

Hvorfor nok en bog om C++?

1

Ved sidste optælling var der mindst 2.768.942 bøger om C++ på markedet, plus massevis af kurser, multimediebase-rede selvstudier, tidsskrifter og C++-happenings. Når man undersøger alle de C++-bøger, som findes hos boghandleren, er det som at kigge på kalendere, idet alle C++-bøgerne basalt set har samme indhold og faktisk kun adskiller sig ved deres vægt og farven på omslaget. Ifølge min optælling er 2.768.940 af bøgerne for begyndere, knytter sig til en bestemt oversætter eller omhandler udelukkende C++-syntaksen. For alle, som allerede kender C++-sproget, og som ønsker at bevæge sig op på det næste niveau, er det en frustrerende oplevelse. Man er nødt til at læse et kapitel her og der i forskellige bøger for at finde frem til noget, som man ikke allerede ved i forvejen. For dem, som allerede befinder sig på et højt C++-niveau, er det basalt set blot spild af tid.

Denne bog er anderledes. For det første forudsætter jeg, at du allerede kender C++. Du har formentlig allerede mere end 2 års programmeringserfaring i C++, ligesom du godt ved, at en klasse ikke er en samling børn, som går i folke-

skolen. Du har formentlig en kopi af Stroustrups "Annotated C++ Reference Manual" og citerer fra den i flæng, uden at gøre dig den ulejlighed at forklare detaljerne for den undrende pøbel.

Hvis du kan genkende sig selv – velkommen, kom bare indenfor. En alternativ titel til denne bog kunne være "Guruens vej til C++". Denne bog adskiller sig nemlig fra alle basale C++-bøger ved at hoppe direkte til de avancerede dele af C++.

Jeg kender ikke andre sprog, der som C++ udmærker sig ved at være mere end blot et programmeringssprog, hvis man ønsker at benytte de mere avancerede sider af det. Der er mere tale om en subkultur, som indeholder sine egne tips, tricks og arkitekturmønstre, som ikke umiddelbart kan læses ud af selve sproget. Dette sprog-i-sproget behandles sjældent i bøger eller artikler. De fleste C++-programmører finder selv ud af disse tricks og tror, at de har opfundet noget helt specielt. Senere finder man ud af, at andre også har kæmpet sig frem til præcis de samme løsninger. Andre

er så heldige, at de bliver oplært af en C++-guru, men der er desværre ikke guruer nok i omløb til, at vi alle kan blive oplært af en. Denne bog er et forsøg på at tilbyde en ny partner, nemlig en selvstudieguide til større forståelse af C++. Bogen henvender sig også til folk, som allerede befinder sig på et højt C++-niveau, men som ønsker at få tingene sat i perspektiv og lære mere om sprogets finurligheder. Endelig kan bogen også bruges af folk, som blot gerne vil have lidt C++-hjernegymnastik.

Det allerhelligste

C++ er et sprog, som man lærer i flere faser. Først når man er nået til sidste fase, begynder det endelig at give mening. Man begynder at kunne se sammenhængen i de løsevne tips og tricks, der tilsammen går op i en højere helhed og dermed åbenbarer C++'s allerhelligste sider. Jeg opfatter det at lære C++ som at køre i elevator – ding! Så er du nået til anden sal. C++ er et mere fornuftigt C. Det indeholder et stærkt typecheck, så længe man ikke opfører sig for tabeligt, og så er der de smarte //-kommentarer. Alle de C-programmører, som ikke er interesserede i at få en ledelsesmæssig karriere, har ledt efter en anden karrieremulighed, og den har Bjarne Stroustrup – Gud bevare ham – opfundet.

Ding! Tredje sal. C++ er et godt, men ikke fantastisk objektorienteret programmeringssprog. Det er ikke Smalltalk, men hvad forventer du af et programmeringssprog, som er så drønende hurtigt? C++ er 90'ernes COBOL – politisk korrekt og vellidt af dine chefer. Faktisk er det tit muligt at få et

ekstra stort budget, hvis man nævner C++ tit nok i et projektforslag. Og det er måske også meget godt, eftersom der ikke rigtig er nogen, som ved, hvordan man styrer et C++-projekt, samtidigt med at der ikke rigtig er nogle værktøjer, som for alvor kan hjælpe en.

Ding! Øverste etage, alle ud. Hov, hvor blev de alle sammen af? Der er godt nok koldt heroppe. C++ er faktisk yacc++. Det er ikke så meget et programmeringssprog, men mere en måde at skabe sit eget sprog på. Det er elegant, ikke fordi det er simpelt, men pga. sit potentiale. Bag hvert eneste krogede designproblem ligger C++ på lur med en smart løsning, som får problemet til at forsvinde som en isterning på en varm sommerdag. C++ løser problemet lige så elegant som *rigtige* programmeringssprog som f.eks. Smalltalk eller LISP. En væsentlig forskel er dog, at løsningen ikke får sort røg til at stige op fra CPU'en samtidig med, at aktiekurserne på de firmaer, som producerer hukommelses-chips, stiger med lynets hast. C++ er ikke et programmeringssprog – det er en oplevelse, et bevidsthedsudvidende stof.

Der var jo ordet igen – *elegant*. Der hviler en helt specielt ånd over at designe et program i C++. Man er faktisk nødt til at lade være med at koncentrere sig om at lave elegante programmer for derved at opnå ægte elegance. C++ er mest af alt næste generation af C. Det oversætter hurtigt og kører stærkt. Det har en meget brugbar blokorienteret grammatik samt praktiske forkortelser for det, man bruger mest, som f.eks. `i++`. Der er navneord, udsagnsord, tillægsord og en masse slang som f.eks.:

```
Cout << 17 << endl << flush;
```

C++-programmører er tit og ofte blevet sat i skammekrogen af sprogpuritanerne. Puritanerne tror, at et sprog som udelukkende består af atomer og parenteser, er det vi alle længes efter. De syntaksterrorister, der opfinder programmeringssprog, hvor det ikke er muligt at se forskel på en variabel, et funktionskald eller en makro, tror, at de har skabt det ottende vidunder. Men helt ærligt, hvis de er så smarte, hvorfor er de så ikke også blevet utroligt rige? I det virkelige liv vil folk kun betale for de programmeringssprog, hvor de forskellige sprogelementer også rent fysisk ser forskellige ud. De "simple og konsistente" programmeringssprog har aldrig fået den store fanskare uden for universiteterne, mens de blokstrukturerede programmeringssprog er blevet foretrukket af de brede masser. Og hvordan kan det i virkeligheden overraske nogen? Programmeringssprog skal læres og læses med de samme hjerneceller, som vi benytter til at lære og læse de naturlige sprog. Kan du hurtigt lige nævne et naturligt sprog, som ikke indeholder navne- og udsagnsord, men som derimod benytter parenteser i store mængder? Jeg tænkte det nok! Lingvistikken fortæller os helt klart, at indlæringstiden falder, og læsevenligheden forøges, hvis sproget indeholder alle de påståede "grimme" faciliteter, fordi i++ rent faktisk er hurtigere at læse end $i := i + 1$, ligesom de fleste mennesker har lettere ved at læse $x = 17 + 29$ end f.eks. (setq x (+ 17 29)). Det har ikke noget at gøre med selve designet af programmeringssproget, men en hel masse at gøre med den måde vi er designet på. C++ er ikke specielt pænt, simpelthen fordi

vi ikke er specielt pænt designet. Når først du har lært de smarte genveje og er holdt op med at bekymre dig om matematisk konsistens, vil du hurtigt komme til at elske alle de "grimme" faciliteter, og du er på den rette vej direkte mod det elegante C++. Det som gør C++ nemt at læse er, at det stolt og oprigtigt præsenterer sig for læseren, så det er nemt at forstå, hvilke koncepter der gemmer sig på de enkelte C++-sider.

Ligesom i LISP, Smalltalk og andre dynamiske programmeringssprog (i modsætning til C) stiller C++ de nødvendige håndtag til rådighed, så det er muligt at manipulere direkte med datamaskinen. Du kan lave dine egne datatyper og få datamaskinen til at adoptere dem, som om de var dens egne. Du kan kontrollere, hvordan funktionerne bliver kaldt, hvordan adgangsrettighederne til data er, hvordan lageret bliver allokeret og deallokeret, samt hvordan og specielt hvornår ting bliver initialiseret og ryddet op, alt sammen uden at ofre effektivitet og typecheck. Modsat de andre programmeringssprog vil C++ blot gå ned, hvis man bruger disse faciliteter på en forkert måde. Selv om det ikke direkte går ned, vil dine kollegaer garanteret give dig det røde kort, hvis du ikke er i stand til at benytte det rette designkoncept i forbindelse med et givent problem. Daidalus og hans søn Ikaros flygtede fra et fængsel på Kreta ved at lave vinger af fjer og voks. Daidalus, som var hovedarkitekten og opfinderen, svævede til fjerne kyster. Hans uforsigtige søn fløj for tæt på solen og styrtede i havet. Hmm, nu når jeg tænker nærmere over det, er det måske ikke så god en analogi. Daidalus har også designet

en labyrint, som var så kompliceret, at folk enten døde af sult, mens de ledte efter udgangen, eller blev spist af Minotaurus. Måske er det bedre med en mere jordnær analogi. Hver gang man bruger disse lavniveaufaciliteter, er det ligesom at sige: "Stol på mig, jeg ved, hvad jeg gør". Datamaskinen skal så blot lukke øjnene og gøre, som det bliver sagt!

C++ er fascinerende pga. dets indbyggede modsætninger. Det er kraftfuldt, fordi det indeholder faciliteter, som dog nemt kan misbruges. Det har et programmeringsmiljø, som kan udbygges i det uendelige, uden at programmerne bliver langsomme eller for store. Det kan være elegant i hænderne på én programmør, mens det kan være katastrofalt i hænderne på en anden. Det er simpelt og komplekst på en gang. Selv efter at man har brugt det i mange år, kan man stadig ikke bestemme sig til, om man skal beundre sproget eller flygte så langt væk fra det som muligt. Den ægte ekspert kender dog sprogets underliggende koncepter, hvilket får vægten til at tippe til fordel for C++. Disse koncepter kan man ikke lære på et øjeblik. De skal indlæres ved, at man anvender sproget i mange forskellige sammenhænge og over en længere periode. Visse arkitektoniske paradigmer løses bedst med bestemte dele af sproget, men hvis man kommer til at vælge de forkerte, skaber man flere problemer end løsninger. Hvis man finder den rette kombination, bliver resultatet fint og elegant.

Tre superbe ideer i C++

Der kan skrives så meget om avanceret C++, at det er svært at vide, hvor man skal begynde. Har du nogen sinde set et af de billeder, som i starten blot ligner et tilfældigt mønster, men langsomt begynder at ligne et eller andet bestemt, efter at man har kigget på det et stykke tid. Pludselig begynder prikkerne og stregerne at give mening, fordi man forstår det underliggende tema. Netop dette forhold gør, at det kan være ret frustrerende at lære de arkitekturer og koncepter, som indgår i C++. Der findes en stor kasse fuld af tilfældige tricks og ikke rigtig nogen regler, som viser, hvordan og specielt, hvornår man skal benytte dem. Denne bog vil kaste lys over disse tricks. Der er mange måder at organisere avanceret C++-tanker på, men denne bog samler dem i nogle få simple dele, nemlig:

- Viderestilling
- Sammenhængende klassehierarkier
- Lagerstyring

Hvert enkelt område understøttes af noget specifikt C++-syntaks og -funktionalitet, og sammenlagt kan de løse en overraskende bred vifte af problemer. Der er en lang række andre designprincipper og tricks, som man kunne medtage i denne bog, men de tre områder tilsammen giver en god sammenhængende struktur i bogen, samtidig med at de dækker et meget stort felt.

I den første del af bogen findes en gennemgang af de mange vigtige sidegrene inden for det store syntaktiske

C++-cirkus. Mange C++-programmører har ikke særlig meget erfaring med f.eks. at overdefinere operatører, selv om de måske har læst om fænomenet i en eller anden bog. Jeg har oplevet, at utrolig mange C++-programmører aldrig har brugt templates eller exception-håndtering, og kun ganske få forstod, hvordan man benyttede iostreams ud over ganske simple kald til cout og cin. Første del er et forsøg på at fylde hullerne i din C++-viden, så vi alle starter på samme niveau. Du kan frit vælge, om du vil studere første del af bogen indgående, blot skimme den eller helt springe den over, alt efter dit nuværende C++-niveau.

Termen *viderestilling* dækker over en lang række individuelle emner, som dog alle bygger på det samme koncept, nemlig at et klientobjekt sender en forespørgsel til et andet objekt, som dog med det samme uddelegerer arbejdet til et tredje objekt. Viderestillingen foretages af objektet i midten. Nogen vil sikkert påstå, et det svarer til ordbogens definition af ordet uddelegering, en hjørnestein i objektorienteret design, men i C++ er dette koncept understøttet på en sådan måde, at det går videre end det, som opfattes som uddelegering i andre programmeringssprog. Jeg bruger begrebet pointer meget i denne bog. Faktisk optræder ordet i hvert eneste kapitel. Pointere kan gøre en masse ting i C++. De kan bestemme, hvor på disken, i lageret eller på nettet et givent objekt findes. De kan også se, hvornår objektet bliver slettet, om det bliver opdateret, eller om det er skrivebeskyttet, ja faktisk, om det overhovedet eksisterer, eller om det blot peger på et abstrakt sted i lageret, som

venter på at blive initialiseret. Alt dette kan ske uden nogen aktiv indsats fra den, som bruger pointeren. Vedkommende kan faktisk være helt ligeglad med alle de spidsfindige ting, som sker bag ryggen på vedkommende. Kraftige sager, som dog bygger på en buket af meget simple værktøjer.

Der er blevet skrevet en masse om, hvordan man designer klassehierarkier. Meget af det er ganske fornuftigt, men der findes også meget vrøvl i stil med: "Bare lav objekterne, så de ligner den virkelige verden". De fleste af argumenterne kunne ligeså godt anvendes på andre objektorienterede programmeringssprog, men jeg har ingen intentioner om at forplumre denne C++-bog med mine egne meninger om objektorienteret design. Der er dog en speciel type af nedarvning, nemlig sammenhængende nedarvning, som er overraskende brugbar i forbindelse med C++. I et sammenhængende klassehierarki får alle de nedarvede klasser deres offentlige interface fra en fælles basisklasse. Faktisk er moderen til alle klasserne en helt ren virtuel klasse (pure virtual). Den har ingen datamedlemmer, og alle dens medlemsfunktioner er rene virtuelle funktioner. I C++ er der mange kraftfulde design- og programmeringsmønstre, som passer fint ind i dette koncept.

Ideen bag lagerstyring rummer mere end blot simpel lagerhåndtering. I C++ er det muligt at overskrive operatørerne "new" og "delete". Dette giver mulighed for at bestemme, hvordan objekterne bliver skabt, og hvordan de bliver slettet. Man kan også skabe abstrakte samlinger af objekter,

som gør, at det ikke altid er muligt at se, om man arbejder på et rigtigt eller et abstrakt objekt. Disse højtflyvende tips, som du ser dukke frem i horisonten, er de nye objektorienterede koncepter, som kommer fra firmaer som Microsoft, Apple og Taligent. Disse nye tiltag kræver, at man ændrer sin tankegang om, hvor objekterne befinder sig, og hvordan man kan flytte rundt på dem. Disse nye emner sniger jeg let og elegant ind i lagerstyringsdelen. Lagerstyring er velegnet til at bestemme typen af et givent objekt, noget som desværre mangler i C++. Der bliver talt en masse om at styre lageret i den sidste del af bogen, men de omtalte koncepter kan dog bruges til mange andre ting end blot lagerstyring.

Læsevejledning

Denne bog indeholder ikke som en manual en masse vejledning i, hvordan man håndterer specifikke problemer. Bogen indeholder ideer og udfordringer. Jeg vil opfatte det som en succes, hvis du føler, at du har fået udvidet din C++-værktøjskasse efter at have læse hele denne bog. Jeg vil ikke gøre nogle forsøg på at fortælle dig, hvordan og hvornår du skal bruge disse værktøjer.

Det er umuligt grundigt at indlære alt materialet i et enkelt kapitel, uden at man først har læst alle de andre kapitler, men jeg har dog gjort mit bedste til, at materialet er umiddelbart anvendeligt, når man har læst hvert enkelt kapitel. Hvert kapitel bygger på det foregående, således at man langsomt får bygger sin viden op. Når man så er færdig,

kan materialet også bruges som referenceværk. Bogen kan tjene som en personlig og tit meget savnet opslagsbog om avancerede ideer og koncepter i C++.

Gennem mine år som C++-underviser har jeg lært, at selv erfarne C++-programmører har huller i deres viden. Resten af denne del fylder disse huller ud, så vi alle starter på samme niveau. Der er ikke tale om en introduktion til programmeringssproget, men mere om en C++-version af Trivial Pursuit, som fokuserer på de dele, som bruges senere i bogen. Kapitel 2 er en hulter til bulter gennemgang af programmeringssproget. Kapitel 3 handler om templates, som langsomt bliver mere vigtige, eftersom flere og flere oversættere nu understøtter dem. Kapitel 4 omhandler exception-håndtering på den måde, den er beskrevet i ANSI-standard. Der vil også være nogle få kommentarer til ikke-standardiseret exception-håndtering, som den bruges i den virkelige verden.

Anden del af bogen omhandler pointere: dumme, smarte, smartere, kloge og brillante. Der er tale om den grundsten, som alt andet i denne bog bygger på, og jeg er sikker på, at alle læserne vil få noget ud af denne del.

Tredje del af bogen handler om design og implementation af typer og klassehierarkier i C++. Delen fokuserer på en speciel type af hierarkier, nemlig den sammenhængende variant. Klasseobjekter og andre lignende kuriøse emner tages også op i denne del af bogen. De fleste vil sikkert læse hele denne del, men du er velkommen til at bladre

igennem og vælge det, du synes bedst om. Og bedst som man tror, at alt hvad der er værd at sige om pointere, er blevet sagt, dukker de op igen. Denne gang i forbindelse med de sammenhængende typehierarkier.

Fjerde del af bogen håndterer det mest frygtede C++-emne, nemlig hukommelses- eller lagerstyring. Diskussionen spænder fra det naive, over det dybsindige, til det grotesk konstruerede, men hele tiden ligger vægten på de problemer, en C++-programmør måske løber ind i, og anviser veje til, hvordan man via C++-sprogets faciliteter kan løse problemerne. Jeg opfatter de første par kapitler i denne del som værende nødvendige for, at det er muligt at leve et lykkeligt liv med C++, men hvis man ikke interesserer sig for fjernelse af storskrald, vil jeg foreslå, at man finder på noget mere produktivt at lave end at læse de sidste par kapitler.

Lige et par ord om kodestilen

Specielt 3 ord: Jeg er ligeglad. Er det kortfattet nok? Hvis folk blot ville bruge deres tid på at designe deres programmer i stedet for at spekulere på, hvor } skal stå, ville C++-miljøet blive langt mere produktivt. Jeg mener, at de enkelte projekter skal være konsistente, men i mine 20 år i computerindustrien er jeg aldrig stødt på en bog eller kodeguide, som kan slå et møde på en times tid i starten af et projekt, hvor man fastlægger kodestilen. Jeg har heller aldrig set en bog om kodestil, som kan få en slamkoder til at producere læselig kode. Rent faktisk bruges koderegler

tit mere som en undskyldning for ikke at tage fat på det egentlige problem, nemlig at alle skal skive kode, som er letlæselig. Endelig har jeg aldrig oplevet, at en programmør er i stand til at overbevise en anden om, at vedkommendes kodestil er den rigtige, så argumenter om dette emne er spild af tid.

Jeg har mine egen stil, men i denne bog vil jeg bryde den en gang imellem for at vise de forskellige stilarter, jeg har oplevet. Bogen fokuserer på koncepter, ikke på placeringen af }, eller hvornår man skriver med stort eller småt. Så ved at bruge forskellige stilarter håber jeg, at jeg har fornærmet alle lige meget.

Jeg har også taget mig nogle friheder med inline-medlemsfunktioner, specielt de virtuelle. Den politisk korrekte måde at skrive en inline-medlemsfunktion på er vist nedenfor:

```
class Foo {
public:
    void MemberFn();
};
inline void Foo::MemberFn()
{
...
}
```

Igennem hele denne bog vil det dog være skrevet på følgende måde:

```
class Foo {  
public:  
    void MemberFn() {...}  
};
```

Jeg har endda lavet virtuelle inline-medlemsfunktioner, selv om de fleste oversættere ikke tillader denne syntaks, og gør de det endelig, håndterer de den sjældent korrekt.

Grunden til at vælge denne notation er udelukkende at spare plads. Hvis jeg skrev alle disse inline-funktioner for sig selv, ville bogen blive meget større, og der ville komme sideskift midt i næsten alle programlistningerne. Så du skal ikke tage det alt for tungt med disse inline-funktioner.

Sæt dig i en god stol, tænd for musikken, tag en kop the og – god fornøjelse.