Inledning

Följande dokument används för att förmedla vanliga fel vi ser vid rättning av inlämnad kod. Observera att detta är generella punkter menat för alla C++-kurser vi håller och därmed kanske inte alla punkter gäller just din kurs.
Dokumentet är indelat i logiska delar (se punkt 1-8) för att lättare kunna hitta i dokumentet.

1 Koduppdelning och struktur (vita tecken)

Många av punkterna i denna del är rekommendationer som går att bryta emot om man använder en egen stil och gör det *konsekvent*. Använder du en publicerad kodstil får du gärna skriva en referens som kommentar i början av filen. Använder din kurs en specifik kodstil ska såklart den användas.

1-1. Deklarera endast en variabel per rad.

Exempelvis pekarvariabler blir då lättare korrekt deklarerade och initierade.

1-2. Skriv endast en sats per rad.

Se till att varje rad gör en sak, händer flera saker blir koden svårläst.

1-3. Använd konsekvent indenteringsstil.

Rekommenderas att indentera med 4 mellanslag per nivå och Allman style för placering av måsvingar. Dvs måsvingar skall aldrig utelämnas, och måsvingar skall alltid stå på egen rad.

1-4. Radbryt långa rader på lämpligaste sätt.

80 tecken per rad rekommenderas.

Vid uppdelning av uttryck med operatorer ska radbrytningen göras innan operatorn.

Efterföljande rader ska indenteras ytterligare för att tydligt visa att uttrycket fortsätter.

1-5. Upprepade, liknande rader bör justeras med extra mellanslag för att tydligt visa likheter.

t.ex.

```
Date start   {    1,  1,  1};
Date now     { 2012, 11,  3};
Date default {  199,  2, 28};
```

1-6. Separera operatorer och operander med mellanslag.

Vid deklarationer av pekare (*) och referenser (&) placeras tecknet intill variabelnamnet för att vara konsekvent med kompilatorns tolkning.

Medlems- och pekaroperatorer separeras ej med blanksteg (. -> *).

Vanliga binära operatorer (+ &&) har blanksteg på båda sidor.

Kommatecken har ej blanksteg innan, och ett efteråt.

Komplett lista finns på Wikipedia i syntax-kolumnen

1-7. Separera programtekniskt olika delar av koden med tomrader.

Använd tomrader så strukturen framhävs. Undvik onödiga tomrader.

Exempel på olika delar: inkluderingar, deklarationer, klassdefinitioner...

Var konsekvent med antalet tomrader som används.

1-8. Gruppera relaterade funktioner. Se även 1-7.

1-9. Deklarera variabler i början av den grupp satser som använder dem.
I korta funktioner (< 10 rader), deklarera alla variabler i början av funktionen.
I längre funktioner (> 20 rader), gruppera satserna i funktionen enligt vad varje grupp satser ska åstadkomma och placera deklarationer i början av varje grupp.

1-10. Undvik djup nästling av satser (block i flera nivåer).
Använd funktioner för att flytta ut kod ur djupt nästlade satser eller strukturera om koden.
Om man returnerar en bool inuti en if-sats kan man returnera jämförelsen direkt.

1-11. Följ konvention för programstruktur.
* Inkludera bibliotek och separata filer.
* Definiera viktiga datatyper och globala konstanter.
* Deklarationer av underprogram.
* Definition av huvudprogrammet.
* Definitioner av underprogram.
Deklarationer och definitioner av hjälpfunktioner placeras med fördel i egna filer som inkluderas.

1-12. Dela upp klasser i headerfiler och implementationsfiler.
a) I headerfilen definieras klassen, och dess funktioner och datamedlemmar deklareras.
b) Korta funktioner med max en sats kan implementeras i headerfilen om det underlättar läsbarheten, övriga funktioner ska definieras i implementationsfillen (.cc)
c) Interna hjälpfunktioner som ej är relevanta för användaren av klassen bör deklareras högst upp i implementationsfilen eller som privata medlemsfunktioner.

2 Generell kodstil

2-1. Använd parenteser för att styra prioritet och associativitet eller för att göra uttryck mer lättlästa, undvik onödiga parenteser i övrigt.

2-2. Använd engelska i kod.
Svenska är ok i kommentarer och litterära strängar om det används konsekvent.

2-3. Använd en konsekvent namngivningsstil.
Detta gäller både språk, användning av gemener och versaler samt separering av ord.
Separera ord med understreck, använd endast gemener för variabler, endast versaler för konstanter och stilen En_Typ för klasser och egna typer.

2-4. Inkludera C-bibliotek enligt C++-stil.
Ta bort ".h" och sätt ett 'c' först i biblioteksnamnet.
Exempel: <stdio.h> ska i C++ skrivas <cstdio>

2-5. Ge tydliga och relevanta felmeddelanden till programmets användare.

2-6. Följ POSIX-konvention för huvudprogrammets returvärde.

Huvudprogrammet (main) ska enligt konvention returnera noll (0) när programmet lyckats, och en noll-skild positiv felkod för varje typ av fel som gör att programmet misslyckas. Värden över 255 bör inte användas.

2-7. Använd konstanter istället för upprepade literaler (DRY - Don't Repeat Yourself).
Konstanter kan även placeras globalt om de används flera gånger.

2-8. Använd std::cout vid vanlig ledtext, std::cerr vid fel som inte kan hanteras.
Exempel på fel: anrop med felaktiga kommandoradsargument, systemfel (exempelvis slut på minne), filer som krävs men inte kan läsas osv.

2-9. Undvik varning om oanvänd parameter genom att inte namnge parametern.

2-10. Undvik explicita hopp i koden.
Satser som gör hopp i koden förstör läsbarhet.
Använd inte goto eller exit.
Undantagsvis kan continue och break användas om det finns en bra anledning.
Flera return-satser bör inte användas i långa funktioner.

2-11. Inkludera samtliga kodbibliotek som används i filen i fråga.
Detta gäller även om de råkar inkluderas av andra headers (exempelvis <string>).

2-12. Undvik fullständig uppräkning.
Likartade test som listas med en mängd if-satser eller villkor skall skrivas om på ett generellt sätt, exempelvis en upprepningssats, rekursion eller sökning i en datastruktur. För att hantera ytterligare ett fall skall programmet inte behöva utökas med ytterligare villkor eller satser, en ändring i en datamängd skall vara tillräcklig.

2-13. Undvik långa eller komplicerade lambdafunktioner.
Implementera större lambdafunktioner som en vanlig funktion (när det går) eller som ett funktionsobjekt (går alltid) med lämpligt namn. Detta flyttar komplexiteten i implementationen samtidigt som namnet på funktionen eller funktionsobjektet tydligt berättar vad som åstadkoms. Abstraktion och läsbarhet uppnås.

2-14. Inkludera endast headerfiler, inte implementationsfiler.
Ange implementationsfil vid kompilering.

2-15. Använd konsekvent namngivning för filer.
Klasser får gärna inledas med stor bokstav.

2-16. Dela upp långa komplicerade funktioner.
Varje funktion bör endast göra en sak för att underlätta läsbarheten och göra det enklare att återanvända funktionalitet. Se även 4-6.

3 Framhäv det viktiga

3-1. Skriv inte funktionsdefinitioner i headerfilen.
Göm dem i implementationsfilen.

3-2. Skriv de publika medlemmarna först i klassdefinitioner.

Det är de publika medlemmarna som är intressanta för den som använder klassen.

3-3. Ta bort kod som bara finns i debugsyfte.
Den är ej intressant i en fungerande slutprodukt.

3-4. Flytta komplexa beräkningar till underprogram.
Beräkningen blir "namngiven" och abstraktion uppnås.

4 Onödig kod

4-1. Undvik onödiga variabler.
Använd däremot hjälpvariabler om de har som syfte att namnge mellansteg i beräkningar eller dela upp långa rader över flera rader.

4-2. Använd inte nyckelordet this i onödan.
this används endast vid behov (exempelvis satsen "return *this") eller för att tydliggöra.

4-3. Upprepa inte manipulatorer med effekt som gäller tills vidare.
Exempelvis setprecision(n) och fixed.

4-4. Undvik att specificera klassnamn före medlemmar som används i medlemsfunktioner.
Klassnamn:: behövs bara i sällsynta fall av arv.

4-5. Inkludera enbart bibliotek som faktiskt används.

4-6. Upprepa inte snarlik kod.
Liknande kod kan abstraheras bort till en funktion alternativt kan koden struktureras om.

4-7. Ta bort kod som inte behövs för programmets funktion.
Kod som aldrig exekveras förstör läsbarheten. Se även 3-3.

4-8. Introducera inte undantagshantering i onödan.
Om inte felet kan hanteras lokalt behövs troligen inget catch-block, felet kommer automatiskt kastas vidare till en hanterare för aktuellt fel.

4-9. Ta bort utkommenterad kod.
Om versionshantering eftersöks bör det göras med t.ex. git.

5 Självförklarande kod

5-1. Välj namn som tydligt visar kodens intention.
Undvik användandet av förkortningar.

5-2. Välj de styrsatser som bäst visar kodens intention.

5-3. Använd standardbiblioteket.
Använd de färdiga algoritmer och datastrukturer som finns i språket istället för att försöka göra samma sak själv.

5-4. Använd referens till konstant (const&) för inparametrar av klasstyp.
Det syns då tydligt att man inte kommer ändra på objektet (const), och det kopieras då ej heller i onödan (&). För grundläggande datatyper (int, char, etc) är det onödigt då det ej är effektivare och endast gör det svårläst.

5-5. Använd dokumenterande kommentarer för kod som inte är självförklarande.
Kommentarer bör beskriva kodens mål, inte hur den når dit. Hur koden når målet visas av de satser och beräkningar som utförs (koden själv). Kommentarer ska tillföra information som annars inte enkelt framgår.
Korta kommentarer som sammanfattar övergripande syftet med de närmaste 5-10 raderna kan vara lagom.
a) Kommentarer placeras på egna rader över det kodavsnitt som kommenteras.

5-6. Undvik negering av stora eller komplicerade villkor.
Villkoren blir lätt svårförståriga vid flera negationer. Se även 3-4

5-7. Undvik komplicerade "smarta" konstruktioner.
Skriv hellre enkel kod som går rakt på sak än att hitta luriga, kluriga, "smarta" lösningar, även om det blir ett par rader extra.

6 Klasskonstruktion

6-1. Visa enbart klassens externa gränssnitt publikt.

6-2. Placera medlemmar som är gemensamma för klasser i en klasshierarki i en gemensam basklass.

6-3. Basklasser med virtuella medlemsfunktioner ska ha virtuell destruktor.

6-4. Nyckelordet virtual används endast i basklassen. Överskuggning visas med override i subklasser.

6-5. Tänk alltid på de sex speciella medlemsfunktionerna.
Fundera alltid på om default-konstruktor, destruktor, kopieringskonstruktor, movekonstruktor, kopieringstilldelning och movetilldelning ska finnas.
a) Om de kompilatorgenererade fungerar och ska finnas bör de deklareras "= default".
b) Om de inte ska finnas deklareras de "= delete"
c) annars ska de implementeras.

6-6. Deklarera medlemsfunktioner som inte ändrar på objektets tillstånd const.
Annars kan de inte användas på konstanta objekt, och andra programmerare ser att en konstant medlemsfunktion inte ändrar på objektet.

6-7. Felaktig användning av static.
Datamedlemmar deklarerade static i klasser motsvarar globala variabler och bör om möjligt undvikas.

6-8. Felaktig användning av mutable.
mutable används endast för datamedlemmar som beskriver ett internt tillstånd, något som inte syns utifrån. Vill du ändra på något tillstånd synligt externt ska din funktion inte vara const-deklarerad.

6-9. Använd ref-qualifier för funktioner som endast ska kunna användas på lvalue-objekt.
Detta så man inte kan göra onödiga modifikationer på temporära objekt.

6-10. Använd explicit för att undvika implicita typkonverteringar.
Konstruktorer som tar en parameter, och andra typkonverterande funktioner, kan annars användas implicit av kompilatorn. Om man vill göra en typkonvertering bör det göras explicit för att vara tydlig. Se även 7-7.

6-11. Överlagrade funktioner och operatorer bör följa standarden vad gäller returvärde och funktionalitet.

6-12. Undvik användande av friend i onödan.
Om en funktion endast behöver använda det publika gränssnittet är det helt överflödigt. I några fall behövs funktionaliteten och ibland underlättar det läsbarheten, i vilka fall det är ok.

7 Kodsäkerhet

7-1. Initiera variabler med klammerparenteser (måsvingar, spetsparenteser) om möjligt.
Undantaget är exempelvis auto som måste initieras med =.

7-2. Använd medlemsinitierarlista för att initiera medlemmar i konstruktorer om möjligt.

7-3. Öppna inte namnrymder i headerfiler (using namespace).
Alla som inkluderar din headerfil kommer automatiskt få direktaccess till öppnade namnrymder.

7-4. Använd medlemsfunktioner som ger const-iteratorer.
Använd cbegin och cend för att öka tydligegheten och undvika fel.

7-5. Använd inte globalt tillgängliga variabler.
Globala konstanter kan dock användas i undantagsfall. Se även punkt 2-7.

7-6. Kontrollera om filöppning misslyckas och hantera felet.
I detta fall kan det vara ok att skriva ut vilken fil som inte kunde öppnas och sedan avsluta programmet, beroende på mål i uppgiften.

7-7. Använd C++-stil för omvandling mellan olika datatyper.
I första hand skall du välja datatyper som undviker behov av typkonvertering. Överstrukna former av typkonvertering ska inte användas.
C-stil (osäker): ~~(int)my_double_var~~
Funktionsform (osäker): ~~int(my_double_var)~~
C++; normal konvertering (typsäker): static_cast<int>(my_double)
C++; tvingad omtolkning (används med måtta): reinterpret_cast<char*>(&my_double)
C++; konvertering mellan basklass- och subklasspekare eller referens (typsäker): dynamic_cast<Sub*>(base_ptr)
C++; konvertering för att ändra const (bör normalt sett inte användas): const_cast<Type>(my_constant_expression)

7-8. Återlämna resurser såsom öppnade filer och dynamiskt allokerat minne så snart det går.

7-9. Undantag ska kastas som värde och fångas som (const-)referens.

Dynamisk allokering ska inte användas. Mottagning med referens gör att även alla subklasser kan fångas korrekt i undantagshanterare.

7-10. Skriv undantagssäker kod.
Tänk specifikt på vad som händer med allokerade resurser då undantag kastas.

7-11. Skriv undantagsneutral kod.
Din kod bör i hög grad vara undantagsneutral, ändra inte på undantagets typ eller kasta nya undantag vid undantagshantering om du inte kan hantera hela felet.

7-12. Använd inläsningsoperationen som villkor i upprepningssatser.
Om du använder funktioner som kontrollerar felflaggor i strömmen kommer du troligen försöka läsa en extra gång.

7-13. Deklarera funktioner noexcept om de inte kommer kasta undantag.
Detta visar användaren av din funktion att undantagshantering inte behövs. Det gör även koden effektivare.

7-14. Använd includeguards (headerguards) i alla headerfiler.
Detta för att undvika fel om flera implementationsfiler inkluderar samma headerfil. Det går även att använda #pragma once.

7-15. Jämför ej flyttal med likhetsoperatorn (==).
Detta för att undvika att avrundningsfel påverkar logiken i programmet. Jämför istället differansen mellan flyttalen och en väldigt litet konstant (cirka 0.0001 är förmodligen tillräkligt).

8 Logiska fel

8-1. Möjlig minnesläcka.

8-2. Möjlig användning av oinitierad variabel.

8-3. Möjlig indexering utanför giltiga gränser.

8-4. Möjlig division med noll.

8-5. Möjligt försök till lagring av värde utanför variabelns värdemängd.
Exempelvis overflow i int.

8-6. Möjlig avreferering av end-iterator eller pekare med nullptr-värde.
Innan du avrefererar en iterator eller pekare måste du alltid säkerställa att den är giltig.

8-7. Felhanteringen vid inläsning är inte korrekt utförd.

8-8. Koden uppfyller inte specifikationen.

8-9. Koden är alldeles för ineffektiv.
Tids- eller rymdkomplexiteten är alldeles för stor.