Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2017-10-16

Funktioner och sammansatta datatyper

Mål

Denna laboration handlar primärt om att skapa abstraktioner - att göra något enkelt att använda utan att behöva tänka på hur det fungerar eller dess beståndsdelar. Nu är "användaren" primärt den programmerare som skall använda datatyper och funktioner du skapar.
I detta fall har "användaren" redan skrivit koden (i form av ett testprogram) som använder de typer och funktioner du skall skapa. Du måste alltså skapa dem så att hens program fungerar utan modifikation.
Funktioner (underprogram) är det viktigaste verktygeten programmerare har för att dela upp ett stort problem i hanterbara delar. En funktion löser ett litet delproblem och sätter ett talande namn på lösningsproceduren. Detta skapar en abstraktion - namnet berättar vad funktionen åstadkommer och den är enkel att anropa - men när du anropar behöver du inte bry dig om hur den fungerar internt.
Användning av funktioner med talande namn ger program som är lätta att läsa och förstå. En bra skriven funktion är även enkel att återanvända i nästa program som behöver lösa samma delproblem.
Funktioner samlar ihop kod som löser ett specifikt delproblem och sätter ett namn på det. Samma sak - med liknande fördelar - kan göras med data. Med hjälp av en class, struct, (och ibland en std::tuple eller ett std::pair) kan data som hänger intimt samman samlas i en "låda" som namnges. Detta skapar en datatyp som sedan kan användas för att skapa variabler av "lådtypen". Det är t.ex. så `std::string` är skapad.
I denna laboration ska du skapa en enkel datatyp för att representera en punkt och en mer avancerad datatyp för att representera en "kollissionsbox". I den senare ska du även implementera ett antal så kallade medlemsfunktioner. Du ska noga fundera på hur du kan använda andra abstraktioner, både datatyper och medlemsfunktioner, för att underlätta den du håller på implementera.
Du kan även behöva implementera ett antal fristående hjälpfunktioner för att t.ex. välja det största av två värden (max), det minsta av två värden (min) eller byta plats på två värden (swap).

Läsanvisningar

- Funktioner, parameteröverföring, returvärden (`return`)
  - Inparametrar för enkla datatyper (pass by value)
  - Inparametrar för klassdatatyper (pass by const reference)
  - Utparametrar (pass by reference)
- Funktionsöverlagring (function overloading)
- Sammansatta datatyper (`class`, `struct`[1])
  - Medlemsvariabler (member variable)
  - Åtkomstskydd (`public`, `private`)
  - Medlemsfunktioner (member function, method)
  - Konstruktorer (constructor)
  - Punkt och piloperatorn (., ->)
- Datatypnamngivning (`using` eller `typedef`[2])
- Filuppdelning
- Deklarationsfil (*.h)
- Deklarationsgard
- Implementationsfil (*.cc)

[1] I C++11 rekommenderas `class` och `using`. Konstruktionerna `struct` och `typedef` är gammalt arv från C.
[2] Se fotnot 1

Uppgift: Kollisionsdetektering med Axis-Aligned Bounding Box

Du ska skapa ett programbibliotek för att hantera kollisionsdetektering. I detta representeras objekt med en tätt omslutande rektangel ("bounding box") där varje sida är parallell med antingen x-axeln eller y-axeln ("axis-aligned").
Det är meningen att du skall öva dig i att rita upp egna figurer på papper för att med hjälp av dem själv tänka igenom vad som egentligen står i beskrivningarna nedan. Detta måste du göra för att komma fram till exakta beräkningar och slutlig programkod. Kör du fast visar du dina figurer för assistenten så kan ni diskutera tillsammans.
Koordinatsystemet vi tänker oss har origo i övre vänstra hörnet. Detta gör att x-axeln ökar åt höger och y-axeln ökar nedåt. Detta synsätt är konvention för koordinatsystemet som används för grafiska programfönster.
Programbiblioteket skall bestå av **en struktur för att representera en punkt** och **en klass (ADT) för att representera en Axis-Aligned Bounding Box** (eller kortare "AABB") med funktioner för följande operationer:

(construct) Skapa en AABB genom att låta användande programmerare ange koordinaterna för övre sidan, vänstra sidan, nedre sidan och högra sidan. Detta skall ske via en konstruktor. Kontroll bör ske att vänstra sidan är till vänster om högra sidan och att övre sidan är över undre sidan, och om så är fallet rätta till det. Konstruktorn ska göra att felaktiga AABB's blir omöjliga att skapa. Exempel på deklarationer som skall vara möjliga:

```
int top{0}, left{0}, bottom{10}, right{10};
AABB my_aabb{top, left, bottom, right};
AABB my_aabb{5, 10, 43, 12}; // Boxen (x=10,y=5) till (x=12,y=43)
```

(construct) Som ovan konstruktor skall denna skapa en korrekt AABB, men i denna anges endast två hörnpunkter som inparametrar.

contain Ska vara en medlemsfunktion med två inparametrar `x` och `y`. Funktionen returnerar `true` om $[x, y]$ är en punkt inuti denna AABB.

contain Ska vara en medlemsfunktion med en punkt som inparameter. Funktionen returnerar `true` om punkten finns inuti denna AABB. Denna funktion får inte använda några jämförelser, om du tänker vad du precis gjort ovan så finns ett enklare sätt.

intersect Ska vara en medlemsfunktion med en inparameter `a` av AABB-typ. Funktionen returnerar `true` om någon del av `a` överlappar denna AABB (eller omvänt).

Tips 1: Börja med att rita upp alla möjliga fall av överlapp på ett papper (det finns åtminstone 10 fall om vi räknar med alla möjliga speglingar och rotationer). Problemet *kan* lösas med enbart fyra jämförelser och utan någon if-sats.

Tips 2: Under antagandet att rektangel A's vänstra sida finns till vänster om rektangel B's högra sida så ser du i en uppritad figur enkelt att överlapp (i x-led) finns om rektangel A's högra sida finns till höger om rektangel B's vänstra sida. Motsvarande resonemang kan göras i y-led.

min_bounding_box Ska vara en medlemsfunktion med en inparameter `a` av AABB-typ. Funktionen returnerar den minsta AABB som omsluter både denna AABB och inparametern `a`.

will_not_collide Ska vara en medlemsfunktion med en inparameter `from` av AABB-typ och en inparameter `to` av punkt-typ. Funktionen returnerar `true` om det enkelt går att avgöra att `from` under rätlinjig förflyttningen till `to` omöjligt kan kollidera med denna AABB. Om en tyngre beräkning måste ske för att avgöra om kollision sker returneras `false`.

Tips: För att kontrollera detta undersöker vi om detta objekt överlappar den AABB som omger hela förflyttningen. I fallet att en överlappning finns behöver anroparen utföra en mer detaljerad (långsammare) kontroll för att avgöra om, och i så fall var, en kollision inträffar.

collision_point Ska vara en medlemsfuntion med en inparameter `from` av AABB-typ, en inparameter `to` av punkt-typ och en utparameter `where` av punkt-typ. Funktionen beräknar om, och i så fall var, en kollision med detta objekt inträffar. Om en kollision sker returneras `true` och `where` uppdateras med koordinaterna för övre vänstra hörnet av den AABB som förflytats, när den befinner sig precis före kollisionen. Om ingen kollision sker returneras `false` och `where` lämnas orörd.

Tips: En enkel men ineffektiv metod att genomföra kontrollen är att flytta `from` till `to` i steg så små att den största förflyttningen i x och y-led sker i steg om 1, och kontrollera kollision i varje delsteg. För att få fram antalet förlyttningssteg tar du den största sträckan av förflyttningen i x-led och förflyttningen i y-led. Sträckan som din AABB behöver flytta i varje steg i varje led kan då fås fram genom att dividera avstånden i respektive led med största avståndet (den ena divisionen kommer att ge 1.0 som resultat och den andra kommer bli mellan 0.0 och 1.0). Tänk till så tecken för förflyttningen i varje steg blir korrekta, gör du rätt blir det rätt utan att hantera fyra olika fall.

Tänk till så du inte skriver kod snarlik kod du redan skrivit.

(getters) Ibland är det även bra att ha små funktioner i sin klass som hämtar ut data, t.ex. bredd och höjd. Här får du tänka själv vad som kan behövas. Dessa funktioner är mycket enkla och inte något krav i denna laboration.

Filuppdelning

När du skapar en klass är det normalt förfarande att skriva koden för klassen i två separata filer namngivna efter klassens namn.
I deklarationsfilen skrivs *vad* som finns att använda. Alltså enbart deklarationer av funktioner. Denna fil är för andra programmerare som vill använda din klass. Här ska du inte visa mer än vad som finns för dem att använda i den publika delen av klassen, samt eventuella fristående datatyper eller funktioner som hör ihop med klassen. I privata delen lägger du saker som den som använder klassen inte ska behöva veta om. Du ska anstränga dig lägga så mycket som möjligt i `private` för att få en bra och enkel abstraktion. Ju mindre den som använder klassen behöver veta, ju lättare blir det för hen.
I deklarationsfilen behövs även en speciell "gard" som är till för att förhindra att kompilatorn ser dubbla definitioner av datatyper. En typisk deklarationsgard omsluter hela filen och ser ut som:

```
#ifndef KLASSNAMN_H
#define KLASSNAMN_H

  // allt innehåll i deklarationsfilen

#endif
```

Det är även viktigt att *inte* använda `using namespace` i deklarationsfilen eftersom den som vill inkludera din fil kanske inte vill ha just den namnrymden du använder som default.
I implementationsfilen skrivs sedan definitionen av alla funktioner, *hur* det som finns att använda faktiskt fungerar i detalj. Alltså själva koden som löser uppgiften. Implementationsfilen börjar normalt med att inkludera deklarationsfilen så att kompilatorn kan kontrollera att allt stämmer överens:

```
  #include "klassnamn.h"

  // allt innehåll i implementationsfilen
```

En sista viktig sak att känna till är hur filerna sedan används i ett program. Grundregeln är då att deklarationsfilen *inkluderas* och implementationsfilen *kompileras*. Skulle du råka kompilera en deklarationsfil kommer du sannolikt råka illa ut. Så här skulle ett huvudprogram som använder klassen se ut:

```
  #include "klassnamn.h"

  int main()
  {
    // program som använder klassen 'klassnamn'
  }
```

Och så här kompileras koden för respektive fil för att till slut länkas ihop (sista kommandot) till ett körbart program:

```
$ g++ -Wall -Wextra -pedantic -g -std=c++17 klassnamn.cc
$ g++ -Wall -Wextra -pedantic -g -std=c++17 huvudprogram.cc
```

```
$ g++ klassnamn.o huvudprogram.o
```

Alternativt kompileras båda filerna tillsammans:

```
$ g++ -Wall -Wextra -pedantic -std=c++17 klassnamn.cc huvudprogram.cc
```

Testprogram

Kontrollera att dina funktioner är korrekta genom att testa dem var för sig med en uppsättning indata där du redan vet korrekt svar. Tänk till så att alla fall täcks in. Här är viktigt att rita ut på papper hur olika fall ser ut grafiskt för att kunna täcka in allt.
För att hjälpa dig med detta tillhandahåller vi ett rudimentärt testprogram. Du kan själv lägga till egna testfall genom att modifiera datafilen med testfall. Programmet läser in en rad från testfilen, ritar ut de rektanglar som är aktuella, och ritar längst ned ut en grön eller röd fylld rektangel beroende på om det testfall som prövades gav förväntat resultat eller inte.
Tänk på att färgen är för testet, inte för funktionen. Om en funktion "misslyckas" och ger falskt resultat blir det alltså ändå "grönt" om funktionen *skulle* misslyckas given denna indata.
Det givna testprogrammet kompileras som ovan, men behöver även tillgång till ett programbibliotek som heter SFML. Kompileringsen ser då ut så här:

```
$ g++ -Wall -Wextra -pedantic -g -std=c++17 aabb.cc
$ g++ -Wall -Wextra -pedantic -g -std=c++17 unit_test.cc
$ g++ aabb.o unit_test.o -lsfml-system -lsfml-window -sfml-graphics
```

Eventuellt behöver du börja med att installera SFML på ditt system:

```
$ sudo apt-get install libsfml-dev
```