2D1387 Programsystemkonstruktion med C++
Lösningsförslag, tentamen 25 oktober 2003

Obs: Dessa lösningar är just lösningsförslag. Rättningen kan se något annorlunda ut.

Uppgift 1

Utskriften blir

```
{
  A a;                // A()
  B t;                // A()
  B b = t;            // A(A)              kopiering
  A aa[3] = {a, b};   // A(A), A(A), A()   tre A-objekt
  A &ar1 = aa[1];     //                   referens till A
  A &ar2 = b;         //                   referens till B
  ar1.foo();          // A::foo()
  ar2.foo();          // B::foo()
}                     // sex stycken ~A()
```

Uppgift 2

a) Exempel på implementation som svarar på frågan om [`it2`, `end2`) är en delsekvens av [`it1`, `end1`).

```
template<class It1, class It2>
bool subseq(It1 it1, It1 end1, It2 it2, It2 end2)
{
  for(; it1 != end1; ++it1)
  {
      if(it2 != end2 && *it1 == *it2)
          ++it2;
  }
  return it1 == end1 && it2 == end2;
}

// exempel på anrop:
int a[] = {1, 2, 3, 4};
int b[] = {1, 2, 3};
cout << subseq(a, a+4, b, b+3);  // true
cout << subseq(a, a+4, b, b);    // true, tom sekvens
```

b) Implementationen ovan svarar `true` för en tom sekvens.

Uppgift 3

Konstanter, deklarationer och definitioner:
a) `const A *`: det utpekade minnet är skrivskyddat. `A * const`: man kan inte flytta den pekare man får som returvärde. `const A &`: man kan inte ändra det objekt referensen pekar ut. `foo() const`: man kan inte ändra det objekt som anropade medlemsfunktionen `foo`.
b) `A & const` är fel. Det finns inga skrivskyddade referenser eftersom referensen ändå inte kan tilldelas. Den kan bara initieras.
c) Det värde som skickas till `bar` och `baz` kopieras innan funktionen körs. Om argumentet är konstant eller inte påverkar bara implementationen. De två varianterna är följdaktligen ekvivalenta för anroparen och behandlas därför som samma överlagring.
d) Argument av typen `A &` resp. `const A &` ger två olika överlagringar av funktionen `boo` eftersom de två överlagringarna exponerar olika gränssnitt för användaren (det ena kan ändra indatat, det andra inte). Exempel: i anropet `const A a; boo(a);` kan bara `const A &` anropas och i `boo(A())` kan bara `const A &` anropas av de två eftersom en icke-konstant referens inte kan bindas till en temporär variabel.

Uppgift 4

a) Den första varianten är effektivare eftersom den andra kopierar hela vektorn. Priset blir att man inte behåller det ingående argumentet orört.
b) När man är osäker på vilken syntax en operator har bör man härma beteendet hos den inbyggda typen `int`. Den andra varianten uppfyller detta villkor. Den första varianten har en bakvänd semantik eftersom `a + b` verkar modifiera `b` genom den icke-konstanta referensen.
c) Huvudregel: man bör alltid föredra referenser. Har man inget objekt att referera till kan man kasta ett undantag. Annars bör man använda pekare för att kunna returnera `null`. Pekare används också när man pekar ut första elementet i en array. Problemet är då att mottagaren inte vet storleken på arrayen.

Att returnera pekare gör att gränssnittet blir otydligt. Har nytt minne allokerats och vem är då ansvarig för det minne som pekas ut? Om minne har allokerats, är det med `new`, `new[]` eller `malloc`? Om minnet allokerats med `new[]`, vilken typ av objekt döljer sig bakom `A *`? Om arrayen består av nedärvda objekt finns inget sätt att förmedla detta och `delete[]` och pekararitmetik är direkt olämpliga på pekaren. Dessutom finns de vanliga problemen med pekare. Måste jag testa för `null` vid varje anrop? Pekaren kan vara oinitierad, peka på redan frigjort minne eller ogiltigt minne, t.ex. förbi slutet på en array (såsom `end()` i STL). Syntaxen blir krångligare pga avrefereringen.