Trein

Hopelijk dit jaar, maar waarschijnlijk volgend jaar, komt de ISO met een nieuwe specificatie voor C++. Behalve verbeteringen op de taal zelf en uitbreidingen van de standaard library, worden ook threads en lambda-functies in C++ opgenomen. Daarmee sluit de nieuwe specificatie aan bij de parallellisering van software en meta-programming.

Nieuwe C++-standaard zet in op parallellisering

Werken volgens de Unix-filosofie

Op dit moment werkt de ISO (International Organization for Standardization) aan de volgende versie van C++. In navolging van het origineel C++98 en de update C++03 heeft de nieuwe standaard de voorlopige naam C++0x meegekregen. Daarmee wordt gesuggereerd dat de C++ standaardisatie-commissie zijn werk nog dit jaar zal kunnen afronden. Of dat echt zal lukken wordt echter betwijfeld. Een veelgehoorde grap is om die 0x dan maar hexadecimaal op te vatten. Dat betekent dat de volgende versie van C++ wel eens als C++0A bekend zou kunnen worden.

Features

De features van de volgende versie zijn inmiddels wel bekend. Belangrijkste echte nieuwigheid zijn de zogenaamde concepts. Deze beschrijven de voorwaarden waaraan een class moet voldoen om in een bepaalde template te passen. Maar daarnaast zijn er nog een hele zwik kleinere verbeteringen, aanpassingen en optimalisaties op de taal zelf en diverse uitbreidingen (reguliere expressies, smart pointers en hash tabellen) op de standaard library.

Wat betreft de bibliotheken zullen threads en lambda-functies standaard onderdeel van de taal worden. Dat heeft alles te maken met de verschuiving van hogere kloksnelheden naar meerdere processorkernen. Waar programmeurs hogere prestaties voorheen cadeau kregen, zullen zij de beschikbare parallelle performance straks zelf moeten uitnutten. We schreven daarover al eerder in SRM.

Clean en compatibel

Belangrijkste doel bij de ontwikkeling van de volgende C++-specificatie is natuurlijk om de nieuwe versie downward compatibel te maken met de eerdere versies van deze programmeertaal. Omdat C++ een Object Oriented (OO) uitbreiding is van C, houdt dat meteen ook in dat de compatibiliteit met C bewaard blijft.

Tegelijkertijd moet het aantal uitbreidingen beperkt blijven. Niet-generieke functies worden opgenomen in de standaard library. En uitbreidingen voor specifieke toepassingen hebben überhaupt niets in de standaard onderdelen van de taal te zoeken maar dienen in aparte bibliotheken te worden ondergebracht.

Op deze manier blijft de belangrijkste eigenschap van de C/C++-taal behouden, namelijk dat de primitieven zo generiek en eenvoudig mogelijk moeten zijn. Complexer zaken dienen zo veel mogelijk te worden opgebouwd uit simpeler basisbouwstenen.

Unix-filosofie

Met deze ambitie doet de ISO-commissie recht aan de achtergrond en de historie van de programmeertaal. C zoals die destijds door Dennis Ritchie is gedefinieerd voor gebruik bij het Unix opererating system, is uiterst strak en sober van ontwerp. Dat maakt dat de taal voor nieuwkomers snel te leren is.

Deze manier van werken sluit bovendien naadloos aan bij de Unix-filosofie. Daar worden bij voorkeur hele kleine, uiterst flexibele tooltjes gedefinieerd, die vervolgens op eenvoudige wijze aan elkaar geknoopt kunnen worden tot grotere, krachtige commando's. Niet toevallig is Ritchie behalve bedenker van de taal C ook een van de ontwerpers van Unix.

Aad Offerman
is afgestudeerd in de Technische Informatica aan de TU Delft, waar hij zich heeft gespecialiseerd in Computer Architectuur en Digitale Techniek. Hij is actief als technologie-auteur en New Media specialist in de Nederlandse ICT markt. Hij is te bereiken op adrian@offerman.net

Threads

Na processen bieden threads het volgende niveau van parallellisme. Waar processen hun eigen (virtuele) adresruimte hebben, wordt die echter gedeeld door de verschillende threads onder een proces. Daarmee is de communicatie tussen de threads een fluitje van een cent; ze werken immers allemaal op dezelfde data. Op proces-niveau moet je je behelpen met de relatief langzame IPC-faciliteiten (Inter-Process Communication) van het operating system. Denk dan bijvoorbeeld aan files, sockets en pipes. Threads hebben vanzelfsprekend wel hun eigen register file en stack.

Threads zijn ook beschikbaar op single-processor systemen. Daar zorgt de scheduler van het besturingssysteem dat de verschillende threads (net als de processen) om beurten worden uitgevoerd (multiplexing). Threads (en processen) komen echter pas goed tot hun recht op een SMP- (Symmetrical Multi-Processor) of multi-core-systeem. Daar kunnen de verschillende threads immers daadwerkelijk parallel naast elkaar worden geëxecuteerd. Multi-core processoren hebben daarbij een belangrijk voordeel. De communicatie tussen threads op verschillende kernen hoeft niet over de externe bus heen. Dat betekent dat cache-coherency protocollen tot de hogere niveau's in de cache beperkt kunnen blijven.

Synchronisatie

In C en C++ worden threads meestal geïmplementeerd middels de Pthreads library. Deze API (Application Programmer Interface) is ooit ontwikkeld als onderdeel van de POSIX-standaard (versie 1003.1c), maar is ook beschikbaar op andere platforms dan Unix. Na de pthread.h header kunnen direct nieuwe threads worden gedefinieerd en opgestart (create). Verderop in de code kan indien nodig weer worden gewacht (join) totdat een bepaalde thread klaar is met zijn taak (exit).
De moeilijkheid met threads zit 'm niet zo zeer in het management ervan maar vooral in de onderlinge synchronisatie. Vanwege de gedeelde resources moet die op code-niveau plaatsvinden. Het eenvoudigste voorbeeld is de verhoging van een variabele. Een thread moet zeker weten dat de variabele die hij heeft ingelezen niet veranderd wordt voordat de aangepaste waarde weer is weggeschreven. Het zou immers heel goed kunnen dat een andere thread tegelijkertijd ook bezig is om dezelfde variabele te verhogen. Men spreekt bij dit soort interactie-problemen van race-conditions: verschillende threads willen zo snel mogelijk hun nieuwe waarde wegschrijven, voordat een ander de eerder ingelezen waarde alweer heeft aangepast.

Atomische operaties

De oplossing ligt in het gebruik van atomische operaties. Dat wil zeggen, stukjes code die gegarandeerd niet door andere threads onderbroken kunnen worden. Kritieke secties, waarin toegang wordt gezocht tot gedeelde resources, worden beveiligd met een mutex. De naam ervan, kort voor mutual exclusion, zegt al wat deze flag doet: deze zorgt dat maar één thread tegelijkertijd met de beschermde resources aan de slag mag. Elke kritieke sectie begint dus met een verzoek om een mutex te zetten. Lukt dat, dat wil zeggen dat de bijbehorende resource op dit moment niet is gereserveerd voor een andere thread, dan wordt dat verzoek toegekend. Lukt dat niet, dan wordt een thread geblokkeerd totdat de mutex weer wordt vrijgegeven, of moet de thread verder met een andere taak.
In de praktijk betekent dat dus dat voor elke gedeelde resource een mutex wordt gedefinieerd. Elke thread begint de kritieke sectie met een verzoek om de bijbehorende mutex te zetten (lock, of de niet-blokkerende trylock). Bij het verlaten van de kritieke sectie wordt de mutex weer vrijgegeven (unlock), waarna een van de wachtende threads de resource krijgt toegekend. Omdat kritieke secties wachtende threads tot gevolg hebben en daarmee de parallelle uitvoering beperken, is het natuurlijk zaak de kritieke secties zo klein mogelijk te houden. Dat heeft niet alleen consequenties voor de programmacode zelf maar ook voor de manier waarop de datastructuren en andere resources worden georganiseerd.

Vlaggetjes

Tenslotte is het nog goed om te weten dat mutexen niet meer zijn dan vlaggetjes. Voor welke gedeelde resource ze gebruikt worden, is niet hard vastgelegd bij de definitie ervan. Die relatie is gewoon een conventie in het betreffende proces, waar alle threads zich aan hebben te houden. Functies die al hun mogelijk gedeelde resources op deze manier beschermen, worden thread-safe genoemd.
Behalve synchronisatie met behulp van joins en mutexen, bestaat ook nog de mogelijkheid om condition variabelen te gebruiken voor de onderlinge afstemming. Threads kunnen wachten totdat een bepaalde variabele true wordt voordat ze verder gaan (wait, timedwait). De betreffende variabele wordt gezet door een andere thread zodra die een bepaald punt heeft bereikt (signal, broadcast). Omdat condition variabelen per definitie worden gedeeld, worden ook hier weer mutexen gebruikt.

Deadlock

Wat het gebruik van al deze verschillende synchronisatie-mechanismen lastig maakt, is dat je het risico loopt dat bepaalde threads voorgoed vastlopen. Het kan bijvoorbeeld zo zijn dat thread staat te wachten totdat een bepaalde mutex vrij komt, terwijl er geen andere thread (meer) is om dat te doen. Men spreekt dan van een deadlock. Vaak zijn die echter ingewikkelder dan dit voorbeeld. Threads kunnen allemaal op elkaar staan te wachten, terwijl niemand meer verder kan zonder dat een ander eerst zijn reserveringen opgeeft (een circulaire afhankelijkheid, vergelijk het probleem van de Dining Philosophers). Of threads draaien nog wel, maar komen niet verder (een lifelock).
Hetzelfde kan gebeuren niet op logisch maar op executie-niveau: een semi-deadlock wordt veroorzaakt door een thread die zijn taak probeert af te krijgen maar daarbij wordt gehinderd door andere threads die actief almaar opnieuw checken (busy waiting) of ze verder mogen (spinlock).
Het resultaat van al deze varianten is steeds hetzelfde: resource starvation. Voor de programmeur betekent dit dat hij bij het ontwerp van zijn programma goed moet uitzoeken of dit soort situaties niet kunnen ontstaan.

Concurrentie

Tenslotte is de strakke definitie van C/C++ ook de belangrijkste onderscheidende waarde ten opzichte van de grote concurrent Java. Deze laatste heeft een veel bredere basis dan C++. Wil je lekker met Java uit de voeten kunnen, dan heb je al gauw 'een meter O'Reilly' op je boekenplank staan. Hoewel je als programmeur in Java standaard meer faciliteiten tot je beschikking hebt, is de leercurve daardoor langer. Bovendien is het moeilijker je Java-kennis te onderhouden als je niet elke dag met die programmeertaal bezig bent. Dat maakt C/C++ beter geschikt voor 'parttime programmeurs'.
Met deze tweedeling, die niet alleen geldt voor de programmeertaal maar ook voor de ontwik-

Lambda-functies

Lambda-functies zijn van origine een mathematische modellering uit de theoretische wiskunde. Hieruit zijn echter de functionele programmeertalen ontwikkeld. Die blijken eigenschappen te hebben die ze makkelijk te parallelliseren maakt. Bovendien bieden ze mogelijkheden tot meta-programming. Daarbij worden nieuwe functies gegenereerd of bestaande functies veranderd. Drijf je dat te ver door, dan neem je echter afscheid van de Von Neumann computer-architectuur waarbij instructies en data strikt gescheiden zijn.

Programma's geschreven in een functionele taal in zijn pure vorm, bestaan uitsluitend uit functies. Belangrijke eigenschap is dat ze stateless zijn. Dat in tegenstelling tot het klassieke, imperatieve model, waarbij programmacode tijdens zijn berekening de status en datastructuren juist continu aanpast. Omdat ze niet afhankelijk zijn van de omgeving, geven stateless lambda-functies dus ook altijd hetzelfde resultaat voor dezelfde parameters, een eigenschap die referential transparency wordt genoemd.
Daarnaast is het in deze wereld heel gebruikelijk om ook functies zelf weer als parameter mee te geven (vergelijk functie pointers). Op die manier kun je bijvoorbeeld met de ene functie samengestelde datastructuren doorlopen terwijl je steeds een andere operatie uitvoert met een andere functie (meegegeven als parameter). Het veranderen van functies (door zogenaamde hogere-orde functies) wordt in moderne programmeermodellen als not-done beschouwd en wordt ook niet ondersteund door moderne architecturen.

Recursie

De belangrijkste functionele programmeertalen zijn Erlang, Haskell en Scheme (een dialect van LISP). Wie bijvoorbeeld bekend is met Scheme, herinnert zich vast nog de dubbele drempel die bij het leren ervan genomen moest worden. Behalve de conceptuele verschillen met imperatieve talen waren er ook de vele haakjes die paarsgewijs precies moesten kloppen maar tegelijkertijd je code onleesbaar maakten.
Waar functionele talen erg goed in zijn is de recursieve specificatie van algoritmen. Sterker nog, andere flow controls bestaan in pure lambda-functies niet eens. Dat maakt functionele talen in het algemeen wel minder efficiënt dan imperatieve talen. Al die recursieve functie-aanroepen doen immers een zwaar beroep op je stack. Een iteratieve loop heeft al die overhead niet. Uitzondering zijn de zogenaamde tail recursieve functies. Daarbij eindigt een recursieve functie-definitie met een aanroep naar zichzelf, zonder dat er nog verdere berekeningen op de return value hoeven worden uitgevoerd. Omdat er niets op de stack bewaard hoeft te worden, kan een compiler die functies zo optimaliseren dat bij de volgende call geen nieuwe stack entry hoeft te worden gemaakt.

Omdat lambda-functies niet afhankelijk zijn van de globale executie-status, kunnen ze direct worden geëvalueerd zodra hun parameters bekend zijn. Bovendien kan de evaluatie van een lambda-functie zonder side-effects compleet worden overgeslagen als dat vanuit logisch perspectief niet nodig is (denk aan logische expressies die vaak maar gedeeltelijk hoeven te worden geëvalueerd om de uitkomst al te weten). Dat maakt lambda-functies uitermate geschikt voor geautomatiseerde, asynchrone en parallelle verwerking.

C/C++ is beter geschikt voor 'parttime' programmeurs

kelomgeving en de cultuur, hebben we gelijk het grootste deel van de markt beschreven. Volgens het Global Developer Population and Demographics Report 2008 van Evans Data gebruikt 47 procent van alle ontwikkelaars op dit moment Java. Nummer twee is C/C++ met 41 procent.
De aanpassingen in C++0x behelzen zowel de taal zelf als de standaard library. Voor wat betreft dat eerste gaat het vooral om een hele trits kleinere verbeteringen. Denk dan aan aanpassingen die de snelheid van compileren en executie ten goede komen enerzijds, en aan uitbreidingen die de programmeurs meer functionaliteit bieden anderzijds.
Eenvoudig voorbeeld is het label 'constexpr', dat aan functies die altijd dezelfde waarde teruggeven (een constante expressie) kan worden meegegeven. Zo weet de compiler dat de desbetreffende functie al tijdens het compileren volledig kan worden geëvalueerd. Op die manier worden de prestaties tijdens de executie verbeterd.

Initializer lists

Meer op de programmeur gericht is een breder gebruik van initializer lists. Deze worden ingezet om structs en arrays in één keer te vullen. Dat kan door een complete lijst met waarden aan een variabele toe te kennen. Voor complexer (lees: samengestelde) datastructuren als arrays van structs of structs van structs worden dan geneste lijsten gebruikt.
Diezelfde initializer lists zijn straks ook beschikbaar voor de constructors van objecten. Bij de instantiation van een nieuw object kan dan een initializer list als parameter worden meegegeven. Nog een voorbeeld is de ondersteuning van range-based for-loops. Daarbij worden de waarden die de loop-variabele doorloopt niet bepaald door een

rekenkundige expressie maar door de inhoud van een array of een andere range. Deze mogelijkheid is afkomstig uit de boost bibliotheek en vergelijkbaar met het foreach statement uit andere talen.
Ook nieuw zijn de concepts. Daarmee wordt een verzameling voorwaarden beschreven waaraan een class moet voldoen om in een bepaalde template te passen. De class moet immers alle operaties in de template definiëren om de volledige template interface te kunnen leveren.
Deze uitbreiding is vooral bedoeld om het leven van de debuggende programmeur te vergemakkelijken. Waar foutmeldingen bij een mismatch nu een hele brei moeilijk leesbare tekst ophoesten, kan straks gewoon een melding worden gegeven dat niet wordt voldaan aan een bepaald concept.
Definities voor concepten kunnen bestaan uit operators, type conversies of combinaties daarvan. Maar ook het nesten of juist afleiden (vergelijk inheritance) van concepten is mogelijk.

Klaar?

De grootste uitbreiding op de C++-specificatie zelf is de incorporatie van lambda-functies en threads. Deze worden in C++0x onderdeel van respectievelijk de taal zelf en de standaard bibliotheek. Zoals gezegd zijn ze vooral bedoeld om programmeurs te helpen de performance van multi- en many-core processoren uit te nutten.
Belangrijke vraag bij de verbeteringen in de C++0x-specificatie is natuurlijk of ze genoeg zijn om de programmeertaal de komende jaren door te helpen. Java heeft sinds zijn introductie in 1995 enorm snel aan marktaandeel gewonnen. Ondanks zijn fundamenteel andere executie-model (interpretatie van intermediate bytecode in plaats van compilatie en native executie), lijkt Java met zijn zowel impliciete als expliciete ondersteuning van threads beter voorbereid op het many-core tijdperk. «

Patches Patches Patches Patches Patches Patches Patches P

Artikelen over onderwerpen als software-ontwikkeling, Java, UML, eXteme Programming en nog veel meer vindt u in het Online Archief van Array Publications. Vaktijdschriften als Software Release, Java Magazine, Database Magazine en ons Oracle vakblad Optimize hebben hun artikelenarchief online gezet. Dankzij de heldere zoekstructuur vindt u snel wat u zoekt op www.release.nl.

Autodesk neemt BIMWorld over

Autodesk, leverancier van 2D- en 3D-ontwerpsoftware, heeft alle activa overgenomen van BIMWorld, een private onderneming die is gespecialiseerd in de productie en de verspreiding van branded BIM content ten behoeve van fabrikanten van bouwproducten. Met de overname zal Autodesk de content van Autodesk Seek kunnen uitbreiden. Autodesk Seek is de online database waarin Autodesk ontwerpinformatie over bouwproducten beschikbaar stelt. Na de overname zal BIMWorld in deze database geïntegreerd worden. Met de combinatie van BIMWorld en Autodesk Seek krijgen architecten en ontwerpers toegang tot een online tool waarmee ze gedetailleerde informatie over en beeldmateriaal van bouwproducten rechtstreeks vanuit hun ontwerpapplicaties kunnen opzoeken en downloaden. De webservice Autodesk Seek is beschikbaar voor de Amerikaanse 2009-versies van de Revit-gebaseerde softwareapplicaties voor BIM en voor AutoCAD, AutoCAD Architecture en AutoCAD MEP. Autodesk Seek is toegankelijk via http://seek.autodesk.com.

Atos Origin Technical Automation samen met TASK24

De overname van Atos Origin Technical Automation (TA) door Gilde is afgerond. TA en TASK24 zijn vanaf 1 januari onder het label TASK24 een nieuwe Nederlandse speler op het gebied van technische automatisering.
De nieuwe combinatie beschikt met 550 medewerkers over een bredere technologische expertise en stelt TASK24 in staat invulling te geven aan grote opdrachten en uitdagingen. Door medewerkers en klanten van TA en TASK24 is positief gereageerd op de krachtenbundeling, aldus het bedrijf. De kerndiensten van TASK24 bestaan uit consultancy, projecten, development en engineering op basis van diverse business modellen.

Snel ROI bij migratie van mainframe naar open

De ROI van het migreren van mainframe naar open omgeving wordt bij circa zestig procent van de organisaties binnen een jaar gerealiseerd. Dat blijkt uit een enquête van het Nederlandse Asysco, specialist in mainframe-migraties. De meeste ondervraagde bedrijven geven aan dat de kosten voor het onderhouden van een mainframe dusdanig hoog zijn, dat de investering in een migratie snel is terugverdiend.
Asysco heeft onderzoek uitgevoerd onder IT-managers in Europa. Bijna negentig procent van de ondervraagden beschikte over het Unisys A-Series-platform. De resterende tien procent maakte gebruik van de Unisys OS 2200-besturingsomgeving. Op de vraag naar de voornaamste reden voor het migreren van het mainframe geeft negentig procent van de ondervraagden aan de kosten van het onderhoud van het mainframe. Dertig procent noemt dat zij eindgebruikers een betere interface wilden bieden en een betere aansluiting wilden realiseren bij webapplicaties of internet. Andere genoemde redenen zijn de kosten van het beheer (twintig procent), betere performance voor de toekomst (negentien procent), een betere aansluiting bij andere platformen (zeventien procent), onafhankelijkheid van hardware (vijftien procent) en het beschikken over onvoldoende mainframe-kennis (tien procent).