1)

p1 och p2 pekar på c (15, 4)

a = (11, 12)

b = (3, 14)

c = (15, 4)

d = (17, 5)

x = 3

y = 3

z = 4

w = 4

2)

Göra den som en referens: int& Foo(int &x, int y) {

return x;

}

3)

const 1)

Kan inte ändra på resultatet som lämnas ut.

const 2)

kommer inte att ändra på argumentet inne i funktionen.

const 3)

funktionen ändrar inget av klassens medlemmars data om de inte markerade som mutable.

En funktion som deklareras int Foo::Bar(int random\_arg) const {} är detsamma som Foo::Bar(const Foo\* this, int random\_arg) const {}

Const värden kalkuleras vid compile time och inte under runtime.

Const kan endast användas för icke statiska medlemms funtioner.

Constexpr kan användas tillsammans med både medlemms och icke medlemms funktioner samt konstruktorer.

4)

En generaliserad pekare. Används för att stega igenom en datastruktur.

Olika sorters iteratorer(5 stycken).

Alla iteratorer kan bli incrementrade samt X b(a); b=a;

Input iterator: Stödjer ==, !=, kan bli dereferancerade som rvalue \*, ->.

Output iterator: Kan bli dereferencerade som lvalues \*a = t, \*a++ = t.

Forward iterator som input och output samt default konstruktor X a; X(). {b=a; \*a++; \*b;}.

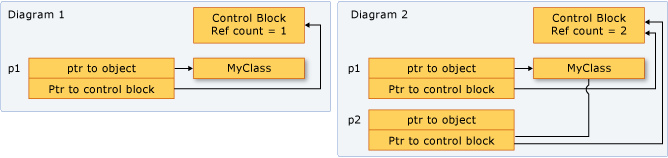
Bidirectional: som ovanstående samt Kan bli decrementrad dvs --a, a--, \*a--.

Random Access: som ovanstående samt operatorerna +, -, <, >, <=, >=, +=, -=, [].

5)

Memory.h tillhandahåller 2 smart pointers. Smart pointers beter sig som vanliga pekare med skillnaden att de deletar objektet de pekar på. De är templates, skicka med viktig info (data typen som den perkar på).

Efter instansiering av shared\_ptr kan användaren kopiera, skicka det som ett argument i funktioneshuvudet, eller tilldela den till andra shared\_ptr. När vi kopierar eller tilldelar en shared\_ptr, varje shared\_ptr håller räkningen på hur många andra shared\_ptr som perkar på samma objekt mha count (count finns tillgängligt i ett kontrollblock som alla shared\_ptr’s har tillgång till). Den ägaren som sist förlorar ägandeskapet av pekaren ansvarar för raderingen. När count == 0 frigör shared\_ptr destruktor alla objekten som har blivit allokerade. shared\_ptr sköter den dynamiska minneshanteringen. Konstruktorn sköter initializeringen.



## Dynamisk minnes allokering

Det finns tre anledningar till att använda dynamisk minnesalokering:

1. Vet inte hur många objekt som behöver allokeras.
2. Vet inte vilken data typen är.
3. Vill dela datan mellan flera objekt.

Det är alltid ok att använda delete på en nullptr även om inget minne har allokerats tidigare för variabeln som är = nullptr;

Funktioner som returnerar pekare till dynamiskt minne frigör sig från ansvaret att deleta objektet utan ansvaret ligger på den som anropar funktionen.

**Problem:**

1. Glömmer frigöra minnet.
2. Använder ett objekt efter dess minne har blivit frigjort.
3. Frigör minnet 2 ggr. Två olika pekare pekar på ett objekt. När den sista kvarvarande pekaren frigör minnet kommer det bli korrumperat. Efter delete kan pekaren sättas till nullptr vilket klargör att det inte pekar på något objekt.

6)

Tre fall:

**Static:** Reserverar minne vid kompilering(compile-time). Frigörs när runtime avslutas.

**Stack (FILO):**, varje gång en funktion deklarerar en ny variabel pushas den till stacken. Varje gång en funktion avslutas kommer alla variabler som är pushade till stacken av den funktionen frigöras(poppas).

**heap:** konstruktorn när man gör new och destruktorn när man gör delete.

7)

value = v sätts efter cout nummer 1, value har skrivits ut vilket genererar storleken av value (85 \* 10^6) vilket är ett odefinerat värde.

I cout numer 2 skrivs 3 ut.

8)

Ta bort det redan allokerade minnet.

Kopiera objekten som högersidan innehåller.

(char\*) allokera minne som är stort nog för det högra värdet + 1.

Undvik självkopiering.

9)

template<typename T>

struct hp {

typedef T type;

};

template<typename T>

struct hp<float> {

typedef double type;

};

Om funktionerna ovan används till något annat genereras kompileringsfel.

10)

Det finns ingen konstruktion c++ som tvingar det att bete sig som ett interface i andra språk, dvs abstrakta funktioner utan implementation.

**Fördelar:** Kan gå ifrån definitionen för att åstadkomma saker men då är det inte längre något interface av definitionen.

**Nackdelar:** Kan inte garantera att alla metoder är abstrakta (det är möjligt att implementera en abstrakt metod samt anropa den). En klass i C++ är abstrakt om det finns minst en abstrakt funktion i den.

**Nackdelar med multipelt arv:** ökad komplexitet. Diamond problem.

Diamond Problem (<http://www.programmerinterview.com/index.php/c-cplusplus/diamond-problem/>). Liger ärver både från Tiger och Lion. Vilka har sin egen kopia av data medlemmarna och metoder från Animal klassen. Liger objektet obj vill inneha 2 subobjekt av bas klassen Animal.

Diamond problem kommer resultera i kompileringsfel eftersom kompilatorn inte vet om obj refererar till kopian från Tiger eller Lion. Ett sätt att lösa problemt är att använda virtual på Tiger och Lion klassen.

