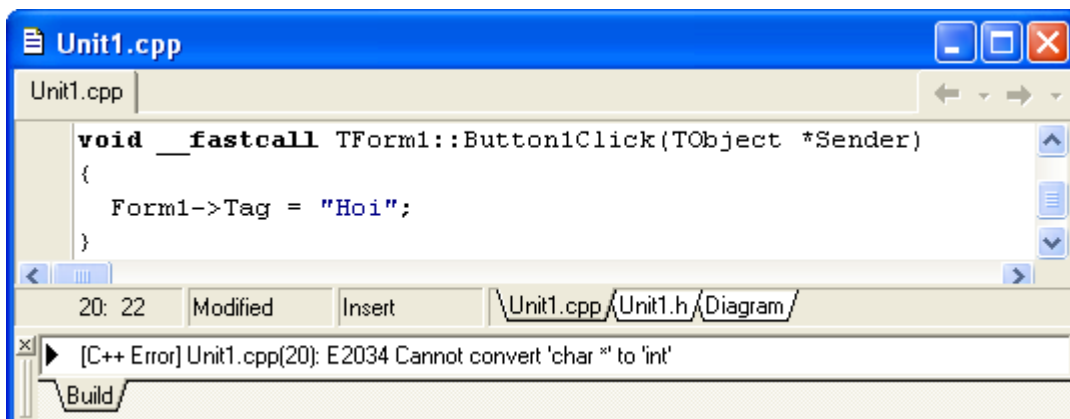


```
Unit1.cpp
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Form1->caption = "Hoi";
}
```

20: 26 Modified Insert \Unit1.cpp\Unit1.h\Diagram/

[C++ Error] Unit1.cpp(20): E2316 'caption' is not a member of 'TForm1'

Build



```
Unit1.cpp
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Form1->Tag = "Hoi";
}
```

20: 22 Modified Insert \Unit1.cpp\Unit1.h\Diagram/

[C++ Error] Unit1.cpp(20): E2034 Cannot convert 'char *' to 'int'

Build

De compiler (datgeen wat de foutmelding vind) heeft **ALTIJD GELIJK**. Er is altijd een logische reden waarom deze zijn werk niet kan doen. Leer de feedback van de compiler waarderen en herkennen.



Geef de voorkeur aan compile-time fouten boven run-time fouten. Oftewel: wees blij dat je compiler je fout ontdekt, inplaats van dat jij die moet gaan zoeken.

Soms geeft de compiler een waarschuwing. Zorg dat deze waarschuwing verdwijnt, want waarschijnlijk heeft de compiler een typefout van je ontdekt.



Compileer schoon, dus zonder waarschuwingen op het hoogste warning level.

Er is een uitzondering: als het geheugen van de computer 'in de war is' (programmeurs veroorzaken dit vaker dan gebruikers). Save het programma, 'File | Close All', 'File | Open Project' en probeer opnieuw. Sluit anders Kylix een keer af. In het ergste geval: herstart de computer. Is de fout nog steeds aanwezig, dan heeft de compiler echt gelijk.



Zit je te lang naar een fout te staren, vraag dan iemand om een frisse blik of neem een pauze.

1.16. Linker errors

Als je Kylix een Event aan laat maken, dan moet je Kylix deze ook zelf op laten ruimen. Als je de tekst in een Event weghaalt, dan verwijdert Kylix de Event.

Verwijder je met de hand een Event, dan krijg je onderstaande foutmelding (met een OnClick Event van Button1):

```
[Linker Error] Unresolved external '__fastcall  
TForm1::Button1Click(System::TObject *)' referenced from  
D:\PROGRAM FILES\BORLAND\CBUILD6\UNIT1.OBJ
```

Ga naar 'Unit1.h' met behulp van CTRL-F6 of op het tabblad onderaan de Text Editor te klikken. Zoek naar de genoemde regel, in dit geval 'void __fastcall Button1Click(TObject *Sender);' en haal deze weg.

1.17. Een echte applicatie

Nu weten we genoeg om met wat creativiteit een echte applicatie te bouwen!

Enkele tips:

- * Een Timer is een Component die een OnTimer Event heeft. Deze OnTimer Event gaat elke ingestelde tijd (zelf uitzoeken!) af. Nuttig voor animaties.
- * De Property Tag is handig om getallen in op te slaan, want deze wordt nergens voor gebruikt. Tag is hier zelfs voor bedoeld!
- * Kom je een 'opslagplaats' te kort, maak dan bijvoorbeeld een extra Label aan. Dan kun je dit Label eventueel onzichtbaar maken.

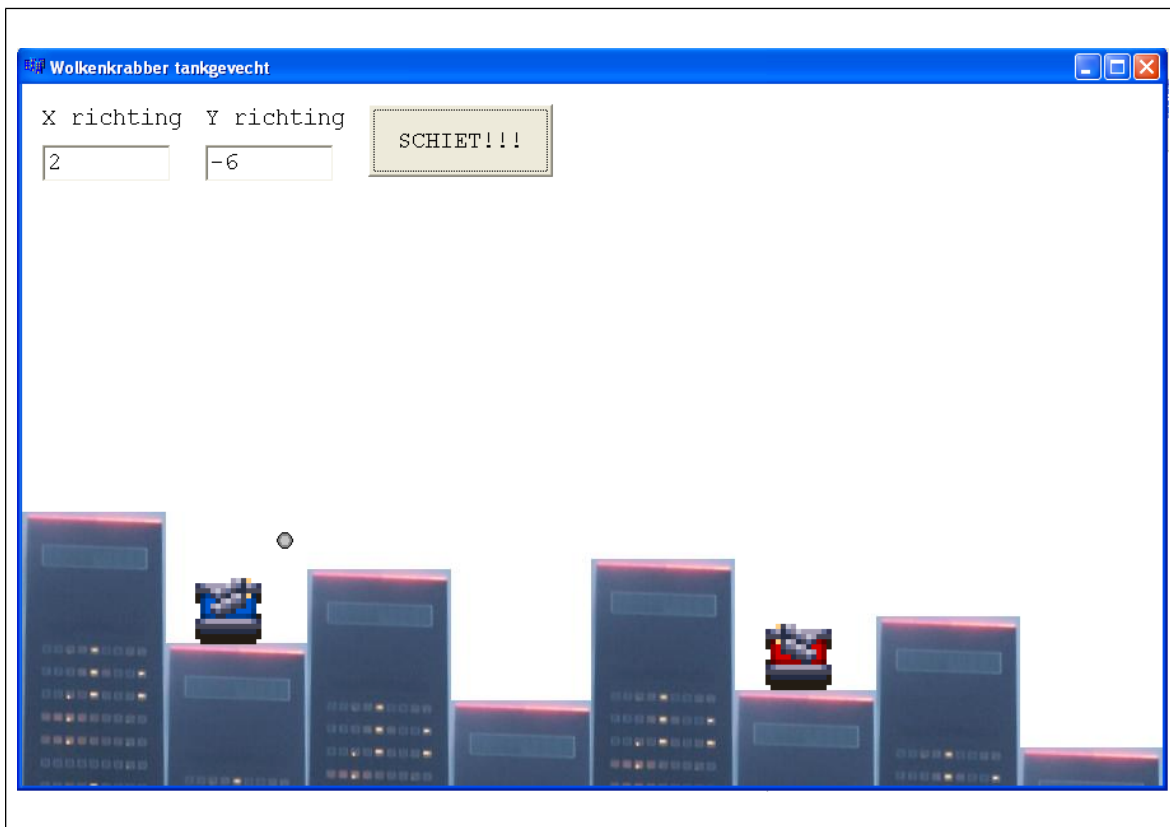
Enkele applicaties die nu mogelijk zijn:

- * Een plaatje laten stuiteren, zie eventueel Appendix A voor een voorbeeld.
- * Een kluiscombinatieslot programma
- * Memory
- * Drie op een rij

1.18. Volgende week

Volgende week leren we hoe we getallen naar woorden kunnen converteren en vice versa. Ook leren we hoe deze data typen officieel heten. Ook leren we het **switch** en **while** statement en een manier om onze eigen 'Events' te maken.

2. Dag 2



Een mogelijk eindprogramma van vandaag.

Nu we kunnen werken met Kylix is het tijd te leren werken met de verschillende data typen. We hebben al gezien dat er woorden, getallen en andere 'dingen' als Color bestaan. We zullen zien hoe we een woord en getal kunnen bewaren en deze naar elkaar te converteren. Ook leren we twee broertjes van het **if**-statement kennen: **switch** en **for**. Ook leren we met de debugger onze code te controleren.

2.0. Data types en variabelen

Een data type is het type van een waarde. Een naam is bijvoorbeeld een woord, in plaats van een getal. Het getal `pi` is een gebroken waarde.

C++ is een type-safe taal. Dit houdt in dat een getal niet achter de schermen om geconverteerd kan worden en omgekeerd. Dit lijkt op het eerste gezicht onhandig, maar dit een belangrijke barriere tegen programmeerfouten. En in een goed niet-triviaal programma zullen ook relatief weinig conversies zitten. Ook een voordeel aan data types is dat de compiler (datgeen wat code omzet naar een werkend programma) kan optimaliseren op snelheid.

C++ kent een aantal primaire datatypen en uitbreidingen hiervan. De primaire datatypen verschijnen dikgedrukt in de Code Editor, bijvoorbeeld **char**. **char** staat voor 'character' en kan een enkele letter opslaan. Een programmeur kan van deze typen zijn eigen data type maken, bijvoorbeeld een `String`. `String` is een verzameling **chars** en is zelf niet dikgedrukt.

Hieronder een lijstje van enkele data types:

Data type	Omschrijving
bool	Booleaanse waarde: true of false
char	Karakter
int	Geheel getal
double	Gebroken getal
AnsiString	Woord, zin of een geheel tekstdocument

Inplaats van AnsiString kun je net zo goed 'String' geschrijven.

Elke waarde, ook wel variable genoemd, heeft een data type. Hieronder staat hoe je een variabele aanmaakt:

```
//Algemene syntax
// [data type] [identifier];
// [identifier] = [waarde];

//Declareer een AnsiString met de naam 'naam'
String naam;

//Definieer de waarde van deze variabele
naam = "Richel";

//Zet deze Caption van Form1 op deze naam
Form1->Caption = naam;
```

Een declaratie is het vertellen aan de compiler dat je een variabele van een bepaald data type wilt gaan gebruiken. Deze variabele-naam wordt ook wel identificer genoemd. Een definitie is het toekennen van een waarde aan een variabele. Een declaratie en definitie kunnen ook in een keer, met een ander voorbeeld:

```
//Algemene syntax
// [data type] [identificer] = [waarde];

//Declareer en defineer een integer
int leeftijd = 27;

//Zet de Tag van Form1 op deze leeftijd
Form1->Tag = leeftijd;
```



Initializeer je variabelen altijd.

Bovenstaande code bestaat uit statements, ofwel 'dingen die gebeuren' en staan altijd ergens tussen twee accolades. Bijvoorbeeld de accolades van een Event.

Onderstaande code zorgt ervoor dat de Captions van twee Buttons omgewisseld worden:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    String s1 = Button1->Caption;
    String s2 = Button2->Caption;
    Button1->Caption = s2;
    Button2->Caption = s1;
}
```

2.1. Scope van variabelen

Elke variabele heeft een levensduur, ook wel scope genoemd. Een variabele bestaat pas nadat deze gedeclareerd is en bestaat tot een accolade.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    //Hier bestaat 's' nog niet

    String s = Button1->Caption;

    //Hier bestaat 's'

} // 's' gaat 'out of scope'
```

Elke variabele naam moet uniek zijn in zijn eigen scope. Onderstaande code is dus niet mogelijk:

```
{ //Een scope, bijvoorbeeld een OnClick Event

    String s = Button1->Caption;
    String s = Button1->Caption; //ERROR: s bestaat al

}
```



Een variabele kan ook globaal zijn: dan kan deze overal vanuit het programma aangeroepen worden. Dat is heel gemakkelijk.



Minimaliseer het gebruik van globale variabelen. Het maakt je programma onoverzichtelijker en moeilijker te debuggen.

Onderstaand voorbeeld mag wel:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    String s = Button1->Caption;

} //Bovenstaande 's' gaat hier out of scope

void __fastcall TForm1::Button2Click(TObject *Sender)
{
    //Nog geen 's' bekend

    String s = Button2->Caption;

    //Nu is de 's' van deze scope bekend

} //Deze tweede 's' gaat hier out of scope
```

Een scope kun je maken met twee accoladen. En een **if** statement heeft twee accolades. Daarom is onderstaand voorbeeld legaal:

```
if (Button1->Tag == 1)
{
    String s = Button1->Caption;

    Form1->Caption = s;
}
else
{
    String s = Button2->Caption;

    Form1->Caption = s;
}

//Form1->Caption = s; //ERROR: 's' bestaat hier niet meer!
```



*Declareer je variabelen zo lokaal mogelijk,
oftewel: zorg dat je variabelen zo kort mogelijke
scope hebben.*

2.2. Conversie van variabelen

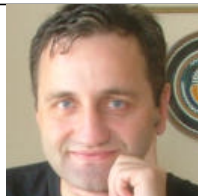
De conversies die we tot nu toe nodig hebben zijn die tussen **int**, **double** en **String**. Hieronder een overzicht.

Van	Naar	Voorbeeld
double	int	int i = static_cast<int> (d);
double	String	String s = FloatToStr(d);
int	double	double d = static_cast<double> (i);
int	String	String s = IntToStr(i);
String	double	double d = s.ToDouble();
String	int	int i = s.ToInt();

De logica lijkt ver te zoeken. De enige 'pure C++' conversies zijn de twee **static_cast**'s tussen **int** en **double**. Dit wordt een 'expliciete cast' genoemd: het is duidelijk te zien dat er een type wordt geconverteerd.



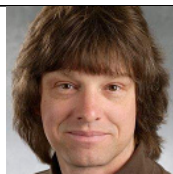
*De syntax van **static_cast** heb ik expres lelijk gemaakt: een cast duidelijk zichtbaar zijn en het gebruik van casts moet ontmoedigd worden.*



*De syntax van **static_cast** maakt de syntax van een cast uniform: wat je ook naar wat cast, de schrijfwijze is hetzelfde. Dit heeft veel voordelen bij generiek programmeren (oftwel met templates).*

Als we een getal naar **String** willen converteren, dan moeten we de Borland-specifieke functies **IntToStr** en **FloatToStr** gebruiken. De functie '**DoubleToStr**' bestaat niet.

Een **String** naar een getal krijgen, dat gaat 'via een punt'. De **String** is zo geprogrammeerd dat deze een eigen manier heeft om de conversie te doen. Later zullen we dit een 'methode' noemen.




Minimaliseer het gebruik van casts.

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    String s;
    s.
}

```



```

Function AnsiString & AnsiString::Insert (const AnsiString &,int)
//-- Function AnsiString & AnsiString::Delete (int,int)
Function AnsiString & AnsiString::SetLength (int)
Function int AnsiString::Pos (const AnsiString &) const
Function AnsiString AnsiString::LowerCase () const
Function AnsiString AnsiString::UpperCase () const
Function AnsiString AnsiString::Trim () const
Function AnsiString AnsiString::TrimLeft () const
Function AnsiString AnsiString::TrimRight () const
Function AnsiString AnsiString::SubString (int,int) const
Function int AnsiString::ToInt () const
Function int AnsiString::ToIntDef (int) const
Function double AnsiString::ToDouble () const
Function int AnsiString::WideCharBufSize () const
Function wchar_t * AnsiString::WideChar (wchar_t *,int) const
Function AnsiString::TStringMbcByteType AnsiString::ByteType (int) const
Function bool AnsiString::IsLeadByte (int) const
Function bool AnsiString::IsTrailByte (int) const
Function bool AnsiString::IsDelimiter (const AnsiString &,int) const
Function bool AnsiString::IsPathDelimiter (int) const
Function int AnsiString::LastDelimiter (const AnsiString &) const
Function int AnsiString::AnsiPos (const AnsiString &) const
Function char * AnsiString::AnsiLastChar () const

```

Als we de punt schrijven, verschijnt er een lijst met functies die een String kan. Het verschijnen van dit lijstje wordt Class Browsing genoemd. De functies ToInt en ToDouble staan er ook tussen.

2.3. Rekenen met int en double

Een **int** en een **double** rekenen anders:

```
int    a = 1    / 3    ; //a wordt 0
int    b = 1.0  / 3.0; //b wordt 0
double c = 1.0  / 3.0; //c wordt 0.333
double d = 1    / 3    ; //d wordt 0.0
```

Merk op dat een 1 de **integer** 1 is en 1.0 de **double** 1.0 is.

Als de integer 1 wordt gedeeld door de integer 3, past dit 0 keer. 1.0 gedeeld door 3.0 wordt de 0.333, maar dit getal wordt impliciet gecast tot de integer 0. Ook 0.999 converteerd naar de integer 0.

Wil je dat getallen als **double** delen, cast deze dan eerst:

```
double d
= static_cast<double>(1)
/ static_cast<double>(3); //d wordt 0.333
```



Je kunt ook een C-style cast doen:

```
double d = (double) (1) / (double) (3);
```



Gebruik geen C-style casts.



*Alleen als je de regels weet, weet je dat
double d = static_cast<double>(1) / 3;
Bij twijfel: cast altijd alles naar **double**.*

2.4. De for-loop

De for-loop maakt het mogelijk een stuk code te laten herhalen, zolang een bepaalde conditie waar is, met elke mogelijk 'stap'.

```
for (
    /* initialisatie */ ;
    /* conditie      */ ;
    /* stap          */
)
{
    //Doe iets
}
```

```
for (
    int i = 0; //Defineer de integer i op nul
    i != 10;   //Ga door zolang i ongelijk is aan tien
    i = i + 1  //Tel steeds 1 op bij i
)
{
    Memol->Lines->Add(i);
}
```

Merk de volgende dingen op:

- * Er kan in het initialisatie-gedeelte een variabele worden gedeclareerd. De scope van deze variabele is binnen de **for**-loop.
- * For-loops zijn meestal van het type **int**.
- * For-loops beginnen vaak bij 0 te tellen (hoewel dit bij bovenstaand voorbeeld zinloos zou zijn). Dit wordt later duidelijker waarom.
- * For-loops lopen vaak tot een waarde (in plaats van tot-en-met een waarde).
- * De drie delen van de for-loop zijn alle drie optioneel.

2.5. Verkorte rekenschrijfwijze

Volgende schrijfwijzen zijn equivalent:

```
//Tel twee op bij de integer i
i = i + 2; //Lees: de nieuwe waarde van i
           // is de oude waarde van i plus 2
i += 2;    //Lees: verhoog i met 2
```

```
//Tel een op bij de integer i
i = i + 1; //Lees: de nieuwe waarde van i
           // is de oude waarde van i plus 1
i += 1;    //Lees: verhoog i met 1
i++;       //Lees: verhoog i
++i;       //Lees: verhoog i
```

De '++i' notatie wordt veel gebruikt in **for**-loops. '--i' kan gebruikt worden om de integer i met een te verlagen.



*Prefereer de kortst mogelijke schrijfwijze.
Prefereer '++i' boven 'i++'.*



*Gebruik 'i++' alleen als je weet waarom deze
schrijfwijze nodig is. In praktijk is dit bijna
nooit.*

2.6. Meer over for-loops

Er zijn nog drie sleutelwoorden die nuttig kunnen zijn in een for-loop.

```
for ( /* iets */ ; /* iets */ ; /* iets */)
{
    if ( /* iets */ )
    {
        continue; //Ga nu al verder met de loop.
    }
    if ( /* iets */ )
    {
        break; //Ga nu uit de for-loop
    }
    if ( /* iets */ )
    {
        return; //Ga nu uit de Event/methode/functie
    }
}
```

2.7. Speciale for-loops

Een **for** loop kan ook oneindig lopen.

```
for ( ; ; ) //Een oneindige for-loop
{
    if ( /* iets */ ) //Ha, toch niet oneindig!
    {
        break;
    }
}
```

Als de **for**-loop altijd blijft doorgaan, hangt je programma.

2.8. Een truc

```
void __fastcall TForm1::ButtonStartClick(TObject *Sender)
{
    ButtonStart->Tag = 1;
    for (int i = 0 ; ButtonStart->Tag == 1; ++i)
    {
        ButtonStart->Caption = IntToStr(i);

        //Zorgt dat applicatie nog zijn dingen kan doen
        Application->ProcessMessages();
    }
}

void __fastcall TForm1::ButtonStopClick(TObject *Sender)
{
    ButtonStart->Tag = 0;
}
```

2.9. Het switch statement

Het **switch** statement is een soort **if**-statement. Het is geschikt om een hele rits **if**-statement op dezelfde waarde verkort te kunnen schrijven.

Onderstaande stukken code zijn equivalent:

```
int aantal = /* iets */ ;
if (aantal == 1)
{
    Button1->Caption = "Een";
}
else if (aantal == 2)
{
    Button1->Caption = "Twee";
}
else if (aantal == 3)
{
    Button1->Caption = "Drie";
}
else
{
    //Default waarde
    Button1->Caption = "Iets anders";
}
```

```
int aantal = /* iets */;
switch (aantal)
{
    case 1: Button1->Caption = "Een";           break;
    case 2: Button1->Caption = "Twee";          break;
    case 3: Button1->Caption = "Drie";           break;
    default: Button1->Caption = "Iets anders"; break;
}
```

Merk het volgende op:






- * Een **switch** statement is leesbaarder dan een hele rits **if**-statements.
- * Een **case** is altijd een enkel getal.
- * De **default** waarde is optioneel.
- * Elke **case** eindigd met een **break**. Door deze **break** wordt het **switch** statement beeindigd. Zonder deze **break** wordt de volgende **case** behandeld, dit wordt 'fall through' genoemd.
- * In een **switch** statement kunnen geen variabelen worden gedeclareerd.

Een **switch** statement is veel minder flexibel dan een **if** statement. Hierdoor is het gemakkelijker een **switch** te overzien.

2.10. Debuggen

Nu onze programma's complexer worden, is het nuttig en leerzaam om de debugger te leren gebruiken. Een fout wordt ook wel een bug genoemd. Debuggen is dus het verwijderen van fouten.

Voor het debuggen is het nuttig de volgende icoontjes in je menubalk te hebben. Dit kan door met de rechtermuisknop op de menubalk te klikken en dan 'Customize | Commands' en dan de icoontjes te slepen naar de gewenste plek op de menubalk.

Icoon	Naam	Sneltoets	Omschrijving
	Run	F9	Start het programma
	Reset	Ctrl-F2	Stop het programma
	Trace Into	F7	Ga met de debugger in een functie
	Step Over	F8	Ga met de debugger naar de volgende regel code
	View Breakpoints	Ctrl-Alt-B	Bekijk alle breakpoints

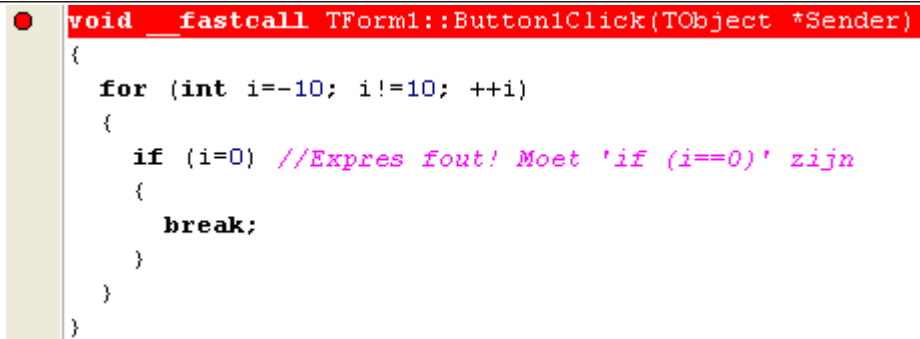
Alle programmeurs maken fouten. Onderstaande code zorgt dat het programma hangt, wegens het vergeten van een is-teken.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    for (int i=-10; i!=10; ++i)
    {
        if (i=0) //Expres fout! Moet 'if (i==0)' zijn
        {
            break;
        }
    }
}
```

De compiler denkt de fout al te zien en geeft dit aan met de volgende melding:

[C++ Warning] Unit1.cpp(21): W8060 Possibly incorrect assignment

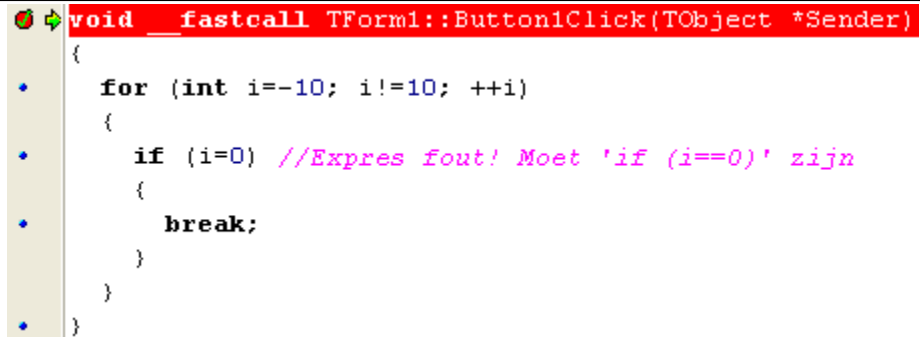
Met de debugger is deze fout gemakkelijk te ontdekken. Dit doen we met een breakpoint. Een breakpoint is een punt in het programma, waarop het programma pauzeert. Een breakpoint kan geplaatst worden door links in de 'gutter' te klikken.



```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    for (int i=-10; i!=10; ++i)
    {
        if (i=0) //Expres fout! Moet 'if (i==0)' zijn
        {
            break;
        }
    }
}
```

Een breakpoint

Als we het programma starten en op Button1 klikken, dan stopt het programma op deze breakpoint.



```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    for (int i=-10; i!=10; ++i)
    {
        if (i=0) //Expres fout! Moet 'if (i==0)' zijn
        {
            break;
        }
    }
}
```

Kylix wacht

Met 'Step Over' (F8) kunnen we naar de volgende regel code gaan. Door met de muiscursor over een variabele te zweven, krijgen we de waarde van deze variabele te zien.



Omdat *i* nog geen waarde heeft gekregen, maar wel een plekje in het computergeheugen heeft, laat de debugger zien wat er op dit moment op het plekje van *i* staat. Dit kan elke willekeurige waarde zijn. Een 82 in dit geval. Na nog een 'Step Over' wordt *i* pas op de waarde -10 gezet.

Na een paar keer 'Step Over' wordt duidelijk dat de waarde van *i* steeds op nul wordt gezet, inplaats van dat er getest wordt of *i* gelijk is aan nul. De fout is gevonden.

2.11. Volgende week

Volgende week leren we hoe we functies kunnen maken. Ook leren we hoe we de voorgeprogrammeerde functie `assert` aan kunnen roepen. `assert` is een belangrijke debug functie. De `TStringGrid` wordt behandeld: een Component voor een twee dimensionale tabel.

3. Dag 3



Een mogelijk eindprogramma van vandaag.

Onze code gaat er steeds groter en onoverzichtelijker uitzien. Ook zal er regelmatig een stuk code gekopieerd worden. Om dit te voorkomen leren we vandaag onze eigen functies te maken. We leren ook de standaard functie `assert` gebruiken, die helpt met het debuggen van onze grote programma's. De `TStringGrid` is het Component van vandaag, een 2-dimensionale tabel van Strings.

3.0. TStringGrid

Het StringGrid kan gebruikt worden als een 1- of 2-dimensionale tabel van Strings.

Onderstaande code vult alle vakjes van een StringGrid met hun x en y coördinaat.

```
int maxy = StringGrid1->RowCount;
int maxx = StringGrid1->ColCount;

for (int y=0; y!=maxy; ++y)
{
    for (int x=0; x!=maxx; ++x)
    {
        String s = "(" + IntToStr(x) + "," + IntToStr(y) + ")";
        StringGrid1->Cells[x][y] = s;
    }
}
```

Merk het volgende op:

- * Inplaats van het aantal rijen en kolommen in de **for**-loop te zetten, is gekozen deze eerst maxx en maxy te noemen. Dit hoeft niet.
- * Een bepaald vakje wordt met de blokhaken geselecteerd. De blokhaken worden dan ook wel de index-operator genoemd. Omdat een StringGrid 2-dimensionaal is, moet er met twee paar blokhaken worden gewerkt.
- * Het eerste vakje heeft index nul. C en C++ beginnen te tellen bij nul.
- * De for-loop loopt van nul tot het aantal kolommen/rijen. Als het aantal kolommen/rijen gelijk is aan 10, loopt de for-loop dus van 0 tot en met 9. Dit zijn dus wel het gewenst aantal van 10 iteraties.
- * Een String kan bestaan uit meerdere opgetelde Strings.

Een StringGrid kan gebruikt worden om veel Strings op te slaan en deze op index aan te spreken. Omdat we al Strings kunnen converteren naar **int** en **double**, kunnen we ook getallen (als Strings) erin opslaan.

3.1. Functies

Een functie is een stukje code dat iets doet en vanuit het hele programma aangeroepen kan worden. Functies maken code

leesbaarder (door een duidelijke naam te kiezen) en maken 'copy-paste code' onnodig.

```
/* Return type */ /* Functienaam */ ( /* Argumenten */ )  
{  
    /* Functie code */  
}
```

```
bool IsPaswoordJuist(String paswoord)  
{  
    bool isJuist = false;  
    if (paswoord == "iloverichel")  
    {  
        isJuist = true;  
    }  
    return isJuist;  
}
```



Geef een eenheid, zoals een functie, een duidelijke verantwoordelijkheid.

De functie 'IsPaswoordJuist' kan als volgt worden aangeroepen:

```
void __fastcall TForm1::Button1Click(TObject *Sender)  
{  
    String s = Edit1->Text;  
    bool gelukt = IsPaswoordJuist(s);  
    if (gelukt == true)  
    {  
        /* Iets */  
    }  
}
```

Merk het volgende op:

- * De functie 'IsPaswoordJuist' heeft een enkel argument van het type String en noemt deze String intern 'paswoord'. Als deze functie aangeroepen wordt, hoeft deze String helemaal niet 'paswoord' te heten. Button1Click noemt het paswoord 's', geeft deze String aan 'IsPaswoordJuist' en deze noemt deze String intern anders. Een functie heeft een eigen, beperkte scope.
- * De functie 'IsPaswoordJuist' geeft een **bool** terug noemt deze **bool** intern 'isJuist'. Als deze functie aangeroepen

wordt, hoeft de teruggegeven bool helemaal niet 'isJuist' te heten. Button1Click noemt deze bool 'gelukt'. Een functie heeft een eigen, beperkte scope.

Een functie is te vergelijken met een mannetje in een machine. Bovenstaande machine (functie) heet 'IsPaswoordJuist' en heeft een enkel in-laatje (argument) waar een String in moet en een enkel uit-laatje waar een bool in komt te zitten. Het mannetje in de machine noemt het in-laatje 'paswoord'. Degene die de machine gebruikt kan er elke String in stoppen die 'ie maar wil en de uitkomende bool noemen hoe 'ie wil.

Een functie kan ook geen return type hebben. Dan is het return type **void** (**void** betekend 'leeg'). Dit wordt een void-functie of procedure genoemd.

```
void LaatCoördinatenZien(TStringGrid * stringGrid)
{
    int maxy = stringGrid->RowCount;
    int maxx = stringGrid->ColCount;

    for (int y=0; y!=maxy; ++y)
    {
        for (int x=0; x!=maxx; ++x)
        {
            String s
                = "(" + IntToStr(x) + "," + IntToStr(y) + ")";
            stringGrid1->Cells[x][y] = s;
        }
    }
}
```

Merk het volgende op:

- * In het argument staat een sterretje. Dit is omdat een TStringGrid van het type 'TStringGrid *' is. Later zullen we dit type een 'TStringGrid pointer' gaan noemen.
- * Er wordt geen gebruik gemaakt van **return**. Bij de accolade-sluiten van deze void-functie gebeurt dit automatisch. **return** kan wel gebruikt worden. Dit is nuttig bij het voortijdig beëindigen van een void-functie.

Een functie kan ook geen argumenten hebben:

```
String GetPaswoord()
{
```

```

    return "iloverichel";
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    String paswoord = GetPaswoord();
    String ingevoerd = Edit1->Text;
    if (paswoord == ingevoerd)
    {
        /* Iets */
    }
}

```

Merk op dat bij het aanroepen van een functie zonder argumenten wel twee ronde haakjes gebruikt worden. Zouden deze haakjes weggelaten worden, dan denkt de compiler dat het geheugenadres van de functie wordt aangeroepen.



*In C heeft een functie zonder argumenten **void** tussen de ronde haken staan (bijvoorbeeld '**bool** DoeIets(**void**);'. Dit is walgelijk ('an abomination') in C++.*

3.2. De debugger en functies

Zet eens een breakpoint op een regel waarin een functie wordt aangeroepen, bijvoorbeeld onderstaande regel uit bovenstaand stuk code.

```
String paswoord = GetPaswoord();
```

Als het programma op dit punt stopt, kun je twee dingen doen:

- * Step Over (F8) om naar de volgende regel te gaan
- * Trace Into (F7) om in de functie te gaan

Met 'Trace Into' op een regel met een functie aanroep gaan we in een functie.

3.3. Waar zet ik mijn functies in Kylix? Unit1.h?

Een goede plek om functies neer te zetten is in 'Unit1.cpp' en 'Unit1.h'. Met CTRL-F6 kan gewisseld worden tussen deze twee.

Een Unit is gedefinieerd als een header file met een implementatie bestand met dezelfde naam. De header file heeft als extensie **.h** of **.hpp**, een implementatie bestand **.cpp**.

Een header file bevat in de regel definities en een implementatie bestand declaraties. Een definitie is 'wat er is', een declaratie is 'hoe dit werkt'.

Hieronder staat een voorbeeld van een 'Unit1.h' bestand:

```
//-----
#ifndef Unit1H
#define Unit1H
//-----
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
#include <QGrids.hpp>
//-----
class TForm1 : public TForm
{
    __published: // IDE-managed Components
        TStringGrid *StringGrid1;
        TButton *Button1;
        TButton *Button2;
        TEdit *Edit1;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
    private: // User declarations
    public: // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//-----
extern PACKAGE TForm1 *Form1;
//-----
bool IsPaswoordJuist(String paswoord);
void LaatCoordinatenZien(TStringGrid * stringGrid);
//-----
#endif
```

Merk het volgende op:

- * Er is een soort 'primitief if-statement' te zien met het #ifndef, #define en #endif trio. Dit wordt een #include guard genoemd.
- * Er worden veel dingen #included. Met #include kan een header file worden gelezen. Alle 5 #includes zijn automatisch hier geplaatst door Kylix en zorgen ervoor dat de compiler al onze Components kent. Het ligt voor de hand dat 'QForms.hpp' de informatie voor een Form bevat en 'QGrids.hpp' voor TStringGrid.
- * Op het Form staan een StringGrid, twee Buttons en een Edit Component gedeclareerd.
- * Deze Components hebben als identifiers StringGrid1, Button1, Button2 en Edit1.
- * StringGrid1 is van het data type 'TStringGrid *'. Ook de andere Components hebben een asterisk in hun data type declaratie. Een 'TStringGrid *' wordt als 'TStringGrid pointer' uitgesproken.
- * De Event 'Button1Click' en 'Button2Click' zijn gedeclareerd.
- * De functies 'IsPaswoordJuist' en 'LaatCoordinatenZien' zijn gedeclareerd. Een declaratie van een functie eindigt met een puntkomma.
- * Alle declaraties staan tussen de #ifndef en #endif.
- * Er is ook een '**extern** PACKAGE' TForm1 pointer genaamd Form1. Dit is de declaratie van de pointer waar we altijd al mee hebben gewerkt.

Hieronder staat de implementatie van Unit1.cpp:

```
//-----
#include <clx.h>
#pragma hdrstop

#include "Unit1.h"
//-----
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    if (IsPaswoordJuist(Edit1->Text) == true)
```



```

    Button1->Caption = "Gevonden!";
}
//-----
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    LaatCoordinatenZien(StringGrid1);
}
//-----
bool IsPaswoordJuist(String paswoord)
{
    if (paswoord == "iloverichel")
        return true;
    else
        return false;
}
//-----
void LaatCoordinatenZien(TStringGrid * stringGrid)
{
    int maxy = stringGrid->RowCount;
    int maxx = stringGrid->ColCount;

    for (int y=0; y!=maxy; ++y)
    {
        for (int x=0; x!=maxx; ++x)
        {
            stringGrid->Cells[x][y]
            = "(" + IntToStr(x) + "," + IntToStr(y) + ")";
        }
    }
}
//-----

```

Merk het volgende op:

- * In de eerste regel wordt 'clx.h' ge-#include. Dit is de belangrijkste header file van de CLX, de grafische bibliotheek waarmee we werken.
- * Unit1.h wordt ook ge-#include. Dit is de header file van ons Form.
- * De volgorde van Button1Click, Button2Click, IsPaswoordJuist en LaatCoordinatenZien is niet belangrijk.
- * Er worden verkorte schrijfwijzen gebruikt:
 - * Omdat 'Edit1->Text' een String is, mag deze ook direct in de functie IsPaswoordJuist worden gestopt.
 - * Als een if-statement maar een statement hoeft uit te voeren, mogen de accolades weggelaten worden.
- * De TForm1 pointer Form1 staat hier ook weer, zonder de

woorden **'extern PACKAGE'** ervoor. Dit is de definitie van onze veelgebruikte pointer.

Het is mogelijk een functie definitie in Unit1.h te zetten. Dan hoeft deze functie niet meer gedefinieerd te worden, maar dit mag wel. De volgorde maakt nu wel uit: alleen functies die 'boven' een functie staan kunnen worden aangeroepen.



Op elke plaats in C++ code kunnen alleen reeds gedeclareerde functies worden aangeroepen.

```
#ifndef Unit1H
#define Unit1H

//...

bool FunctieA(String a)
{
    /* iets */
}

bool FunctieB(String b)
{
    if ( FunctieA(b) == true) /* iets */
        /* iets */
}

#endif //Functie declaraties moeten voor de #endif
```

3.4. assert

C++ kent 209 standaard functies (standaard van 2003), waaronder assert. assert is de eerste functie die we leren gebruiken en aanroepen.

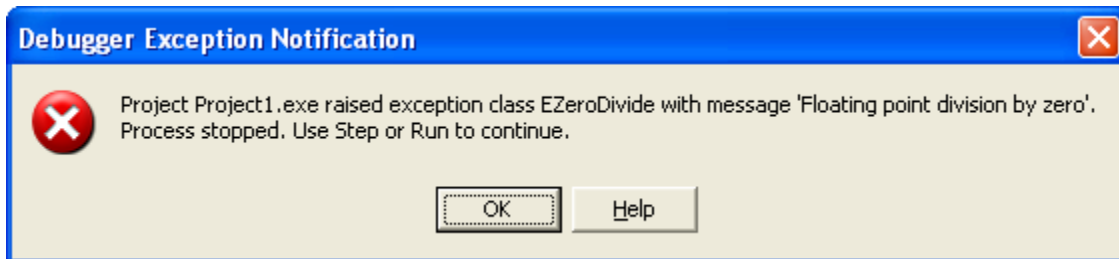
assert wordt gebruikt voor het debuggen en het documenteren van code.

Bekijk wat er mis kan gaan in onderstaande functie:

```
double DeelDoor(double teller, double noemer)
```

```
{  
    return teller / noemer;  
}
```

Wat er is mis kan gaan, is dat een deling door 0.0 onmogelijk is ('delen door nul is flauwekul').



Een niet-oplossing is commentaar:

```
double DeelDoor(double teller, double noemer)  
{  
    //noemer moet nooit 0.0 zijn!!!  
    return teller / noemer;  
}
```

Een **if**-statement kan ook een niet-oplossing bieden:

```
double DeelDoor(double teller, double noemer)  
{  
    //noemer moet nooit 0.0 zijn!!!  
    //is deze dat wel, geef dan een groot getal terug  
    if (noemer == 0.0) return 100000.0;  
    return teller / noemer;  
}
```

De functie is nu minder schoon: een deling door nul zou nooit voor moeten komen. Nu bepaald de functie wat er gebeurt bij een deling bij nul. Maar de aanroeper van de functie moet dit bepalen. Maar de functie moet aan kunnen geven wat 'ie wel en niet aankan.

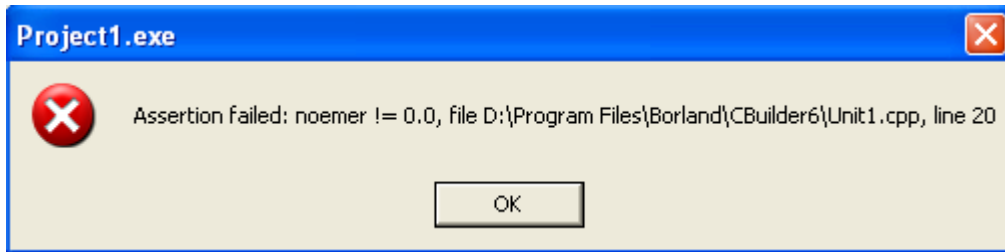


Geef een eenheid, zoals een functie, een duidelijke verantwoordelijkheid.

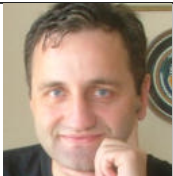
assert kan aangeven wat een functie wel of niet kan:

```
double DeelDoor(double teller, double noemer)
{
    assert(noemer != 0.0);
    return teller / noemer;
}
```

Als de functie DeelDoor nu wordt aangeroepen met een noemer van 0.0, dan komt er een foutmelding op het scherm, die zegt wat er mis gaat en in welke regel in welk bestand.



Merk op dat als deze melding verschijnt er niet op OK moet worden geklikt, want soms laat dit Kylix hangen. Een 'Program Reset' (CTRL-F2) voorkomt dit probleem.



Maak vrijelijk gebruik van assert om aannames te documenteren.

assert is niet meteen beschikbaar. Er moet een header file voor worden ge-#include:

```
#include <cassert>
```

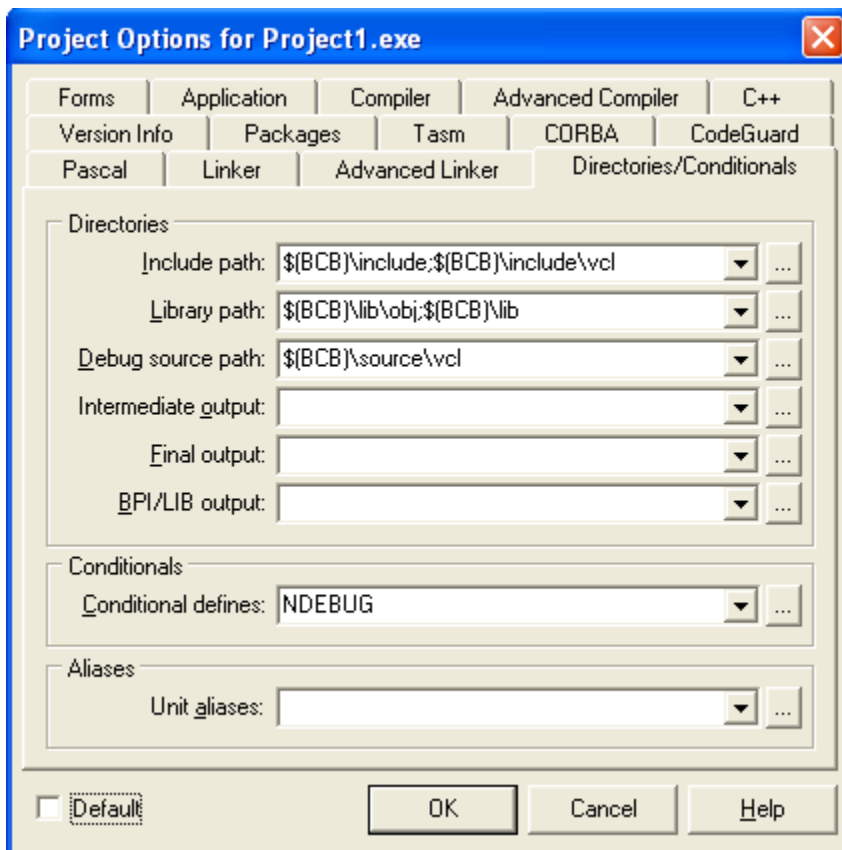
Deze #include kan op meerdere plaatsen staan, maar moet zowieso 'boven' een assert aanroep staan. Een mooie plek is tussen de standaard #includes in Unit1.h en Unit1.cpp.

assert heeft nog een belangrijk voordeel. Immers, we hebben nog niet gezien wat assert nuttiger maakt dan een **if** statement. Onderstaande code zou ook kunnen werken, dat gebruik maakt van ShowMessage, een Kylix-specifieke functie.

```
double DeelDoor(double teller, double noemer)
{
    if (noemer == 0.0) ShowMessage("Error!");
    return teller / noemer;
}
```

Nu, laten we aannemen dat het programma af is: alle checks zijn met **if** en ShowMessage gedaan, en er komen nooit foutmeldingen. Nu het programma geen fouten heeft, dan kunnen alle vertragende **if**-statements eruit. Vaak wordt deze **if** dan commentaar gemaakt. Het probleem is nu, bij grote programma's heb je honderden **if**-statements. Dit is veel werk!

Om alle asserts 'weg te krijgen', ga naar 'Project | Options | Conditionals' en voeg bij 'Conditional Defines' het woord 'NDEBUG' toe. Alle asserts blijven staan in de code, maar worden 'inactief' (specifieker: de pre-compiler verandert de asserts naar lege regels). Om de asserts weer te activeren, haal dan NDEBUG weg uit de 'Conditional Defines'.



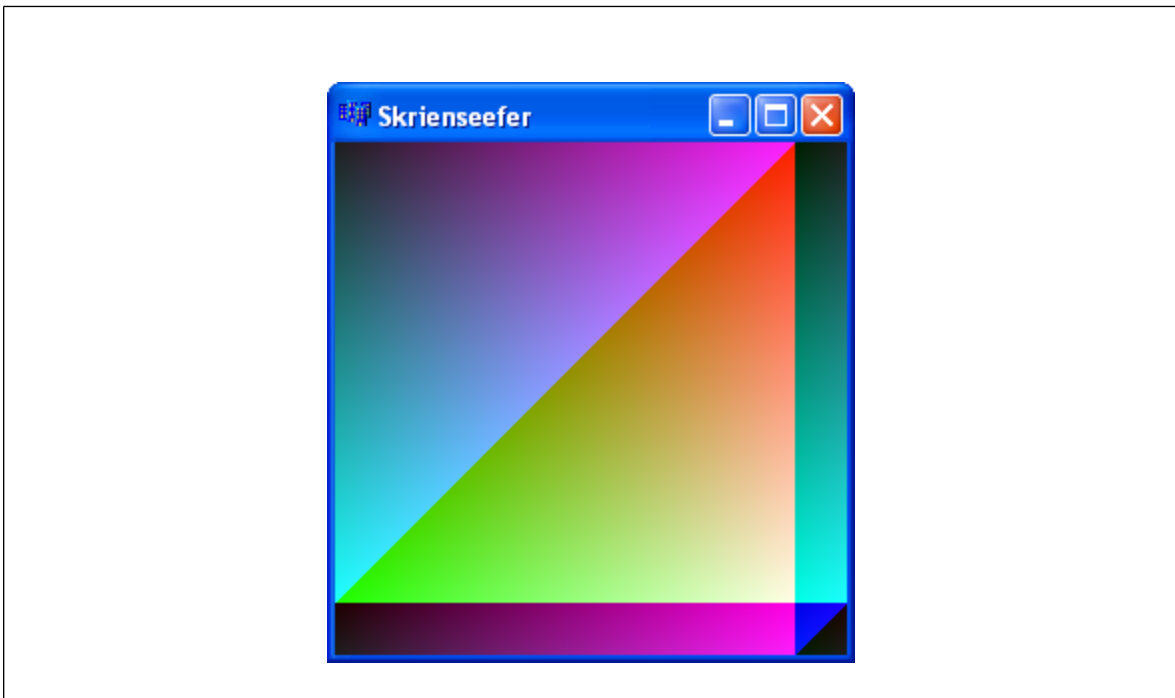
Merk op dat dit ook met code kan, door de volgende regel code toe te voegen voor de #include van de header file cassert:

```
#define NDEBUG
```

3.5. Volgende week

Onze functies kunnen nu nog maar een enkele waarde teruggeven. Volgende week leren we hoe we een functie op een andere manier meerdere waarden kunnen teruggeven. Ook leren we onze variabelen alleen-lezen te maken, wat een boel debuggen voorkomt. Ook leren we pixels te tekenen en hierdoor onze eerste screensaver te kunnen maken.

4. Dag 4



Een mogelijk eindprogramma van vandaag.

Vandaag leren we tekenen, of precieser: pixel manipulatie. Met referencing wordt het mogelijk dat een functie meerdere 'return waardes' heeft. En met het sleutelwoord **const** kan ervoor gezorgd worden dat een waarde niet meer kan veranderen. En het zeer belangrijke reference-by-const idioom wordt geleerd, waarbij en-passent wordt gespiekt naar wat er achter de schermen gebeurt bij een functie.

4.0. **const**

Bij het programmeren is de compiler een goede vriend. Hoe meer je aan de compiler verteld, hoe beter deze je kan helpen.

const is een sleutelwoord waarmee je zegt: deze variabele definieer ik met een waarde en deze kan niet meer veranderen. **const** wordt ook een modifier genoemd: het is geen data type op zich, maar een modificatie op een data type.

```
const int dozijn = 12;
```

Bovenstaande regel leest als: 'ik definieer dat een dozijn de integer 12 is, en dat dit onveranderlijk is'.



*Toen ik het concept van **const** bedacht was het eerst mijn idee dit **readonly** te noemen.*

Onderstaande code bevat een typefout:

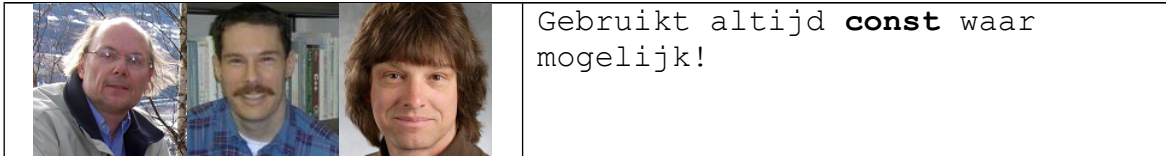
```
const int x = MagischeFunctie();  
if (x = 69) //FOUT: '=' moet '==' zijn  
{  
    /* iets */  
}
```

Omdat x een **const int** is, geeft de compiler de volgende melding:

```
[C++ Error] Unit1.cpp(21): E2024 Cannot modify a const  
object
```

(Merk op dat als x een **int** was, dat dan de compiler slechts een waarschuwing kan geven)

Door variabelen die **const** zouden moeten zijn **const** te maken, kan de compiler beter helpen. Maar het is ook duidelijker voor jezelf: je kunt aan het data type zien wat er nooit zal veranderen, waardoor je aannames tijdens het debuggen kunt maken.



Ook functieargumenten kunnen **const** zijn:

```
double DeelDoor(const double teller, const double noemer)
{
    assert(noemer != 0.0);
    return teller / noemer;
}
```

Hiermee zeg je dat de argumenten in de scope van de functie onveranderlijk zijn.

Pointers, naar bijvoorbeeld Button1, kunnen op twee manieren **const** zijn:

- * Datgeen waar de pointer naar wijst blijft **const**
- * De pointer blijft **const**, ofwel de pointer blijft naar hetzelfde object verwijzen

```
//Wijzigt de Caption van de Button
void SetCaption(
    TButton * const button,
    const String caption)
{
    button->Caption = caption;
}
```

```
//Leest alleen de Caption van de Button
//Deze functie zou moeten werken, maar is dit helaas niet
String GetCaption(const TButton * const button)
{
    return button->Caption;
}
```

Het lezen van een Caption wordt tegengehouden als we TButton **const** maken.

Er zijn twee mogelijkheden:

- * Het lezen van een Caption wijzigt de Button
- * De CLX is niet **const**-correct, oftewel maakt niet of te weinig gebruik van **const**.

Het waarschijnlijkst is de tweede optie: de CLX is niet perfect.



*Het werken met **const** is zo'n gedoe. Het invoeren van **const** in je code geeft een sneeuwbal effect op het moment dat je **const**-correcte code niet-**const**-correcte code aan gaat roepen.*



Gebruikt daarom meteen altijd **const** waar mogelijk!

Er zijn twee oplossingen:

- * TButton niet-**const** maken
- * Het gebruik van **const_cast**

Het niet-**const** maken van TButton is eigenlijk geen optie. De functie GetCaption moet de (terechte) indruk maken alleen de TButton te lezen.

Met **const_cast**, een broertje van **static_cast**, kan de constheid van een variabele worden veranderd. De functie GetCaption wordt dan als volgt:

```
String GetCaption(const TButton * const button)
{
    //const_cast nodig omdat de CLX (nog) niet const-correct
    //is... Niet zo netjes van Borland!
    TButton * const buttonCopy
        = const_cast<TButton *>(button);
    return buttonCopy->Caption;
}
```

Merk het volgende op:

- * De pointer buttonCopy is const.
- * De TButton waar buttonCopy naar wijst niet.
- * De **const_cast** werkt net als een **static_cast**.



*De syntax van **const_cast** heb ik expres lelijk gemaakt: een cast duidelijk zichtbaar zijn en het gebruik van casts moet ontmoedigd worden.*



Vermijd het gebruik van **const_cast**, behalve bij het gebruik maken van **const**-incorrecte bibliotheken. Laat dan wel een sjacherijnige opmerking achter in commentaar!



De syntax van **const_cast** maakt de syntax van een cast uniform: wat je ook naar wat cast, de schrijfwijze is hetzelfde. Dit heeft veel voordelen bij generiek programmeren.

4.1. Referencing

Onderstaande functie zal niet werken:

```
#include <cassert>

void Verwissel(String a, String b)
{
    const String temp = a;
    a = b;
    b = temp;
}

void MijnFunctie()
{
    String s1 = "s1";
    String s2 = "s2";
    Verwissel(s1,s2);
    assert(s1 == "s1"); //Hee, niet veranderd!
    assert(s2 == "s2"); //Hee, niet veranderd!
}
```

Wat ook niet de bedoeling is: als we s1 en s2 **const** maken, komt er geen waarschuwing!

De reden hiervoor is, dat als we iets doorgeven aan een functie, dat dit in eerste instantie een kopie is: achter de schermen om wordt het origineel gekopieerd en aan de functie gegeven. Wat de functie er vervolgens mee doet, dat maakt voor het origineel niet uit. Daarom kan in bovenstaand voorbeeld s1 en s2 **const** zijn.

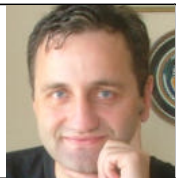
Referencing zorgt ervoor dat er met de originelen gewerkt kan worden:

```
#include <cassert>

void Verwissel(String& a, String& b)
{
    const String temp = a;
    a = b;
    b = temp;
}

void MijnFunctie()
{
    String s1 = "s1";
    String s2 = "s2";
    Verwissel(s1,s2);
    assert(s1 == "s2");
    assert(s2 == "s1");
}
```

Het &-teken, ook wel ampersand genoemd, zorgt ervoor dat er niet met een kopie, maar met het origineel gewerkt wordt.



*De functie Verwissel (in het Engels 'Swap') is vaak de functie die als eerste voorbeeld in generiek (met **template**) programmeren wordt laten zien.*

Maken we de String s1 **const**, dan komt de volgende waarschuwig:

```
[C++ Warning] Unit1.cpp(23): W8030 Temporary used for
parameter 'a' in call to 'Verwissel(AnsiString &,AnsiString
&)'
```

Dit betekent dat de compiler een tijdelijke kopie van s1 heeft gemaakt en deze kopie met de originele s2 heeft omgewisseld. s1 blijft dus ongewijzigd.

4.2. Referencing voor meerdere return-waarden

Ook kan referencing worden gebruikt om meerdere **return**-waardes te hebben:

```
#include <cassert>
```

```

void LowerUpper(
    const String s,
    String& lower,
    String& upper)
{
    lower = s.LowerCase();
    upper = s.UpperCase();
}

void MijnFunctie()
{
    const String paswoord = "SesamOpenU";
    String laag = "";
    String hoog = "";
    LowerUpper(paswoord, laag, hoog);
    assert(laag == "sesamopenu");
    assert(hoog == "SESAMOPENU");
}

```

4.3. Reference-to-const

Met referencing kan er met het origineel gewerkt worden. Dit scheelt dat achter de schermen er een kopie van de variabele wordt gemaakt.

Een String kan uit miljoenen tekens bestaan. Onderstaande functie declaratie zal een kopie maken van een String, deze alleen lezen, en vervolgens de kopie weggooien:

```
int TelAantalSpaties(const String s);
```

Onderstaande functie declaratie gebruikt een reference, waarder er geen kopie van de String wordt gemaakt. Deze is sneller, maar garandeerd niet dat s ongewijzigd blijft:

```
int TelAantalSpaties(String& s);
```

Het reference-to-const idioom garandeerd dat er met het origineel wordt gewerkt, zonder dat deze wordt gewijzigd:

```
int TelAantalSpaties(const String& s);
```

Een String is typisch een data type dat met reference-to-const wordt doorgegeven aan een functie.

Bij een **int** is dit een ander verhaal: het kopiëren van een **int** gaat net zo snel als het doorgeven van een referentie. Daarom worden simpele datatypen 'gewoon' (dus met kopie) doorgegeven aan functies:

```
int Kwadraat(const int x);
```



*Geef de voorkeur een reference-to-const aan een functie te geven, behalve bij ingebouwde types, zoals **int**.*

4.4. References en pointers

References en pointers lijken veel op elkaar. Onderstaande functie maakt geen kopie van een TButton:

```
String GetCaption(const TButton * const button)
{
    //const_cast nodig omdat de CLX (nog) niet const-correct
    //is... Niet zo netjes van Borland!
    TButton * const buttonCopy
        = const_cast<TButton *>(button);
    return buttonCopy->Caption;
}
```

Een pointer bevat een geheugenadres, in dit geval het geheugenadres waar de TButton zich bevindt. Dit is precies hetzelfde als een reference (voor ons huidige begrip)!












*Gebruik functieparameters op de juiste manier:
* input (oftewel 'alleen lezen') is altijd **const**.
* output is een **return** waarde of reference (of pointer).*

4.5. Pixel manipulatie

Een pixel is een puntje op het beeldscherm van een bepaalde kleur. De kleur van een pixel bestaat uit drie kleine lampjes met de kleuren rood, groen en blauw. Een lampje heeft een waarde van 0 tot 256, waarbij 0 betekent dat 'ie uit staat en 255 dat 'ie vol aan staat. Door combinaties met deze drie lampjes te maken kun je alle kleuren maken. Elke

kleur heeft een rood, groen, blauw waarde, of korter: een RGB waarde.

Kleur		Rood	Groen	Blauw
Zwart		0	0	0
Rood		255	0	0
Groen		0	255	0
Blauw		0	0	255
Geel		255	255	0
Paars		255	0	255
Aqua		0	255	255
Grijs		127	127	127
Wit		255	255	255

De kleur van een pixel lezen of te wijzigen is in de CLX relatief moeilijk.

Daarom geef ik hier twee functies die gemakkelijk te gebruiken zijn, maar waarvan de preciese werking nog even onduidelijk zal blijven.

De functie GetPixel haalt door middel van referencing los de RGB waarden op:

```
void GetPixel(
    const TImage * const image,
    const int x,
    const int y,
    unsigned char& red,
    unsigned char& green,
    unsigned char& blue)
{
    assert(image!=0 && "Image is NULL");
    assert(image->Picture->Bitmap!=0 && "Bitmap is NULL");
    assert(image->Picture->Bitmap->PixelFormat == pf32bit
        && "Bitmap must be 32 bit");
    assert( x >= 0 && "x coordinat is below zero");
    assert( y >= 0 && "y coordinat is below zero");
    assert( x < image->Picture->Bitmap->Width
        && "x coordinat is beyond image width");
    assert( y < image->Picture->Bitmap->Height
        && "y coordinat is beyond image height");
    red    = (static_cast<unsigned char*>(
        image->Picture->Bitmap->ScanLine[y]))[x*4+2]);
    green  = (static_cast<unsigned char*>(
        image->Picture->Bitmap->ScanLine[y]))[x*4+1]);
    blue   = (static_cast<unsigned char*>(
```

```

        image->Picture->Bitmap->ScanLine[y])[x*4+0]);
    }

```

```

void SetPixel(
    TImage * const image,
    const int x,
    const int y,
    const char red,
    const char green,
    const char blue)
{
    assert(image!=0 && "Image is NULL");
    assert(image->Picture->Bitmap!=0 && "Bitmap is NULL");
    assert(image->Picture->Bitmap->PixelFormat == pf32bit
        && "Bitmap must be 32 bit");
    assert( x >= 0 && "x coordinat is below zero");
    assert( y >= 0 && "y coordinat is below zero");
    assert( x < image->Picture->Bitmap->Width
        && "x coordinat is beyond image width");
    assert( y < image->Picture->Bitmap->Height
        && "y coordinat is beyond image height");

    static_cast<unsigned char*>(
        image->Picture->Bitmap->ScanLine[y])[x*4+2] = red;
    static_cast<unsigned char*>(
        image->Picture->Bitmap->ScanLine[y])[x*4+1] = green;
    static_cast<unsigned char*>(
        image->Picture->Bitmap->ScanLine[y])[x*4+0] = blue;
}

```

Merk het volgende op:

- * De functie leest/schrijft van/naar een TImage Component.
- * De meeste regels code zijn assert statements. Deze functie geeft dus precies aan wat 'ie aankan, namelijk 32-bit bitmaps. Als een assert statement mislukt, komt er een duidelijke tekst op het scherm. Tekst wordt blijkbaar gezien als **true**.
- * Er wordt een nieuw data type **unsigned char** gebruikt. Dit is een subset van het data type **int**, die als mogelijke waarden 0 tot 256 heeft.
- * De TImage heeft de Property Picture, die zelf ook weer Properties heeft.
- * Er wordt gebruik gemaakt van **static_cast**. Heel lelijk, dit komt omdat ScanLine hetzelfde gedrag vertoond als de Windowsfunctie ScanLine. Windowsfuncties (of nauwkeuriger: Win32 API functies) zijn vaak lelijk en hebben vaak casts nodig.

* ScanLine maakt gebruik van de index operator (de blokhaken) om eerst bij de y-coördinaat te komen, dan nog eens om bij de juiste x-coördinaat en kleur te komen.



*De functie ScanLine geeft een **void** pointer terug. Dit is een pointer zonder data type. Als je code niet type-safe maakt, is dit veel flexibeler!*



*Prefereer nooit een **void** pointer te gebruiken.*

Om aan alle voorwaarden voor bovenstaande functie te voldoen, is het volgende nodig:

- * Plaats een TImage Component op het Form
- * Zet de Picture Property op 24 of 32 bit bitmap. Gebruik een simpel tekenprogramma om zo'n bitmap te maken.

Een pixel te tekenen is nu simpel:

```
SetPixel(Image1,0,0,0,127,0); //Groene pixel op (0,0)
Image1->Refresh();
```

Refresh is nodig om de nieuwe bewerking(en) daadwerkelijk te laten zien. Het hele plaatje kan eerst worden getekend en vervolgens pas op het scherm worden getekend. Dit scheelt tijd. Roep Refresh pas aan als dit nodig is.

4.6. Een plaatje vergroten

Een 'plaatje vergroten' kan twee dingen betekenen:

- * Het plaatje moet groter op het scherm zichtbaar worden
- * De bitmap moet meer pixels gaan bevatten

Beide dingen zijn mogelijk.

Om het plaatje groter op het scherm zichtbaar te maken, zet de Property Stretch op **true** en maak de Width en Height van TImage zo groot als gewenst. Om het plaatje maximaal groot af te beelden, zet Align op 'alClient'.

De bitmap meer pixels geven is mogelijk door een grotere bitmap in te laden. Maar dit soort bitmaps zouden je programma onnodig groot kunnen maken. Zelf gebruik ik altijd dezelfde 32 bij 32 pixels (witte) bitmap en verander deze van grootte onder run-time. Dit doe ik in het nog magische stukje code dat Kylix automatisch genereerd:

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    Image1->Picture->Bitmap->Width  = 256;
    Image1->Picture->Bitmap->Height = 256;
}
```

Het stuk code van Kylix wordt aangeroepen bij de aanmaak van ons Form. Later zullen we dit 'de constructor van TForm1' noemen.

4.7. Rekenen met unsigned char en int

unsigned char en **int** worden impliciet ('achter je rug om') geconverteerd naar elkaar.

Als een te hoge **int** wordt geconverteerd naar **unsigned char**, wordt 'doorgeteld' na 255:

```
const int i = 256;
const unsigned char c = i;
assert(c==0);
```

Beter is echter dit toch expliciet te laten zien:

```
const int i = 256;
const unsigned char c = static_cast<unsigned char>(i);
assert(c==0);
```



*Poeh, wat een boel typewerk, zo'n explicite cast.
Geef mij maar het eerste stuk code!*



Een cast expliciet maken geeft duidelijk aan dat je een cast doet. Omdat casts een bron van run-time fouten zijn, is het nuttig deze snel op het oog te kunnen vinden.

4.8. Volgende week

Pixel manipulatie is handig, tot er grotere dingen moeten worden getekend, bijvoorbeeld een poppetje. Volgende week leren we een bestaande bitmap meerdere malen op het scherm te tekenen. Om bijvoorbeeld coördinaten van meerdere objecten op te slaan, kunnen we een TStringGrid gebruiken, maar deze schiet op andere punten te kort. We leren de 'abstracte versie' van het TStringGrid, de `std::vector`. Ook leren we onze variabelen 'in' ons Form te declareren, inplaats van steeds in een Caption of Tag. We zullen dit 'lid variabelen' of 'member variables' noemen.

4.9. Code van Skrienseefer

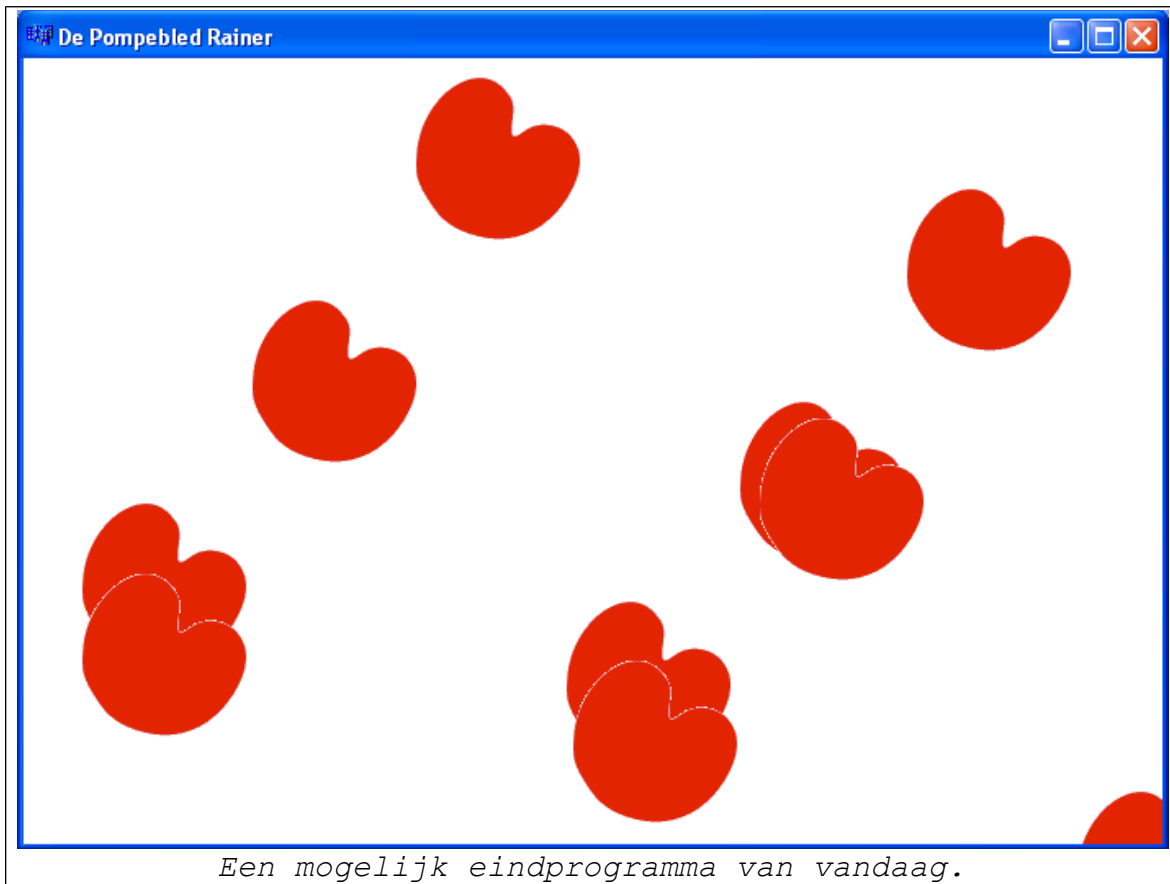
```
//-----  
#include <clx.h>  
#pragma hdrstop  
#include "Unit1.h"  
//-----  
#pragma package(smart_init)  
#pragma resource "*.xfm"  
TForm1 *Form1;  
//-----  
__fastcall TForm1::TForm1(TComponent* Owner)  
    : TForm(Owner)  
{  
    Image1->Picture->Bitmap->Width  = 256;  
    Image1->Picture->Bitmap->Height = 256;  
    Image1->Align = alClient;  
}  
//-----  
void __fastcall TForm1::Timer1Timer(TObject *Sender)  
{  
    ++Timer1->Tag;  
  
    const int z = Timer1->Tag;  
    const int maxx = Image1->Picture->Bitmap->Width;  
    const int maxy = Image1->Picture->Bitmap->Height;
```

```

for (int y=0; y!=maxy; ++y)
{
    for (int x=0; x!=maxx; ++x)
    {
        SetPixel(
            Image1,
            x,y,                                     //Coördinaat
            static_cast<unsigned char>(x+0+z)%256, //Rood
            static_cast<unsigned char>(0+y+z)%256, //Groen
            static_cast<unsigned char>(x+y+z)%256  //Blauw
        );
    }
}
Image1->Refresh();
}
//-----

```

5. Dag 5



Vandaag leren we onze eigen variabelen 'in' ons Form op te slaan, inplaats van steeds Tags en Captions te gebruiken. Ook leren we een niet-visueel TStringGrid kennen, die veel flexibeler en standaard C++ is, de `std::vector`. Dan hebben we alle kennis om in plaats van pixels bitmaps te tekenen.

5.0. Een member variable

Als we een **int** op willen slaan in ons Form, dan doen we dat nu via de Tag of Caption van een Component, bijvoorbeeld een TLabel. Dit is een visuele goed-werkende manier.

Maar om een **int** op te slaan, maken we een TLabel pointer aan. We maken veel meer aan dan we nodig hebben. Dit kan 'korter door de bocht'.

De data die we op willen slaan, kan gedeclareerd worden in Unit1.h zoals onderstaand voorbeeld:

```
class TForm1 : public TForm
{
__published:    // IDE-managed Components
    //Kylis dingen, afblijven dus!

private:    // User declarations
    int mX;

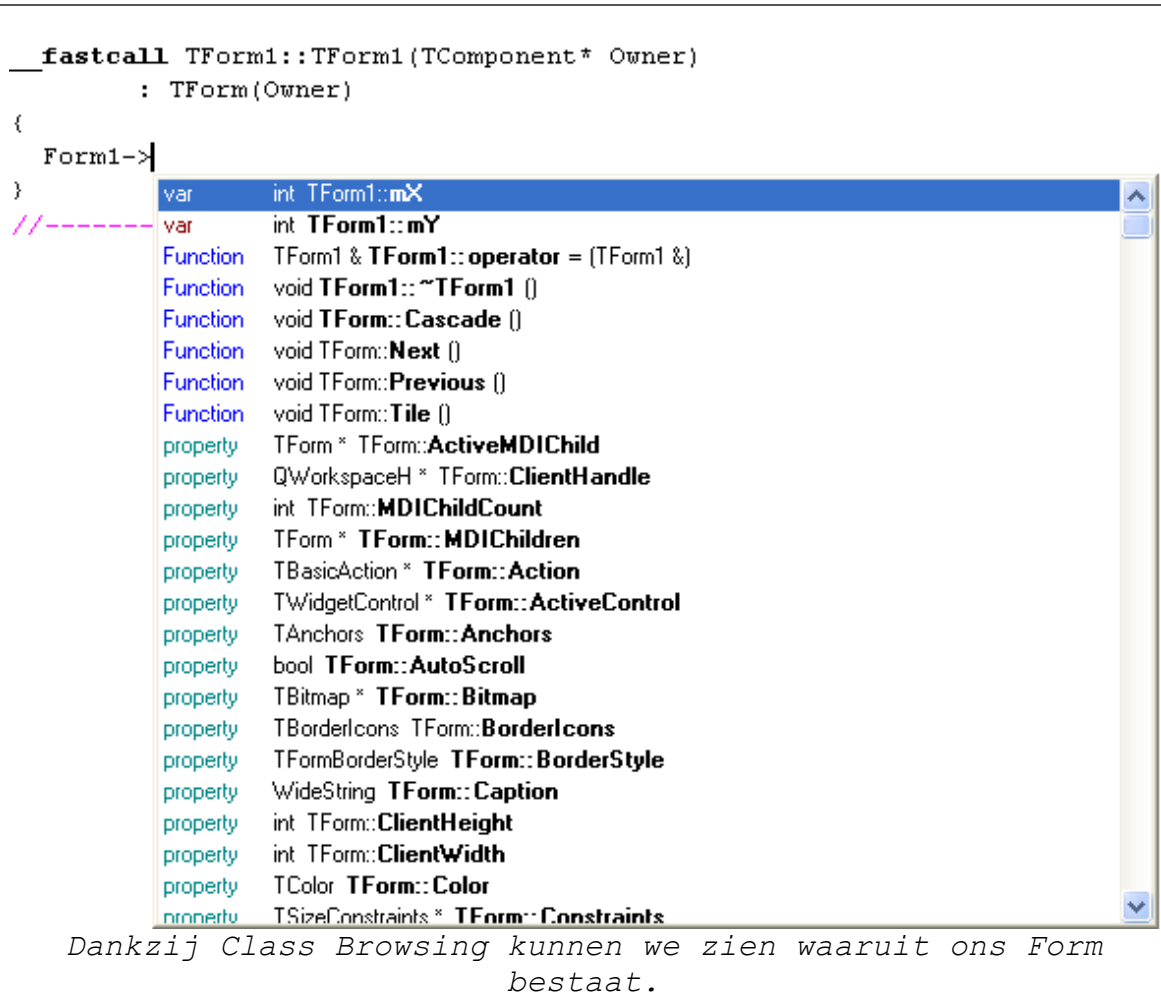
public:    // User declarations
    __fastcall TForm1(TComponent* Owner);
    int mY;
};
```

Merk het volgende op:

- * Het sleutelwoord **class** zou erop kunnen wijzen dat ons Form een klasse is.
- * Tussen de accolades zijn er drie gedeeltes: **__published**, **private** en **public**.
- * In commentaar wordt al aangegeven waar de gebruiker ('user') iets mag declareren
- * Declareer niets in het **__published** gedeelte.
- * Declareer je variabelen in het **private** of **public** gedeelte. Het verschil tussen **public** en **private** wordt pas duidelijk als we iets meer van klassen weten en met meerdere Forms gaan werken.
- * Ik begin deze variabele namen altijd met een 'm' van 'member', om aan te geven dat deze variabelen bij een Form horen. Anderen geven de voorkeur deze variabele namen te laten eindigen op een underscore.

Je kunt wisselen tussen Unit1.cpp en Unit1.h met de sneltoets CTRL-F6.

Hebben we deze variabelen correct gedefinieerd, dan kunnen we dat meteen zien, door in de voorgeprogrammeerde Event van Form1 het volgende te doen:



The screenshot shows a code editor with the following content:

```

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    Form1->
}

```

Below the code, a list of class members is displayed in a scrollable window:

- var int TForm1::mX
- var int TForm1::mY
- Function TForm1 & TForm1::operator = (TForm1 &)
- Function void TForm1::~~TForm1 ()
- Function void TForm::Cascade ()
- Function void TForm::Next ()
- Function void TForm::Previous ()
- Function void TForm::Tile ()
- property TForm * TForm::ActiveMDIChild
- property QWorkspaceH * TForm::ClientHandle
- property int TForm::MDIChildCount
- property TForm * TForm::MDIChildren
- property TBasicAction * TForm::Action
- property TWidgetControl * TForm::ActiveControl
- property TAnchors TForm::Anchors
- property bool TForm::AutoScroll
- property TBitmap * TForm::Bitmap
- property TBorderIcons TForm::BorderIcons
- property TFormBorderStyle TForm::BorderStyle
- property WideString TForm::Caption
- property int TForm::ClientHeight
- property int TForm::ClientWidth
- property TColor TForm::Color
- property TSizeConstraints * TForm::Constraints

Dankzij Class Browsing kunnen we zien waaruit ons Form bestaat.

Merk het volgende op:

- * Onze variabelen mX en mY staan bovenaan (ook al is mX **private** en mY **public**).
- * Het lijstje dat verschijnt laat in de meest linker kolom zien, of het een variabele, functie of Property betreft.
- * Het lijstje laat van elke variabele het data type en de naam zien.
- * Het lijstje laat van elke functie het return data type en de argumenten zien. In dit geval allemaal **void**. Er staat wel overal 'TForm::' of andere dingen voor. Deze aanduiding geeft aan waar de functies bij horen. Later leren we hoe dit precies werkt.
- * Het lijstje laat van alle Properties het data type en de naam zien. Er staat wel overal 'TForm::' of andere dingen

voor. Deze aanduiding geeft aan waar de Properties bij horen. Later leren we hoe dit precies werkt.

De voorgeprogrammeerde Event waarin we werken, heet de constructor van ons Form. Hierin kunnen we onze variabelen mooi initialiseren met de normale schrijfwijze:

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    Form1->mX = 10;
    Form1->mY = 10;
}
```

Dit kan zelfs nog korter:

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    mX = 10;
    mY = 10;
}
```

Normaliter zou de compiler een foutmelding geven ('Undefined symbol 'mX)'), maar omdat mX en mY onderdeel van ons Form kan de compiler deze nu wel vinden. In elke Event van TForm kun je bij de member variables van het TForm.

Merk ook op, dat met onderstaande syntax ook Class Browsing actief wordt:

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    this->mX = 10;
    this->mY = 10;
}
```

Later meer over **this**.

Het declareren van **const** member variables leren we later.

5.1. Een std::vector

De std::vector (spreek uit 'standaard vector' or 'vector') is ons eerste C++ data type waarmee we leren werken. De

`std::vector` is geen onderdeel van de CLX, maar van de STL, de 'Standard Template Library', de officiële C++ standaard bibliotheek. De STL is volledig platform-onafhankelijk. De `std::vector` wordt een 'container class' genoemd, omdat deze gemaakt is om variabelen te bevatten.



Je kunt ook gewoon een array gebruiken.



Gebruik `std::vector` in plaats van een array.

De verschillen tussen `std::vector` en `TStringGrid` zijn als volgt:

	<code>TStringGrid</code>	<code>std::vector</code>
Bibliotheek	CLX	STL
Dimensies	0,1 of 2	Elke
Data type	String	Elke
Visueel	Ja	Nee

Voor het gebruik van de `std::vector`, moet eerst de header file `vector.h` worden ge-`#included`:

```
#include <vector>
```

De syntax om een `std::vector` te declareren is als volgt:

```
std::vector< /* data type */ > /* identifier */;
```

Bijvoorbeeld:

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

Nu hebben we een 1-dimensionale `std::vector` gedeclareerd die **integers** op kan slaan.

Type het volgende om te zien wat een `std::vector` kan:

```

fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    std::vector<int> v;
    v.|
}
Function allocator<int> _STL::vector<int,_STL::allocator<int> >::get_allocator () const
// -- Function int * _STL::vector<int,_STL::allocator<int> >::begin ()
Function const int * _STL::vector<int,_STL::allocator<int> >::begin () const
Function int * _STL::vector<int,_STL::allocator<int> >::end ()
Function const int * _STL::vector<int,_STL::allocator<int> >::end () const
Function reverse_iterator<int * > _STL::vector<int,_STL::allocator<int> >::rbegin ()
Function reverse_iterator<const int * > _STL::vector<int,_STL::allocator<int> >::rbegin () const
Function reverse_iterator<int * > _STL::vector<int,_STL::allocator<int> >::rend ()
Function reverse_iterator<const int * > _STL::vector<int,_STL::allocator<int> >::rend () const
Function unsigned int _STL::vector<int,_STL::allocator<int> >::size () const
Function unsigned int _STL::vector<int,_STL::allocator<int> >::max_size () const
Function unsigned int _STL::vector<int,_STL::allocator<int> >::capacity () const
Function bool _STL::vector<int,_STL::allocator<int> >::empty () const
Function int & _STL::vector<int,_STL::allocator<int> >::operator [] (unsigned int)
Function const int & _STL::vector<int,_STL::allocator<int> >::operator [] (unsigned int) const
Function int & _STL::vector<int,_STL::allocator<int> >::front ()
Function const int & _STL::vector<int,_STL::allocator<int> >::front () const
Function int & _STL::vector<int,_STL::allocator<int> >::back ()
Function const int & _STL::vector<int,_STL::allocator<int> >::back () const
Function int & _STL::vector<int,_STL::allocator<int> >::at (unsigned int)
Function const int & _STL::vector<int,_STL::allocator<int> >::at (unsigned int) const
Function void _STL::vector<int,_STL::allocator<int> >::_M_initialize_aux<_Integer> [_Integer_]
Function void _STL::vector<int,_STL::allocator<int> >::_M_initialize_aux<_InputIterator> [_InputIterator_]
Function void _STL::vector<int,_STL::allocator<int> >::~~vector ()

```

Class Browsing laat ons de functies zien die een `std::vector<int>` kan.

Lees

*`_STL::vector<int, _STL::allocator<int> >`
gewoon als
`std::vector<int>`*

De belangrijkste functies zijn de volgende:

<code>[unsigned int]</code>	De index operator. Lees of schrijf op een index
<code>resize(unsigned int)</code>	Veranderd de grootte
<code>push_back(int)</code>	Prop er een getal extra bij voorbij de hoogste index
<code>pop_back()</code>	Gooi de hoogste index weg
<code>front()</code>	Lees de waarde op index 0
<code>back()</code>	Lees de waarde op de hoogste index
<code>size()</code>	Lees de grootte

Een voorbeeld met deze `std::vector` functies:

```
std::vector<String> v;  
assert(v.size() == 0);  
  
v.push_back("Nul");  
assert(v.size() == 1);  
assert(v.front() == "Nul");  
  
v.resize(2);  
v[1] = "Een";  
assert(v.size() == 2);  
assert(v.back() == "Een");  
assert(v[1] == "Een");  
  
v.push_back("Twee");  
assert(v.size() == 3);  
assert(v.back() == "Twee");  
  
v.pop_back();  
assert(v.size() == 2);  
assert(v.back() == "Een");
```



Geef de voorkeur aan `push_back` om je `std::vector` te vergroten.

We hebben als gezien dat we door middel van een punt te typen, we een `String` kunnen bekijken. Als we een `std::vector<String>` hebben, dan kunnen we met de index-operator bij een `String`. En deze `String` kunnen we dan bekijken met Class Browsing.

```

{
    std::vector<String> v;
    v.push_back("Nul");
    v[0].
}

```

Function AnsiString AnsiString::StringOfChar (char,int)

Function AnsiString AnsiString::LoadStr (int)

Function AnsiString AnsiString::LoadStr (void *,int)

Function AnsiString AnsiString::FmtLoadStr (int,const TVarRec *,int)

Function AnsiString & AnsiString::LoadStringA (void *,int)

Function AnsiString AnsiString::Format (const AnsiString &,const TVarRec *,int)

Function AnsiString & AnsiString::sprintf (const char *,...)

Function int AnsiString::printf (const char *,...)

Function int AnsiString::vprintf (const char *,void *)

Function AnsiString & AnsiString::cat_printf (const char *,...)

Function int AnsiString::cat_printf (const char *,...)

Function int AnsiString::cat_vprintf (const char *,void *)

Function AnsiString AnsiString::FormatFloat (const AnsiString &,const long double &)

Function AnsiString AnsiString::FloatToStrF (long double,AnsiString::TStringFloatFormat,int,int)

Function AnsiString AnsiString::IntToHex (int,int)

Function AnsiString AnsiString::CurrToStr (Currency)

Function AnsiString AnsiString::CurrToStrF (Currency,AnsiString::TStringFloatFormat,int)

Function void AnsiString::~AnsiString ()

Function AnsiString & AnsiString::operator = (const AnsiString &)

Function AnsiString & AnsiString::operator += (const AnsiString &)

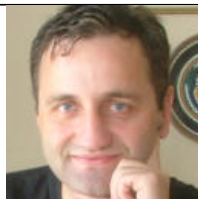
Function bool AnsiString::operator == (const AnsiString &) const

Function bool AnsiString::operator != (const AnsiString &) const

Function bool AnsiString::operator > (const AnsiString &) const

Function bool AnsiString::operator >= (const AnsiString &) const

Als we met de index operator een String te pakken hebben, kunnen we deze ook Class Browsen.



std::vector is een template class. Dit houdt in dat het data type dat de std::vector opslaat bekend is tijdens compilen. Hierdoor kan ook de Class Browser je helpen.



Er zijn nog meer container classes, die elk voor verschillende taken (snelheids)geoptimaliseerd is. Gebruik std::vector tenzij je een heel goede reden hebt een andere container class te kiezen.

5.2. Twee-dimensionale std::vector

Een std::vector kan elk data type bevatten. Een twee-dimensionale std::vector van type **int** is dan stomweg:

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

Merk op dat er een spatie tussen de scherpe haken sluiten moet staan.

Hieronder wat voorbeeldcode om te zien dat een 2D `std::vector` hetzelfde werkt als een 1D `std::vector`.

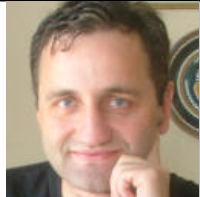
```
std::vector<std::vector<int> > v;

assert(v.size()==0);

v.resize(1);
assert(v.size()==1);
assert(v[0].size()==0);

v[0].resize(1);
assert(v.size()==1);
assert(v[0].size()==1);

v[0][0] = 0;
assert(v[0][0]==0);
```



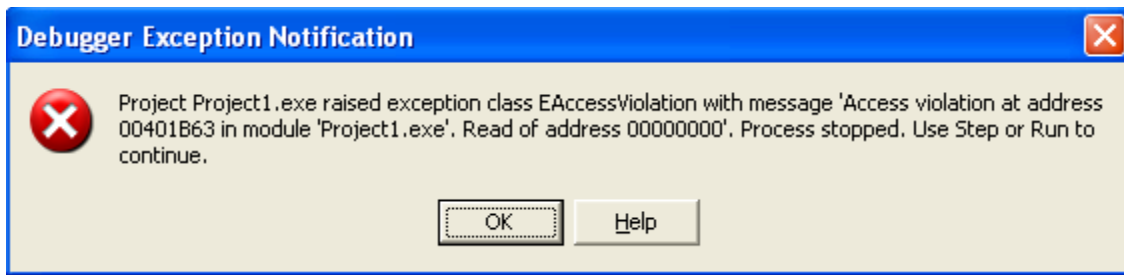
std::vector is een template class. Het data type dat de std::vector opslaat, is de std::vector zelf onbekend. Dit is een van de belangrijkste redenen waarom de std::vector zo belangrijk is.

5.3. Access violation

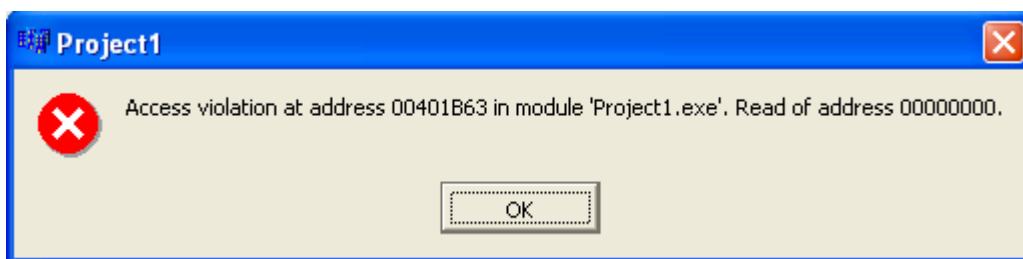
Een van de meest voorkomende en vervelende run-time error is de access violation. Deze treedt op als er wordt gelezen/geschreven van/naar een illegaal geheugenadres. Onderstaande code laat een access violation ontstaan:

```
std::vector<int> v;
for (int i=0; ; ++i)
{
    //Veroorzaak misschien een access violation
    //door voorbij de std::vector te lezen.
    Caption = IntToStr(i) + " : " + IntToStr(v[i]);
}
```

Als de fout wordt gedetecteerd verschijnt onderstaande melding en het schuldige statement wordt aangegeven in de Text Editor.



Bovenstaand gedrag treedt op terwijl we in Kylix werken. Sluiten we Kylix en starten de executable, dan verschijnt de volgende melding:



Deze melding is helemaal niet informatief.

De belangrijkste ergernis aan een access violation is dat deze niet altijd 'optreed'. Een access violation wordt immers niet door het programma zelf, maar door het operating system (Windows of Linux) gedetecteerd. Soms vindt het operating system dit snel, soms laat. Een access violation wordt ook wel een tijdbom genoemd.

Doe jezelf een plezier en leer meteen een assert te schrijven voordat je een index operator aanschrijft:

```
assert( i >=0 );
assert( i < static_cast<int>( v.size() ));
v[i] = /* iets */;
/* iets */ = v[i];
```

De **static_cast** is nodig omdat de methode `size` een **unsigned int** teruggeeft.

5.4. Bitblitting

In plaats van losse pixels te manipuleren, bestaat er ook een snelle manier om een bitmap te tekenen op een andere bitmap. Deze manier van tekenen wordt 'bitblitting' genoemd,

hoewel je dit op het eerste gezicht nergens terugvind.
Bitblitting kan ook transparantie gebruiken en ook groter of kleiner tekenen.

De volgende 3 regels bitblitten op coördinaat (0,0) een bitmap op het Form:

```
Canvas->Draw(0,0,Image1->Picture->Graphic);  
this->Canvas->Draw(0,0,Image1->Picture->Graphic);  
Form1->Canvas->Draw(0,0,Image1->Picture->Graphic);
```

Alledrie de schrijfwijzen doen voor ons hetzelfde. De eerste twee schrijfwijzen zijn echter beter. Waarom wordt duidelijk als we met meerdere Forms gaan werken.



*Er is geen verschil in het gebruik van
Caption = "Hallo";
en
this->Caption = "Hallo";
in een Event van een Form*

Als we van een Image de Property Transparent op true zetten, wordt de bitmap transparent. De kleur van de pixel linksbovenin de bitmap wordt gebruikt als de 'doorzichtige kleur'. Deze kleur is run-time te wijzigen.

Voor een applicatie is het gebruikelijk om met twee soorten plaatjes te werken: een achtergrond en de rest. Elke tijdsstap wordt de Form-vullende achtergrond op het Form geblit, vervolgens alle andere plaatjes. De achtergrond maakt als het ware de oude graphics ongedaan. Deze techniek wordt single-buffering genoemd en je zult zien dat het beeld flikkert ('flickers'). Volgende keer leren we double-buffering, dit geeft een weergave zonder flikkering.

5.5. Volgende week

Double-buffering voor vloeiend uitziende applicaties. En we leren onze eigen data types maken! Daar bovenop leren we nog iets meer STL functies en onze eigen **template** functies te maken.

5.6. Code PompebledRainer

5.6.0. Unit1.h

```
//-----
#ifndef Unit1H
#define Unit1H
//-----
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
#include <QExtCtrls.hpp>
#include <QGraphics.hpp>
#include <QTypes.hpp>
//-----/
/ Richel's #includes
#include <vector>
//-----
class TForm1 : public TForm
{
    __published:    // IDE-managed Components
        TImage *ImagePompebled;
        TImage *ImageBackground;
        TTimer *Timer1;
        void __fastcall FormResize(TObject *Sender);
        void __fastcall Timer1Timer(TObject *Sender);
private:    // User declarations
    std::vector<int> mX;    //X coordinaat
    std::vector<int> mY;    //Y coordinaat
    std::vector<int> mDx;    //Delta x, snelheid in x
    std::vector<int> mDy;    //Delta y, snelheid in y
public:    // User declarations
    __fastcall TForm1(TComponent* Owner);
};

//-----
extern PACKAGE TForm1 *Form1;

//-----
void PaintAll(TImage * const image,
    const char red,
    const char green,
    const char blue );

#endif
```

5.6.1. Unit1.cpp


```
//-----
#include <clx.h>
#pragma hdrstop
#include "Unit1.h"
//-----
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
//-----
//Richel's #includes
#include <cassert>
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
```

```
{
    const int n = 10; //Aantal images
    mX.resize(n);
    mY.resize(n);
    mDx.resize(n);
    mDy.resize(n);

    for (int i=0; i!=n; ++i)
    {
        mX[i] = (i*70) % Form1->ClientWidth;
        mY[i] = (i*90) % Form1->ClientHeight;
        mDx[i] = i+1;
        mDy[i] = i+1;
    }

    ImageBackground->Visible = false;
    ImagePompebled->Visible = false;

    ImageBackground->AutoSize = true;
    ImagePompebled->AutoSize = true;

    ImagePompebled->Transparent = true;

    Caption = "De Pompebled Rainer";

    this->Resize(); //Roept de OnResize Event aan
}
//-----
void __fastcall TForm1::FormResize(TObject *Sender)
{
    ImageBackground->Picture->Bitmap->Width = ClientWidth;
    ImageBackground->Picture->Bitmap->Height = ClientHeight;
    PaintAll(ImageBackground,255,255,255);
}
```

```

}
//-----
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    const int n = mX.size();
    const int maxx = Form1->ClientWidth;
    const int maxy = Form1->ClientHeight;

    //Bitblit de achtergrond
    Canvas->Draw(0,0,ImageBackground->Picture->Graphic);

    for (int i=0; i!=n; ++i)
    {
        //Beweeg de coördinaten
        mX[i]+=mDx[i];
        mY[i]+=mDy[i];
        //Hou de coördinaten in het scherm
        if (mX[i] > maxx) mX[i]=0;
        if (mY[i] > maxy) mY[i]=0;
        //Bitblit de plaatjes
        Canvas->Draw(
            mX[i], //X coördinaat
            mY[i], //Y coördinaat
            ImagePompebled->Picture->Graphic);
    }
    //Form1->Refresh(); //Niet nodig in dit geval
}
//-----
void PaintAll(TImage * const image,
    const char red,
    const char green,
    const char blue )
{
    assert(image!=0);
    assert(image->Picture->Bitmap != 0);
    assert(image->Picture->Bitmap->PixelFormat == pf32bit);

    const int maxx = image->Picture->Bitmap->Width;
    const int maxy = image->Picture->Bitmap->Height;
    for (int y = 0; y != maxy; ++y)
    {
        unsigned char * const myLine
            = static_cast<unsigned char*>(
                image->Picture->Bitmap->ScanLine[y]);
        for (int x = 0; x != maxx; ++x)
        {
            myLine[x*4+2] = red ; //Red

```

```
        myLine[x*4+1] = green; //Green
        myLine[x*4+0] = blue ; //Blue
    }
}
//-----
```

6. Dag 6



Een mogelijk eindprogramma van vandaag.

Onze grafische applicaties geven een knipperend onrustig beeld, omdat deze single-buffered zijn. Vandaag leren we double-buffering, wat een rustig niet-knipperend beeld oplevert. Ook leren we onze eigen template functies te schrijven, enkele STL functies en zien we hoe willekeurig 'willekeurige' getallen zijn. Ook maken we ons eigen simpele data type.

6.0. Template functies

Template functies zijn hetzelfde als normale functies, behalve dat het data type niet gespecificeerd wordt. Dit maakt een functie veel algemener.

Hieronder twee functies die twee 'ietsen' optellen. De eerste is specifiek voor **int**, de tweede is algemeen. De verschillen heb ik in rood aangegeven.

```
int TelOp(const int a, const int b)
{
    return a + b;
```

```

}

template <class T>
T TelOp(const T& a, const T& b)
{
    return a + b;
}

```

Merk het volgende op:

- * Voor een template functie moet aangegeven hoe het template data type heet. Standaard wordt bij simpele functies de letter T hiervoor gebruikt.
- * De functieargumenten worden met behulp van referencing door gegeven. Dit omdat onbekend is welk data type daadwerkelijk gebruikt gaat worden.



*Geef de voorkeur een reference-to-const aan een functie te geven, behalve bij ingebouwde types, zoals **int**.*



Het is mogelijk de functie te optimalizeren over het type dat 'ie als argument krijgt. Zie mijn boek 'Modern C++ Design'.

De template versie van TelOp kun je op dezelfde manier aanroepen als een normale functie.

Hieronder twee functies die de elementen van een `std::vector` optellen. De eerste is specifiek voor `std::vector<double>`, de tweede is algemeen. De verschillen heb ik in rood aangegeven.

```

double TelOp(const std::vector<double>& v)
{
    const int size = v.size();
    double sum = 0.0;
    for (int i=0; i!=size; ++i)
    {
        sum+=v[i];
    }
    return sum;
}

```

```

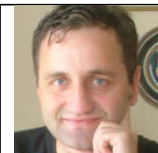
template <class T>
T TelOp(const std::vector<T>& v)
{
    const int size = v.size();
    T sum = 0.0;
    for (int i=0; i!=size; ++i)
    {
        sum+=v[i];
    }
    return sum;
}

```

Merk het volgende op:

- * Deze TelOp functie en de vorige Telop functie kunnen in hetzelfde programma staan. De compiler kan ze aan de hand van hun argumenten uit elkaar houden.
- * De variabele 'size' blijft van het type **int**. De grootte van een std::vector<T> is altijd een (**unsigned**) **int**.

De template versie van TelOp kun je op dezelfde manier aanroepen als een normale functie.



Dit is nog maar een eerste voorbeeld van wat je met templates kunt. Met templates alleen kun je al alle denkbare berekeningen uitvoeren! Dit wordt 'Template Metaprogramming' genoemd.

6.1. STL Header files

C++ heeft een standaard bibliotheek, genaamd de STL. STL staat voor 'Standard Template Library', omdat deze veel met template functies en template klassen werkt. De STL bestaat uit 50 header files, 209 functies en een (mij onbekend aantal) klassen.



Vind het wiel niet opnieuw uit. Gebruik bibliotheken.



Leer de STL kennen.

Het is niet belangrijk alle header files te kennen. We kennen er al twee, hieronder staan ze met vier nieuwe:

Header file naam	Omschrijving
algorithm	std::swap
cassert	assert
cmath	std::sqrt, std::cos, etc.
cstdlib	std::rand, std::srand, std::abs, std::fabs
ctime	std::clock, etc.
vector	std::vector

Een header file bekijken is heel gemakkelijk: ga op een #include op de naam van de header file staan en doe 'Open File at cursor' (rechtermuisknop of CTRL-ENTER). Als er niks gebeurt, kan deze header file niet worden gevonden.

Standaard C++ header files zien er grotendeels obscuur uit! Leer ze op een gepast manier te scannen.

Als je een eerste header file in bent, kom je vaak kom je eerst in een andere header file (een 'wrapper'), waarin je nog eens CTRL-ENTER moet doen. Hieronder staat de header file van 'algorithm'. De header file 'stlport\algorithm' blijkt niet te bestaan, 'oldstl\algorithm.h' wel.

```
/*
 * Wrapper header file used to select STL via defines.
 * Default is STLport.
 */

/*
 * C/C++ Run Time Library - Version 11.0
 *
 * Copyright (c) 2002 by Borland Software Corporation
 * All Rights Reserved.
 */

#ifdef _USE_OLD_RW_STL
# include <oldstl\algorithm.h>
#else
# include <stlport\algorithm>
#endif

#if !defined( USING_STD_NAMES ) && defined( cplusplus)
```

```
using namespace std;
#endif /* __USING_STD_NAMES__ */
```

Het bekijken van header files is nuttig als je op zoek bent naar een bepaalde STL functie of STL klasse.

6.2. STL functies

De belangrijkste STL functies staan hier beschreven:

Functienaam	Omschrijving
std::abs	Verkrijg de absolute waarde van een heel getal
std::clock	Verkrijg de systeemtijd
std::cos	Bereken de cosinus van een double
std::fabs	Verkrijg de absolute waarde van een gebroken getal
std::rand	Verkrijg een willekeurig getal van 0 tot 32767
std::sqrt	Bereken de vierkantswortel van een double
std::srand	Zet de random number generator seed
std::swap	Verwissel twee variabelen van type T

Als je de naam van de functie intoets, je cursor in deze naam zet en op F1 drukt, krijg je hulp over hoe de functie werkt.



*Geef de voorkeur aan STL
functies/klassen/algoritmen in plaats van
'geknutselde' code te gebruiken.*

Ook is het soms nuttig om met de rechter muis op een functie te klikken en dan 'Find Declaration' te kiezen. Nu wordt soms de declaratie van de functie of klasse getoond.

De STL functie declaraties zien er obscuur uit. Dit omdat C++ compatible met C wil zijn. Zo ziet bijvoorbeeld de declaratie van std::rand er zo uit:

```
int __RTENTRY __EXPFUNC rand(void);
```

Lees dit als stond er:

```
int rand();
```


Standaard C++ functies zien er grotendeels obscuur uit! Leer ze op een gepast manier te scannen.



Geloof niet dat de standaard bibliotheek ideaal voor alles is!



Leer (later) de Boost bibliotheek!

6.3. Random numbers

De functie `std::rand` en `std::srand` worden gebruikt om pseudo-willekeurige ('pseudo-random') getallen te genereren. Pseudo-willekeurig, want een computer kan geen 'echte' willekeurige getallen produceren (hoewel hier tegenwoordig quantum-chips voor zijn die dit wel kunnen!). Inplaats daarvan produceert de computer een serie opeenvolgende getallen. De reeks opeenvolgende getallen is zo, dat als je een getal weet, je niet het volgende getal kunt voorspellen.

Maar deze reeks getallen zal altijd hetzelfde zijn, voor dezelfde 'seed'. De seed ('zaadje') is een soort startpunt van de random number sequentie. De seed zet je met `std::srand`.

Onderstaande code zal geen assertfouten opleveren:

```
const int n = 1000;      //Aantal testen
const int seed = 12345;  //Of welke waarde dan ook

std::vector<int> v;
v.resize(n);

{ //Initialiseer v met 'willekeurige' getallen
  std::srand(seed);
  for (int i=0; i!=n; ++i)
  {
    v[i] = std::rand();
  }
}
{ //Lees of 'willekeurige' getallen gelijk zijn aan v
  std::srand(seed);
```

```
for (int i=0; i!=n; ++i)
{
    assert(v[i] == std::rand());
}
}
```

Onderstaande code bevat een denk-fout:

```
//Zet de seed met een random number
std::srand(std::rand()); //Denkfout!
```

Om de seed te kunnen zetten wordt deze vaak gezet met behulp van de systeemtijd:

```
std::srand(std::clock());
```

Zolang een programma niet vaker dan eens per seconde wordt gestart, heeft deze een unieke seed.

6.4. Een eigen data type

Soms zouden we graag enkele samenhangende variabelen bundelen. Dit kan met behulp van een **struct**.

```
struct Coördinaat
{
    int mX;
    int mY;
};
```

Bovenstaande is een (klasse)declaratie van een data type genaamd 'Coördinaat' die uit twee ints bestaat. Merk de puntkomma op bij het sluiten van de declaratie.

Vanaf deze declaratie kunnen we werken met dit data type alsof het een String zou zijn:

```
Coördinaat c;
c.mX = 10;
c.mY = c.mY;
```

Is het data type goed gedeclareerd, dan zal Class Browsing ons helpen als we de punt in toetsen.

Ook is het mogelijk een `std::vector<Coördinaat>` te maken.



Wees duidelijk in wat voor **struct** je wilt maken. Als je er geen zinnige naam aan kunt geven, is dit vaak een zwakte in het ontwerp.

6.5. Double-buffering

Het verschil tussen single-buffering en double-buffering, is dat bij single-buffering alle Images op het Form worden gebitblit, bij double-buffering wordt er eerst naar een 'buffer' gebitblit. Deze buffer, die het complete plaatje bevat, wordt vervolgens op het Form gebitblit.

```
//Background
ImageBuffer->Canvas->Draw(
    0 , 0,
    ImageBackground->Picture->Graphics);

//Spelers
ImageBuffer->Canvas->Draw(
    x1 , y1,
    Image1->Picture->Graphics);
ImageBuffer->Canvas->Draw(
    x2 , y2,
    Image2->Picture->Graphics);

//En naar het Form
Canvas->Draw(
    0,0,
    ImageBuffer->Picture->Graphics);
```

Double-buffering is niet veel ingewikkelder dan single-buffering, maar het voorkomt knippering.

6.6. Volgende week

Op het gebied van graphics is er niets meer te leren. Wat we nu van struct af weten is heel karig, maar zal de eerste stap zijn in het object georiënteerd programmeren. We leren van onze struct een volwaardige klasse te maken. Ook leren we ons Form als een klasse te zien, waardoor we met meerdere Forms kunnen gaan werken.

7. Dag 7



Een mogelijk eindprogramma van vandaag.

Vandaag leren we de basis van een klasse. Hierdoor leren we ons Form als klasse te zien en kunnen we met meerdere Forms werken.

7.0. Een klasse

Een klasse is een data type dat ervoor zorgt dat 'ie altijd in een geldige staat is.

Onderstaand data type garandeert niet in een geldige staat te zijn:

```
struct Klok
{
    int mUren;
    int mMinuten;
};
```

Bovenstaand data type kan immers gemakkelijk ongeldig gemaakt worden:

```
Klok k;  
k.mUren      = 1000; //Oei! Een dag heeft maar 24 uur!  
k.mMinuten = 1000; //Oei! Een uur heeft maar 60 minuten!
```

Een klasse beschermt zijn data. Dit wordt 'encapsulating' genoemd. Een klasse kan zijn data beschermen doordat zijn leden **public** en **private**) kunnen zijn. Het idee is dat de variabelen **private** zijn, en dat deze met gecontroleerde **public** methoden kunnen worden gelezen/veranderd.

```
struct Klok  
{  
    public: //Overbodig  
  
    int GetUren()  
    {  
        assert(mUren >=0 );  
        assert(mUren < 12); //Keuze van de ontwerper  
        return mUren;  
    }  
    int GetMinuten()  
    {  
        assert(mMinuten >=0 );  
        assert(mMinuten < 60);  
        return mMinuten;  
    }  
  
    void SetUren(const int uren)  
    {  
        assert(uren >=0 );  
        assert(uren < 12); //Keuze van de ontwerper  
        mUren = uren;  
    }  
    void SetMinuten(const int minuten)  
    {  
        assert(minuten >=0 );  
        assert(minuten < 60);  
        mMinuten = minuten;  
    }  
    private:  
    int mUren;  
    int mMinuten;  
};
```

Merk het volgende op:

* De schrijfwijze lijkt heel erg op die van ons Form.

Alleen staat de code in een stuk tekst, inplaats van over twee bestanden verdeeld.

- * Er zijn twee secties: de **public** en **private** sectie. Deze worden aangegeven met een dubbele punt op het einde.
- * Het keyword om deze klasse te declareren is **struct**.
- * Een **struct** staat standaard op **public**. Daarom is de regel **'public:'** overbodig.
- * Het aantal uren en minuten zijn **private**. Deze kunnen niet direct worden gewijzigd.
- * Er zijn Get- en Setfuncties. Dit worden 'getters' en 'setters' genoemd.
- * De methoden mogen wel bij het **private** gedeelte.
- * De Setfuncties controleren of deze geldige input binnen krijgen.
- * De Getfuncties controleren of deze geldige waarden hebben.



*Maak je lidvariabelen private, behalve in C-style **structs**.*

Als deze klasse goed ingetypt is, dan kunnen we met Class Browsing zien, dat we inderdaad alleen het **public** gedeelte kunnen zien en kiezen.

```
{
    Klok k;
    k.
}
Function int Klok::GetUren ()
Function int Klok::GetMinuten ()
//-- Function void Klok::SetUren (const int)
Function void Klok::SetMinuten (const int)
```

*De Class Browser laat alleen het **public** gedeelte van onze klasse zien.*

Schrijven we toch onderstaande code, dan treedt de melding eronder op.

```
Klok k;
k.mUren = 69;
```

```
[C++ Error] Unit1.cpp(53): E2247 'Klok::mUren' is not
accessible
```



*Maak het gemakkelijk een klasse goed te gebruiken
en maak het moeilijk een klasse verkeerd te
gebruiken.*

7.1. Constructor

De Klok klasse bevat echter een grote fout. Onderstaande code gaat (vaak) mis:

```
Klok k;  
const int uren = k.GetUren();
```

Bovenstaande code gaat mis, omdat het aantal uren niet geïnitieerd is.

Hieronder staat een niet-oplossing:

```
struct Klok  
{  
    void Initialiseer() { mUren = 0; mMinuten = 0; }  
    //De rest  
};
```

Het probleem met een initialiseer methode is dat de gebruiker van de klasse deze kan vergeten aan te roepen.

Er bestaat een speciale methode om een klasse te initialiseren op het moment dat deze gecreeerd wordt: de constructor.

```
struct Klok  
{  
    Klok()  
    {  
        mMinuten = 0;  
        mUren = 0;  
    }  
    //De rest  
};
```

Merk het volgende op:

* Een constructor is een methode met dezelfde naam als de klasse.

* Een constructor heeft geen return data type.

Deze constructor, de constructor zonder argumenten, wordt de default constructor genoemd.

Voor de constructor is ook een alternatieve, betere schrijfwijze: de initialisatielijst.

```
struct Klok
{
    Klok() : mMinuten(0), mUren(0)
    {
        //Ha, alles is al geïntialiseerd
    }
    //De rest
};
```

Merk het volgende op:

- * Na de klasse naam en ronde haken, wordt er een dubbele punt geschreven.
- * De variabelen worden niet met het '='-teken geïntialiseerd, maar met de waarde tussen ronde haken.
- * De variabelen zijn gescheiden met komma's.
- * Na het laatste argument staat er niets meer tot de accolade openen van de constructor



Geef de voorkeur aan de initialisatielijst boven het gebruik van toekenningen (met de '='-operator, 'assignment')

Het is mogelijk om meerdere constructoren te definiëren:

```
struct Klok
{
    Klok() : mMinuten(0), mUren(0) { }
    Klok(const int minuten, const int uren)
        : mMinuten(minuten), mUren(uren) { }
    //De rest
};
```

Met deze twee constructoren kunnen we onderstaande code schrijven:

```
Klok k1; //Roept de default constructor aan
Klok k2(10,10);
```


Het zou fijn zijn als we zouden kunnen kijken uit welke constructoren we kunnen kiezen. Onderstaand screenshot zou een lijst moeten tonen met welke constructoren er zijn, maar doet dit helaas niet altijd.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Klok k1{
}
```

Om toch te kunnen spieken, gebruik dan even onderstaande truc.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Klok k1{
}
```

Merk het volgende op:

- * Er zijn drie (!) mogelijke constructoren.
- * De bovenste constructor is de default constructor zonder argumenten
- * De tweede constructor toont de **ints** zonder **const**.
- * De derde constructor wordt de copy constructor genoemd. Deze heeft een Klok als argument nodig. Deze Klok wordt met reference-by-**const** doorgegeven.

De copy constructor behandelen we niet, maar onderstaande code is mogelijk door deze automatisch aangemaakte constructor:

```
const Klok k1; //Default constructor
const Klok k2(k1); //Copy constructor
```

Ook String en std::vector hebben meerdere constructoren. Met F1 kun je uitvinden hoe deze werken.

7.2. Const methoden

Een methode kan zelf ook **const** zijn. Als een methode const is, garandeerd deze methode dat de variabelen van de klasse niet worden gewijzigd. Getters zijn bijna altijd **const**.

```
struct Klok
{
    int GetUren()      const { /* code */ }
    int GetMinuten()  const { /* code */ }
    //Rest
};
```



Gebruikt altijd **const** waar mogelijk!

Schrijven we toch onderstaande code, dan verschijnt de foutmelding daaronder.

```
struct Klok
{
    int GetUren() const { mUren = 0; return mUren; }
    //Rest
};
```

[C++ Error] Unit1.h(40): E2024 Cannot modify a const object

Zouden we GetUren niet **const** maken, dan levert onderstaande code de foutmelding daaronder op:

```
const Klok k;
const int uren
    = k.GetUren(); //GetUren is onterecht geen const methode
```

[C++ Warning] Unit1.cpp(21): W8037 Non-const function Klok::GetUren() called for const object



Geef de voorkeur aan compile- en link-time errors boven run-time errors.

Zou Klok niet veranderd mogen worden (omdat deze bijvoorbeeld in een bibliotheek zit), dan moeten we

```
//Klok is uit de VBL (Very Bad Library)
const Klok k;
//GetUren is onterecht geen const methode
//Dan maar een const_cast.
```

```
        //Grumble, grumble...  
const int uren = (const_cast<Klok*>(&k))->GetUren();
```

Bovenstaande code gaan we niet dieper op in, maar merk wel op dat er een Klok* wordt gemaakt en dat GetUren met een pijltje inplaats van met een punt wordt aangesproken.

7.3. Klasse definitie en declaratie scheiden

Ons Form heeft de klasse definitie en declaratie gescheiden. Dat kunnen wij ook. Onderstaande triviale klasse kunnen we uit elkaar trekken, zie de code daaronder.

```
//Een klasse definitie  
struct MijnKlasse  
{  
    MijnKlasse() : mX(0) {}  
    void SetX(const int x) { mX = x; }  
    int GetX() const { return mX; }  
    private:  
    int mX;  
};
```

```
//Een klasse declaratie  
struct MijnKlasse  
{  
    MijnKlasse();  
    void SetX(const int x);  
    int GetX() const;  
    private:  
    int mX;  
};  
  
//Methode definities  
MijnKlasse::MijnKlasse()  
    : mX(0)  
{  
  
}  
  
void MijnKlasse::SetX(const int x)  
{  
    mX = x;  
}
```

```
int MijnKlasse::GetX() const
{
    return mX;
}
```

Merk op dat een methode definitie hetzelfde is als een functie definitie, behalve dat voor de naam van de methode er 'MijnKlasse::' voor moet worden geschreven. Zo kan de compiler zien dat dit geen functie is, maar een methode.

De klasse declaratie staat vaak in een header (.h) file, de methode definities in implementatie (.cpp) bestanden.

7.4. Verschil tussen class en struct

Onderstaande twee stukken code zijn equivalent:

```
class MijnKlasse
{
    public:
    //Code
};
```

```
struct MijnKlasse
{
    //Code
};
```

Het keyword **struct** geeft een klasse aan, waarbij de bovenste sectie standaard **public** is. **class** zet de bovenste sectie standaard op **private**. Omdat vaak het **public** gedeelte bovenaan een klasse staat, is **struct** simpelweg minder typwerk.



*De variabelen in een **class** zijn standaard **private**, omdat het afschermen van data een van de belangrijkste taken van een klasse is.*



*De variabelen in een **struct** zijn standaard **public** om compatibel met C te blijven.*

Zowel een **struct** als een **class** worden een klasse genoemd. Het keyword **struct** bestond al in C, als een bundeling van variabelen (bijvoorbeeld de **struct** Coördinaat!). Een **struct** zonder methoden wordt soms een 'C-style struct' genoemd, of een 'POD-object' ('Plain Old Data'-object).

7.5. De debugger

Onderstaande regel roept de default constructor aan van Klok.

```
const Klok k;
```

Dit is ook te zien met de debugger. Zet een breakpoint op deze regel. Als het programma hierop stopt, kies dan 'Trace Into' (F7). De debugger neemt je naar de constructor.

7.6. Het laatste woord over de klasse Klok

De klasse Klok is zeker niet geniaal:

- * De klasse Klok kan beter bestaan uit twee data typen, genaamd Minuut en Uur. Als Minuut en Uur voor zichzelf zorgen, hoeft Klok weinig meer te checken.
- * Zou je Klok onderverdelen in Minuut en Uur, dan zie je dat Minuut en Uur eigenlijk hetzelfde zijn, behalve dat Minuut tot 60 gaat en uur tot 12 (of 24). Dan kun je beter een Bereik klasse kunnen gebruiken voor beide.
- * De klasse Klok had beter Tijd kunnen heten. Zou de klasse ook een klok op het scherm tekenen, dan zou Tijd een lid van Klok moeten zijn.



*Geef een entiteit een coherente
verantwoordelijkheid.*

7.7. Volgende week

Nu we weten wat een constructor is, zullen we de constructor van ons Form bekijken, aanpassen en meerdere Forms gaan aanmaken. Ook gaan we een platform-onafhankelijke Console Applicatie maken en kennismaken met de `std::string`.

8. Dag 8

```
ZORK I: The Great Underground Empire
Copyright (c) 1981, 1982, 1983 Infocom, Inc. All rights reserved.
ZORK is a registered trademark of Infocom, Inc.
Revision 88 / Serial number 840726

West of House
You are standing in an open field west of a white house, with a boarded front
door.
There is a small mailbox here.

>open mailbox
Opening the small mailbox reveals a leaflet.

>read leaflet
(Taken)
"WELCOME TO ZORK!"

ZORK is a game of adventure, danger, and low cunning. In it you will explore
some of the most amazing territory ever seen by mortals. No computer should be
without one!"

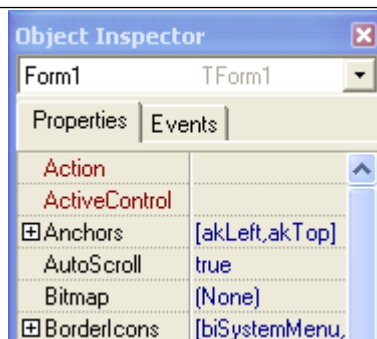
>
```

Een mogelijk eindprogramma van vandaag.

Vandaag leren we met meerdere Forms werken, de constructor van een Form te wijzigen en hoe we een Console Applicatie maken.

8.0. De TForm1 constructor

Ons Form is ook een klasse. De naam van de klasse is te zien in de Object Inspector. Standaard is dit TForm1.



De Object Inspector laat ons zien dat ons Form de naam Form1 heeft en van het data type TForm1 is.

Nu we de klassenaam van ons Form weten, kunnen we de constructoren bekijken.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TComponent * Owner
    const TForm1 &
    TForm1 f;
}
```

Class Browsing laat ons de twee constructors van TForm1 zien

TForm1 heeft twee constructors:

- * 'TComponent * Owner': een TComponent* die de beheerder wordt van het nieuwe TForm1.
- * 'const TForm1&': een copy constructor.

De eerste constructor kan ons bekend voorkomen. Deze hebben we altijd al in Unit1.cpp gehad:

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
}
```

De declaratie van deze constructor staat in Unit1.h.

Je kunt de constructor van TForm1 ook argumenten erbij geven, als je dit in zowel Unit1.cpp en Unit1.h wijzigd. Als we een TForm aan kunnen maken (staat hieronder), dan leren we deze constructor aan te passen.

8.1. Een TForm aanmaken

Onderstaande poging om door een druk op een TButton een nieuw Form aan te maken en te tonen lukt niet:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TForm1 f; //Lukt niet
    f.Show();
}
```

```
[C++ Error] Unit1.cpp(20): E2459 VCL style classes must be
constructed using operator new
```

Door F1 op de foutmelding te doen, verschijnt de Help:

E2459 Delphi style classes must be constructed using operator new Compiler error

Delphi style classes cannot be statically defined. They have to be constructed on the heap.

Example:

```
void foo(void)
{
    TObject    o1;        // Error;
    TObject    *o2 = new TObject();
}
```

Deze Help is blijkbaar uit Delphi (het Windows en Pascal zusje van Kylix) gehaald.

Dankzij deze informatie, wordt de volgende manier geprobeerd:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TForm1 * f = new TForm1();
    f->Show();
}
```

De volgende fout treedt op:

```
[C++ Error] Unit1.cpp(20): E2285 Could not find a match for
'TForm1::TForm1()'
```

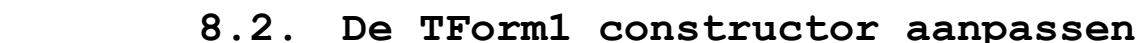
Inderdaad, we hebben gezien dat TForm1 in de constructor een 'TComponent* Owner' mee moet krijgen. Ons Form is gelukkig ook een TComponent pointer:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TForm1 * f = new TForm1(this);
    f->Caption = "Ik ben nieuw";
    f->Show();
}
```

Merk het volgende op:

* We hebben zojuist onze eerste pointer aangemaakt. Dit doe je met **new**. Alleen om een Form te laten verschijnen leren

- * Het keyword **this** laat een klasse een pointer naar zichzelf geven. Een TForm1 pointer in dit geval.
- * De constructor van TForm1 heeft een TComponent pointer nodig, maar krijgt een TForm1 pointer binnen. Dit komt omdat TForm1 een afgeleide ('derived') klasse is van TComponent. Dit wordt overerving ('inheritance') genoemd en gaat te ver voor deze cursus.
- * De pointer wordt niet gedelete. Dit is, omdat de Owner (in dit geval **this**) ervoor zorgt dat **delete** op het juiste moment wordt aangeroepen.



113

Bijvoorbeeld, stel je wilt elk Form een paswoord meegeven, die in de constructor wordt geïnitialiseerd. Voeg de de in rood aangeven code toe aan de header file van TForm1:

```
class TForm1 : public TForm
{
__published:    // IDE-managed Components
    //Kylix dingen
private:    // User declarations
    String mPaswoord;
public:    // User declarations
    __fastcall TForm1(TComponent* Owner, const String& s);
};
```

De methode definitie van de TForm1 constructor ziet er dan als volgt uit:

```
__fastcall TForm1::TForm1(
    TComponent* Owner,
    const String& s)
    : TForm(Owner),
        mPaswoord(s)
```

```
{
}
}
```

Laat in de initialisatielijst de regel 'TForm(Owner)' staan. Anders krijg je onderstaande foutmelding:

```
[C++ Error] Unit1.cpp(14): E2251 Cannot find default
constructor to initialize base class 'TForm'
```

De Help kan verklaren wat er mis is.

We kunnen nu nieuwe TForm1s maken en als onderstaand testen.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TForm1 * f = new TForm1(this, "iloverichel");
    f->Show();
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{
```

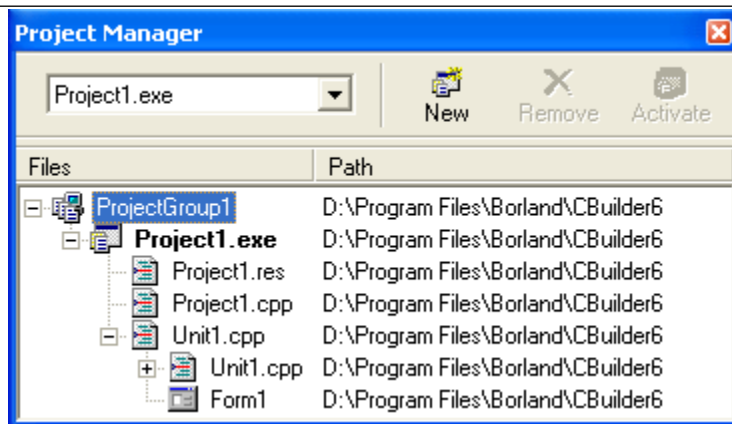
```
ShowMessage(mPaswoord);
}
```

Merk op dat het eerste Form dat in beeld komt een leeg mPaswoord heeft.

8.3. Werken met meerdere verschillende Forms en/of Units

Tot nu toe hebben onze Projects altijd uit een enkele Unit bestaan. Willen we met meerdere Units gaan werken, dan moeten we de Project Manager leren gebruiken.

Met 'View | Project Manager' of CTRL-ALT-F11 kunnen we de Project Manager zien.



De Project Manager van een Project (genaamd Project1) met een enkele Unit (Unit1), die een Form bevat (Form1).

Kiezen we 'File | New | Form', dan verschijnt er een nieuw Form, Form2 genaamd, van het data type 'TForm2'. Deze is ook meteen te zien in de Project Manager. #includen we de header file van dit Form, Unit2.h, dan kunnen we deze in TForm1 aanroepen, zoals bijvoorbeeld in onderstaand voorbeeld:

```
#include "Unit2.h"

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TForm2 * f = new TForm2(this);
    f->Show();
}
```

We kunnen ook een 'lege' Unit aanmaken, om bijvoorbeeld een klasse in te plaatsen. Doe 'File | New | Unit'.

Er is nu een nieuwe Unit aangemaakt, bestaande uit 'Unit3.cpp' en 'Unit3.h'.

Deze code bestaat standaard uit onderstaande tekst:

```
//-----  
#ifndef Unit3H  
#define Unit3H  
//-----  
//Declaraties hiero  
#endif
```

```
//-----  
#pragma hdrstop  
  
#include "Unit3.h"  
//-----  
//Definities hiero  
#pragma package(smart_init)
```

De compiler gaat klagen als je de bovenste #pragma weghaalt, maar beide #pragma's mag je weghalen (hoewel het geen kwaad kan, en het soms beter is, ze te laten staan).

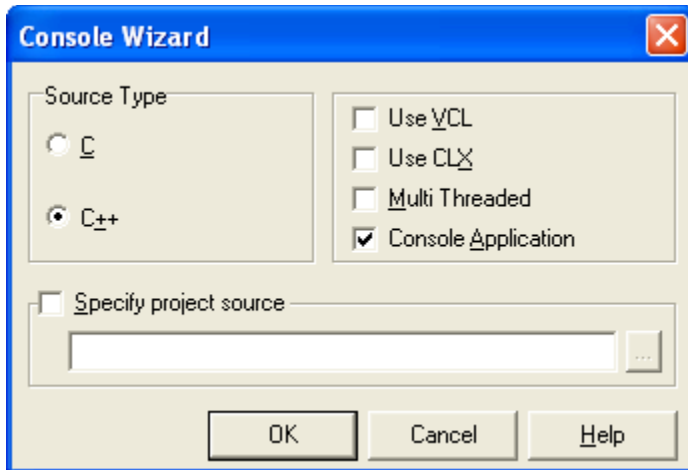
Met de Project Manager kunnen we Units toevoegen en verwijderen uit een Project. Dit is zeer nuttig, omdat je nu Units kunt hergebruiken, bijvoorbeeld een 'About Box'. Een Unit kan prima onder meerdere Projecten vallen.

8.4. Een console applicatie

Een console applicatie is een applicatie die alleen via het toetsenbord aangestuurd kan worden en die alleen letters/tekens gebruikt.

Console applicaties zijn nuttig om hun platform-onafhankelijk, indien je alleen de STL gebruikt (dus geen Kylix-specifieke functies!).

Om een console applicatie te starten, doe 'File | Close All', dan 'File | New | Other', kies dan onder tabblad 'New' de 'Console Wizard' en stel deze als onderstaand in:



De code die dan verschijnt is als volgt:

```
//-----  
  
#pragma hdrstop  
  
//-----  
  
#pragma argsused  
int main(int argc, char* argv[])  
{  
    return 0;  
}  
//-----
```

Dit kun je strippen tot onderstaande, minimale correcte programma:

```
int main()  
{  
  
}
```

Als je het programma start, dan verschijnt er na compilen en linken heel kort een venstertje, dat automatisch sluit.

Onderstaande code laat een 'Hello World' programma zien, waarvan het programma wacht op een enter.

```
#include <iostream>  
  
int main()  
{
```

```
std::cout << "Hello world" << std::endl;
std::cin.get(); //Wacht op een enter
}
```

Merk het volgende op:

- * Een console applicatie start in de functie main.
- * De functie main geeft een **int** terug. Dit is de foutcode van het programma (waarbij een nul 'geen fout opgetreden' betekent).
- * Er zou op het einde '**return 0;**' moeten staan, maar dit staat er niet en de compiler klaagt niet. Dit omdat de C++ standaard zegt dat de accolade sluiten van main hetzelfde effect als '**return 0;**' moet hebben.
- * Er moet een header file worden ge-#include genaamd 'iostream'. 'iostream' staat voor 'Input/Output Stream'.
- * 'std::cout' staat voor 'Character Out', 'std::cin' staat voor 'Character In'.
- * Met std::cout kun je 'karakters naar het beeldscherm laten stromen'. De '<<' operator wordt de 'stream out' operator genoemd.
- * 'std::endl' staat voor 'End line' en zorgt ervoor dat de tekst op een volgende regel verder gaat.
- * std::cin is een klasse met een get methode. De get methode zorgt dat er gewacht wordt op een enter.

std::cout wordt ook wel een 'output stream' genoemd, std::cin een 'input stream'. Streams worden in deze cursus niet behandeld.

8.5. std::string

Onze String klasse is niet platform onafhankelijk. Het heeft weinig zin een platform-onafhankelijke console applicatie te bouwen met onze platform-afhankelijke String klasse.



Geef de voorkeur aan de STL boven andere bibliotheken en eigengemaakte code.

Hieronder een voorbeeld van het gebruik van std::string:

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

```

int main()
{
    const std::string s = "Hello world";
    std::cout << s << std::endl;
    std::cin.get();
}

```

Prefereer het gebruik van `std::string` boven `String`. Gelukkig kunnen beide klassen gemakkelijk naar elkaar worden geconverteerd, zoals onderstaande code laat zien.

```

#include <iostream>
#include <string>
#include <cassert>
#include <CLX.h> //De CLX header

int main()
{
    const String s1 = "Hello World";
    const std::string s2(s1.c_str());
    const String s3(s2.c_str());

    assert(s1==s3);
    assert(s1.Length() == s2.size());
    assert(s1[1] == s2[0]);

    std::cout
        << s1 << std::endl
        << s2 << std::endl
        << s3 << std::endl;
    std::cin.get();
}

```

Merk het volgende op:

- * de methode `c_str` produceert voor beide string klassen een 'char *', die in beide hun constructoren gebruikt kan worden. Een 'char*' wordt ook wel een 'C-style string' genoemd.
- * Het aantal tekens in een `String` is met de methode 'Length' te verkrijgen, bij `std::string` heet deze methode 'size'.
- * De `String` begint te tellen vanaf index 1, de `std::string` vanaf index 0.

8.6. En nu verder

De console applicatie is het type applicatie waar de meeste boeken mee beginnen. Ook zijn voor console applicaties de meeste programmeeromgevingen te vinden. Vanaf dit punt kan ook zonder Kylix worden gewerkt. Dit was een van mijn doelen van de cursus.

We hebben in de cursus ongeveer 50% van de C++ basis behandeld. Maar in principe kun je nu elke niet-netwerk niet-internet programma schrijven.

Iedereen die meer over C++ wil weten, kan nu verder met de vele literatuur over C++ en de vele C++ programmeeromgevingen.

9. Appendix A: Een plaatje laten stuiteren

Onderstaande code maakt gebruik van de Tag Property. Een Tag is een getalletje waar nooit iets mee gedaan wordt.

Ik laat de dingen die ik tijdens design-time heb ingesteld geheim. Ook is de code expres niet kogelvrij.

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    //Beweeg het plaatje
    Image1->Left = Image1->Left + LabelXstap->Tag;
    Image1->Top  = Image1->Top  + LabelYstap->Tag;

    //Check of het plaatje niet uit het beeld gaat.
    if (Image1->Left < 0)
    {
        //Links uit beeld, stuiter in x richting
        LabelXstap->Tag = -LabelXstap->Tag;
    }
    if (Image1->Left + Image1->Width > Form1->ClientWidth)
    {
        //Rechts uit beeld, stuiter in x richting
    }
}
```



```
        LabelXstap->Tag = -LabelXstap->Tag;
    }
    if (Image1->Top < 0)
    {
        //Boven uit beeld, stuiter in y richting
        LabelYstap->Tag = -LabelYstap->Tag;
    }
    if (Image1->Top + Image1->Height > Form1->ClientHeight)
    {
        //Onder uit beeld, stuiter in y richting
        LabelYstap->Tag = -LabelYstap->Tag;
    }
}
```

10. Appendix B: Maar wat is dat dan?

Inderdaad, er is meer code dan alleen de Events die je zelf maakt. Voor de liefhebbers leg ik deze uit.

Onderstaande code (zonder regelnummers) staat in **Unit1.cpp**:

```
//-----  
1) #include <clx.h>  
2) #pragma hdrstop  
  
3) #include "Unit1.h"  
//-----  
4) #pragma package(smart_init)  
5) #pragma resource "*.xfm"  
6) TForm1 *Form1;  
//-----  
7) __fastcall TForm1::TForm1(TComponent* Owner)  
8)     : TForm(Owner)  
{  
}  
//-----  
  
9) void __fastcall TForm1::Button1Click(TObject *Sender)  
{  
  
}  
//-----
```

- 1) Roep de CLX bibliotheek aan. 'CLX.h' is een algemene header file, vandaar de '<' en '>'.
- 2) Zorg dat compilen mogelijk sneller gaat.
- 3) Roep de header file aan van onze Unit. 'Unit1.h' is een lokale header file, vandaar de dubbele aanhalingstekens.
- 4) Gebruik de Packages zoals beschreven in de Project Options.
- 5) Gebruik de .xfm als resource van dit Form.
- 6) Declareer een globale pointer naar het type van ons Form. Ons Form is van het data type TForm1.
- 7) De constructor van ons TForm1. Heeft als argument een TComponent pointer naar de Owner van het Form, in dit geval Application.
- 8) Construct de parent class constructor (TForm) met de Owner.
- 9) De definitie van de methode Button1Click. Dit is een methode van TForm1 en heeft als argument een TObject pointer naar Sender. Sender kan TObject zijn dat deze methode aanroept. De methode is **void**, ofwel deze heeft geen return waarde. **__fastcall** is een Borland extensie die nodig is voor TForms.

Onderstaande code (zonder regelnummers) staat in **Unit1.h**:

```
//-----  
1) #ifndef Unit1H  
2) #define Unit1H  
//-----  
3) #include <Classes.hpp>  
4) #include <QControls.hpp>  
5) #include <QStdCtrls.hpp>  
6) #include <QForms.hpp>  
//-----  
7) class TForm1 : public TForm  
8) {  
9)     __published:      // IDE-managed Components  
10)         TButton *Button1;  
11)         void __fastcall Button1Click(TObject *Sender);  
12)     private:          // User declarations  
13)     public:           // User declarations  
14)         __fastcall TForm1(TComponent* Owner);  
15) };  
//-----  
16) extern PACKAGE TForm1 *Form1;  
//-----  
17) #endif
```

- 1) #include guard
- 2) #include guard
- 3) Roep header file 'Classes.hpp' aan. Dit is een algemene header file, vandaar de '<' en '>'.
- 4) Roep header file 'QControls.hpp' aan. Dit is een algemene header file, vandaar de '<' en '>'.
- 5) Roep header file 'QStdCtrls.hpp' aan. Dit is een algemene header file, vandaar de '<' en '>'.
- 6) Roep header file 'QForms.hpp' aan. Dit is een algemene header file, vandaar de '<' en '>'.
- 7) Start de klasse declaratie van TForm1. Deze erft public over van TForm.
- 8) Accolade openen van klasse declaratie.
- 9) Start van het gedeelte dat door Kylix wordt beheerd.
- 10) Kylix beheert Button1, een TButton pointer.
- 11) Kylix beheert de Button1Click Event.
- 12) Hierachter kunnen de private methode declaraties staan.
- 13) Hierachter kunnen de public methode declaraties staan.
- 14) Declaratie van de constructor van TForm1.
- 15) Accolade sluiten van klasse declaratie.
- 16) Declareer een globale pointer genaamd Form1 van het type TForm1.
- 17) #include guard

11. Appendix C: Sneltoetsen

Effect	Sneltoets
Compileer programma	Alt-F9
Find	CTRL-F
Indent blok naar links	CTRL-SHIFT-U
Indenk blok naar rechts	CTRL-SHIFT-I
Insert code template	CTRL-J
Object Inspector	F11
Object Treeview	ALT-SHIFT-F11
Open File At Cursor	CTRL-ENTER
Project Manager	CTRL-ALT-F11
Project Options	CTRL-SHIFT-F11
Replace	CTRL-R
Toggle tussen accolades	ALT-[en ALT-]
Toggle tussen blokhaken	ALT-[en ALT-]
Toggle tussen haakjes	ALT-[en ALT-]
Toggle tussen header en implementatie file	CTRL-F6
Toggle tussen Text Editor en Form	F12
Trace Into	F7
Run program	F9
Reset program	CTRL-F2
Start programma	F9
Step over	F8

12. Appendix D: Overzicht

Een overzicht van de belangrijkste begrippen.

12.0. Cast

Een conversie van een data type naar een ander data type. Hieronder vallen de vier C++ casts (**static_cast**, **const_cast**, **dynamic_cast** en **reinterpret_cast**) en de C stijl cast.

```
double d = static_cast<double>(i);
```

12.1. Code

De tekst waaruit je programma bestaat.

12.2. Compiler

Het programma dat onze code vertaalt naar object-code. De compiler doet zijn werk na de pre-compiler. De linker koppelt alle object-code aan elkaar tot een werkende applicatie.

12.3. Data type

Het soort gegevens dat in een variabele kan worden opgeslagen. Elke variabele bestaat uit een data type en een identifier.

```
int x; //Declaratie van de variabele met data type int
      //en met naam x. x kan hele getallen opslaan.

String s; //Declaratie van de variabele met data type
          //String. String kan woorden opslaan.
```

12.4. Declaratie

In het algemeen: aangeven dat er 'iets' is, zonder aan te geven wat dit 'iets' is of hoe dit 'iets' werkt. Wordt wel meteen aangegeven wat 'iets' is of hoe 'iets' werkt, dan is dit een definitie. Zie functie declaratie, klasse declaratie, methode declaratie, variabele declaratie.

12.5. Definitie

In het algemeen: aangeven dat er 'iets' is en meteen aan te geven wat dit 'iets' is of hoe dit 'iets' werkt. Wordt niet

meteen aangegeven wat 'iets' is of hoe 'iets' werkt, dan is dit een declaratie. Zie functie definitie, klasse definitie, methode definitie, variabele definitie.

12.6. Event

Een Kylix-specifieke methode die onder run-time bij bepaalde gebeurtenissen wordt aangeroepen en design-time kan worden toegewezen.

12.7. Functie

Een stuk code dat iets kan doen, met eventueel argumenten en een return waarde. Een functie die niets terug geeft (oftewel **return** type is **void**) wordt soms ook een procedure genoemd. Een functie binnen een klasse wordt een methode genoemd.

```
/* return type */ /* identifier */ ( /* argumenten */ );
```

```
double DeelDoor(const double teller, const double noemer);
```

12.8. Functie declaratie

Het aangeven van de identifier, return type en argumenten van een functie, zonder aan te geven hoe deze functie dit doet. Wordt wel aangegeven hoe de functie werkt, dan wordt dit een functie definitie genoemd. Worden de namen van de argumenten niet genoemd, dan heet het een functie prototype.

```
/* return type */ /* identifier */ ( /* argumenten */ );
```

```
double DeelDoor(const double teller, const double noemer);
```

12.9. Functie definitie

Het aangeven van de identifier, return type, argumenten van een functie en hoe deze functie dit doet. Wordt niet aangegeven hoe de functie werkt, dan wordt dit een functie declaratie genoemd.

```
/* return type */ /* identifier */ ( /* argumenten */ )  
{  
    /* code */  
}
```

```
int DeelDoor(const double teller, const double noemer)  
{
```

```
return teller / noemer;
}
```

12.10. Functie prototype

Een functie declaratie waarin de namen van de argumenten niet genoemd worden, slechts hun data types.

```
double DeelDoor(const double, const double);
```

12.11. Generiek programmeren

Zo algemeen mogelijk programmeren, door middel van templates.

```
template <class T>
void Verwissel(T& a, T& b)
{
    const T temp = a;
    a = b;
    b = temp;
}
```

12.12. Header file

Bestand met '.h' als extensie, die vooral decaraties bevat. Het implementatie (.cpp) bestand bevat vooral definities. Een header file en een gelijknamig implementatie bestand wordt soms een Unit genoemd.

12.13. Identifier

De naam van een variabele of funtie.

```
int x; // x is de identifier voor een int

void DoeIets(); // DoeIets is de identifier voor een void
               //functie, die geen argumenten nodig heeft.
```

12.14. Implementatie bestand

Bestand met '.cpp' als extensie, die vooral definities bevat. Een header file (.h) bevat vooral declaraties. Een header file en een gelijknamig implementatie bestand wordt soms een Unit genoemd.

12.15. Initialiseren

Een variabele een eerste waarde geven

```
int x; //Declareer de integer x
x = 0; //Initialiseer de integer x;
```

Een variabele gelijk met de declaratie initialiseren wordt een variabele definitie genoemd.

12.16. Klasse

Een 'slim' data type dat zorgt dat het altijd in een juiste staat verkeerd. Een klasse bestaat uit lidvariabele en methoden.

12.17. Klasse declaratie

Het aangeven van de identifier en het geven van de methode declaraties en lidvariable declaraties waaruit de klasse bestaat. Zouden de methoden meteen gedefinieerd zijn, dan zou dit een klasse definitie zijn, maak vaak staan methode definities apart. Klasse declaraties staan vaak in header files (.h bestanden).

```
/* classe type */ /* identifier */
{
    /* lidfuncties */;
    /* lidvariabelen */;
};
```

```
struct MijnKlasse //Klasse declaratie
{
    int Iets(const int x) const; //Methode declaratie
    int mX; //Lidvariabele declaratie
};
```

12.18. Klasse definitie

Het aangeven van de identifier, lidfuncties en lidvariabelen met hun volledige werking. Zouden de lidfuncties niet uitgewerkt zijn, dan zou dit een klasse declaratie heten. Klasse definities staan vaak in header files (.h bestanden).

```
/* classe type */ /* identifier */
{
    /* lidfuncties */;
    /* lidvariabelen */;
};
```

```
struct MijnKlasseDefinitie
{
```



```

MijnKlasseDefinitie()
: mX(1) //Variabele definitie
{

}
int MijnMethodeDefinitie(const int x);
{
    mX *= x;
    return mX;
}
int mX; //Variabele declaratie
};

```

12.19. Lidfunctie

Synoniem voor methode. Zie methode.

12.20. Methode

Functie binnen een klasse, die toegang heeft tot de (ook **private**) lidvariabelen van dezelfde klasse.

12.21. Methode declaratie

In een klasse declaratie, het aangeven van een methode. Zou de werking van de methode meteen worden geschreven, dan zou dit een methode definitie zijn. Methode definities staan vaak buiten de class declaratie.

```

struct MijnKlasse //Klasse declaratie
{
    int Iets(const int x) const; //Methode declaratie
};

```

12.22. Methode definitie

Binnen een klasse definitie, de werking van een methode beschrijven. Omdat de declaratie van een klasse in een header (.h) file staan, staat de volledige klasse definitie ook vaak in een header (.h) file.

```

struct MijnKlasse //Klasse definitie
{
    int Iets(const int x) const; //Methode definitie
    {
        return x * x * x;
    }
};

```

Buiten een klasse declaratie, de werking van een methode beschrijven. Methode definities staan vaak in een implementatie (.cpp) bestand.

```
struct MijnKlasse //Klasse declaratie
{
    int Iets(const int x) const; //Methode declaratie
};

int MijnKlasse::Iets(const int x) const //Methode definitie
{
    return x * x * x;
}
```

12.23. Procedure

Functie zonder **return** waarde, oftewel met **return** type **void**.

```
void /* identifier */ ( /* argumenten */ );
```

```
void ZegHallo(const String naam);
```

12.24. Referentie

Een alias van een variable, in plaats van een kopie. Een referentie is te herkennen aan de ampersand.

```
void Verwissel(int& a, int& b);
```

12.25. STL

'Standard Template Library'. De C++ platform-onafhankelijke standaard bibliotheek.

12.26. Unit

Een header file en een gelijknamig implementatie bestand.

12.27. Variabele

Een waarde die in het geheugen tijdelijk moet worden opgeslagen. Een variabele heeft altijd een data type en een identifier.

```
int x; //Declareer een variabele van data type int
      //met de identifier x
```

12.28. Variabele declaratie

Het aangeven van het data type en identifier. Het tegelijkertijd toekennen van een waarde aan deze variabele wordt een variabele definitie genoemd.

```
/* data type */ /* identifier */;
```

```
int x;
```

12.29. Variabele definitie

Het aangeven van data type, identifier en een initiele waarde. Zou geen initiele waarde worden toegekend, dan wordt dit een variabele declaratie genoemd.

```
/* data type */ /* identifier */ = /* initiele waarde */;
```

```
int x = 12;
```