C++

Malte Hildingsson
Lead Platform Programmer
EA DICE

Översikt

- Objektorientering
- Ägandeskap
- Typsäkerhet
- Inline
- Virtual
- Templates
- Standard Template Library
- Minneshantering, -lokalitet, -aliasing

Objektorientering

- “Considering a problem domain and logical solution from the perspective of objects (things, concepts, entities, ...)”
  - Analys
    - Hitta och beskriva koncept i problemområdet
  - Design
    - Definiera objekt och hur dom sammarbetar
  - Implementering
    - C++, C#, Java, Eiffel, ...

Objektorientering

- Three Big Lies (Acton):
  - Software is a platform
    - ”Hardware impacts data design. Data design impacts code choices”
  - Code should be designed around a model of the world
    - “If there's a rocket in the game, rest assured that there is a Rocket class”
    - “… without the basic understanding that where there's one thing, there's probably more than one.”
  - Code is more important than data
    - “Writing a good engine means first and foremost, understanding the data“

Objektorientering

- Tänk på..
  - Det ska vara svårt att använda ett objekt på fel sätt
    - Skriv bra interface
  - Do the right thing, do the thing right
    - Skriv inte mer kod än du behöver
  - En speldesigner kommer aldrig att vara nöjd
    - Skriv flexibelt, gör det lätt att refactorera
      - » Betyder inte fritt fram för snabba hack
      - » GRASP: General Responsibility Assignment Software Patterns

Objektorientering

- Low Coupling
  - Minimera beroenden mellan klasser
    - Forward declaration
    - Polymorfism
    - Pointer To Implementation (Pimpl)
  - Delegera
    - **a.process(b);** <> **a.foo(x)->bar(y)->process(b);**
  - Förenklar återanvändning
    - Modul <> motor
  - Reducerar kompileringstider
    - Full rebuild av Mirror's Edge tar >10 minuter på en quad-core à 8GB RAM

Objektorientering

- High Cohesion
  - Mer fokuserad kod i klasser och funktioner
    - Gör det och bara det som ligger i klassens / funktionens ansvarsområde
    - Tips: Lätta att beskriva och döpa – ParticleEmitter, getCurrentLodLevel()
  - Använd gärna verktyg som STL's <algorithm> eller liknande
    - Lös problem istället för att skriva kod / buggar
      - » Desto mindre text i varje funktion desto bättre
  - En bra funktion är tydligare utan kommentarer

Objektorientering

- Pure Fabrication
  - Skapa hjälpklasser om ansvarsområdet är för stort
    - C++ kan även dra nytta av non-member non-friend-funktioner (Pure Fabrication + Composed Method?)
      - » Inkapsling reduceras med antal medlemsfunktioner
      - » Andra än klassägaren kan bygga till ny funktionalitet

```
struct Actor {
   void addComponent(ActorComponent* c);
};

void addAllComponents(Actor* a, vector<ActorComponent*> c) {
   for_each(c.begin(), c.end(), ...);
}
```

Ägandeskap

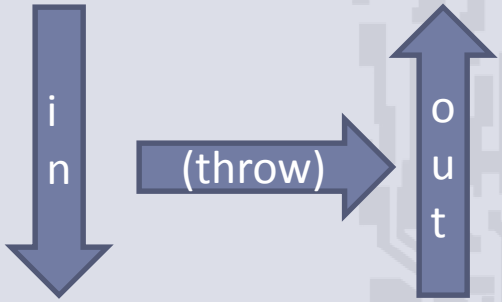
- Stepanov:
  - ”All parts of an object are destroyed when an object is destroyed”
  - ”If two objects share a part, [...] one object is part of the other”
    - ”[I.e.] there is no sharing of parts”
  - ”There is no circularity among objects”
    - ”An object cannot be part of itself, and, therefore, cannot be part of any of its parts” (I.e. acykliskt ägande)

Ägandeskap

- Ex: Scope + composition
  - Skapa/riv alltid allting i rätt ordning, automagiskt
    - Även med exceptions
    - Funkar fint med smart pointers (aggregation)

```
struct GameEngine {
   IOSystem m_iosys;
   Settings m_settings;
   Renderer m_renderer;
};

int main() {
   GameEngine engine;
}
```



Ägandeskap

- Smart pointers (STL, TR1)
  - auto_ptr
    - Delete i destruktor, förflyttning av ägandeskap
  - shared_ptr
    - Delete i destruktor, referensräkning
  - weak_ptr
    - Observer till shared_ptr
    - Nollas automatiskt när shared_ptr går ur scope
  - scoped_ptr
    - Delete i destruktor, ej kopieringsbar
    - Boost (TR2?)

Ägandeskap

- Använd objekt för att hantera livslängden av resurser
  - Ta emot en resurs i konstruktor
    - RAII (Resource Acquisition Is Initialization)
  - Släpp resursen i destruktor
- Placera på heapen (gärna via smart pointer) eller på stacken eller som medlem till ett annat objekt
  - Resursen hanteras automatiskt av scopet

Ägandeskap

```
struct Texture {
    XyzTextureHandle m_handle;
    explicit Texture(XyzTextureHandle h) : m_handle(h) {}
    ~Texture() { XyzReleaseTexture(m_handle); }
};

struct SinglePlayerHud : public Hud {
    Texture m_reticle;
    ...
};

struct GameInfo {
    shared_ptr<Hud> m_hud;
    ...
};
```

Ägandeskap

- Glöm inte att kompilatorn alltid genererar en copy-konstruktor och tilldelningsoperator
  - Säg till om saker inte ska få kopieras!
  - ..eller skriv en vettig copy-konstruktor och operator=

```
struct Noncopyable {
private:
   Noncopyable(const Noncopyable&);
   void operator=(const Noncopyable&);
};

struct Texture : private Noncopyable { ... };
```

Ägandeskap

- Hur bör data exponeras till andra objekt?
  - Getters? –Vad händer om någon refererar datan och ägaren går ur scope?
  - Kanske med ett eget scope? – skicka som const-referens i funktionsanrop av ägaren t.ex.
    - Antyder att argumentet bara är giltigt i anropet, annars får funktionen göra en egen kopia
    - Skicka annan typ, t.ex. pekare, om funktionen får lov att aliasa argumentet (kopiera pekaren)
    - *Functional*esque – färre externa beroenden, reducerar behavioral coupling, mer immutable data; förenklar concurrency

Typsäkerhet

- C++ är hårt typat, undvik casts
  - C++ kan aldrig själv orsaka typfel
    - Vill man avsäga sig en sådan garanti?
  - Downcast is teh evil
    - Dålig design
    - Isolera alltid till implementeringsdetaljer
    - ..i små hjälpfunktioner (undvik exponering)
  - Kan introducera klantfel
    - Vad händer om man castar ett objekt by-value?
  - Kan förhindra vissa kompilatoroptimeringar
    - Håll utkik i kommande slides

Typsäkerhet

- Föredra C++s egna cast-operatorer
  - Tydligare
    - Const, down, up, ... (static_cast)
    - ”Avconsta” (const_cast)
    - Bitwise (reinterpret_cast)
      - » Inte float <-> int
    - Dynamiskt (dynamic_cast)
      - » Kräver RTTI – används sällan p.g.a. performanceskäl
  - Säkrare
    - C++ flyttar runt medlemsdata (ospecificerat, kompilatorspecifikt)
      - » Se speciellt upp med virtual, multipla arv

Typsäkerhet

- Se upp för ofrivilliga, implicita typkonverteringar
  - Gör gärna konstruktorer explicit som default
    - Istället för tvärtom
  - Tänk någon extra gång innan man implementerar implicit-cast operatorn
    - operator T
    - get() / set() istället?

Inline

- Kan undvika många, små funktionsanrop
  - Om prolog + epilog är längre än funktionskroppen bör funktionen alltid inlineas (kolla disasm)
    - Minskar binärstorleken
    - Avlastar instruktionscachen
  - Privata / lokala funktioner som bara anropas på ett ställe i hela programmet inlineas med gott samvete
    - Påverkar inte binärstorleken
- Kompilatorer är bättre på att optimera större block

Inline

- Tänk på..
  - Anropande funktioner blir snabbt större..
    - Mindre minne kvar (tänk på konsollerna)
    - Kan få negativ effekt på instruktionscachen
    - Se upp för inline som anropar inline
    - Se upp för ofrivilliga implicita definitioner
      - » Speciellt med templates
  - Rekursiva funktioner inlineas inte
    - ..inte heller funktionspekare
      - » STL använder istället funktionsobjekt som argument till t.ex. std::sort()
    - ..eller virtuella funktioner

Virtual

```
t = (T*) v;

memcpy(t, v, sizeof(V));
```

OK?

```
struct T
{
   int m;
   void f();
};
```

```
struct V
{
   int m;
   virtual void f();
};
```

Virtual

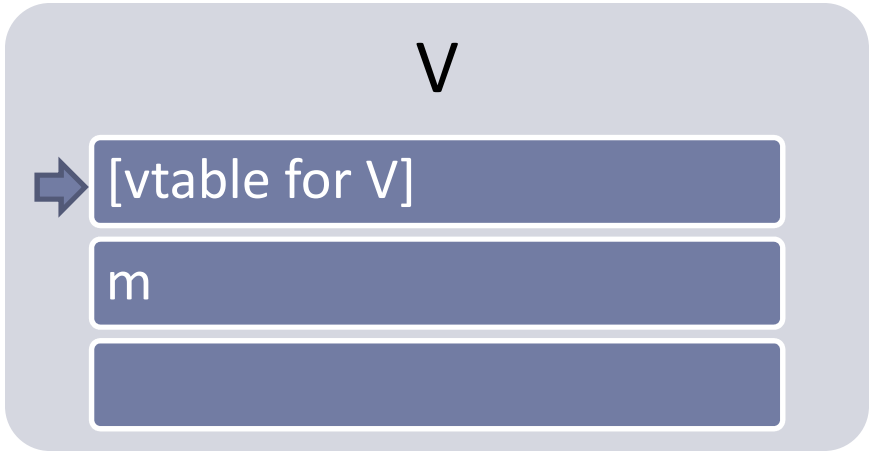
sizeof(T) == 4

```
struct T
{
   int m;
   void f();
};
```

sizeof(V) == 8

```
struct V
{
   int m;
   virtual void f();
};
```

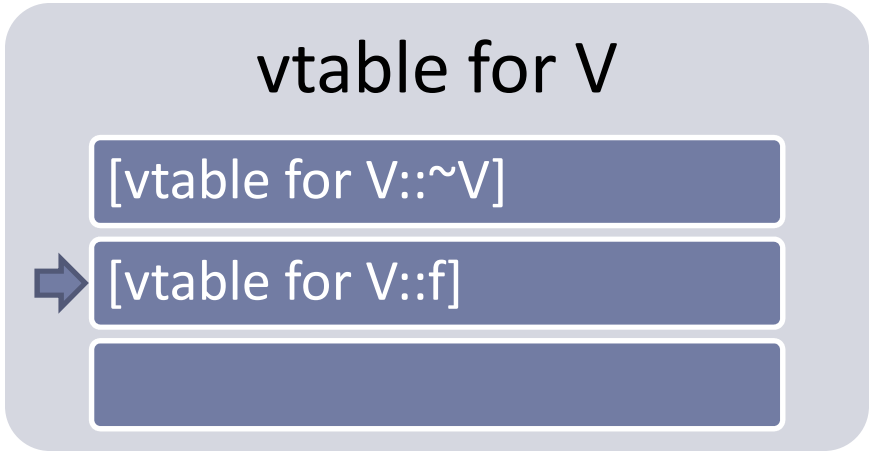
Virtual

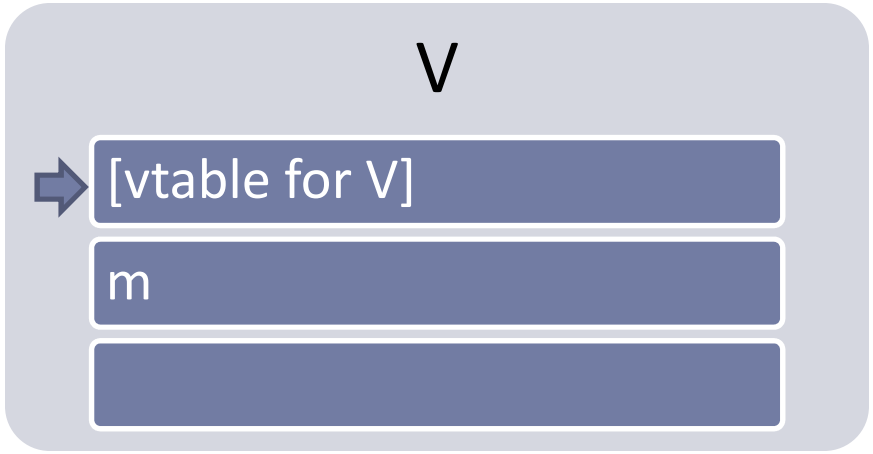



```
struct V
{
   int m;
   virtual void f();
};

V v; v.f();
```

Virtual

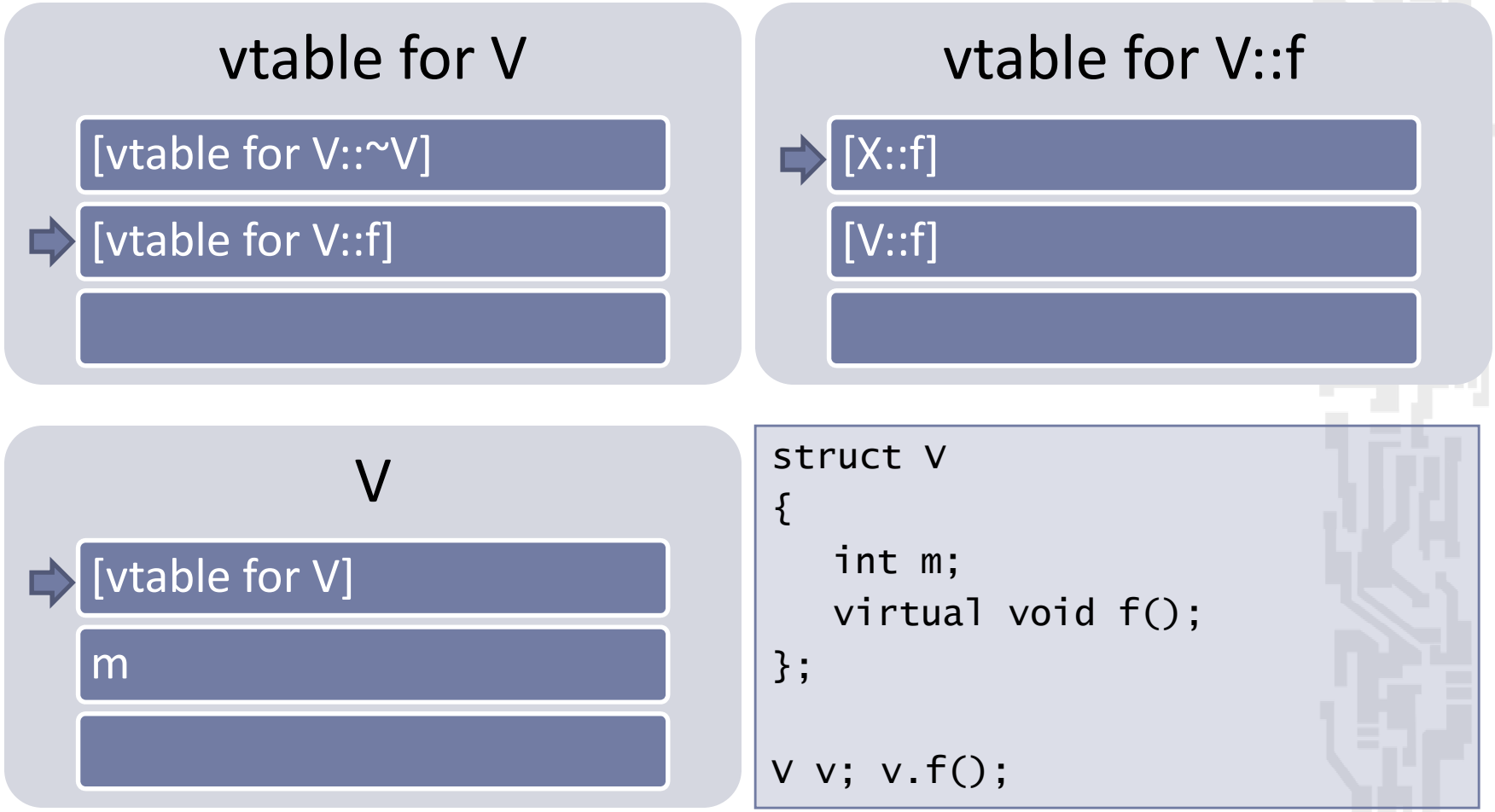





```
struct V
{
   int m;
   virtual void f();
};

V v; v.f();
```

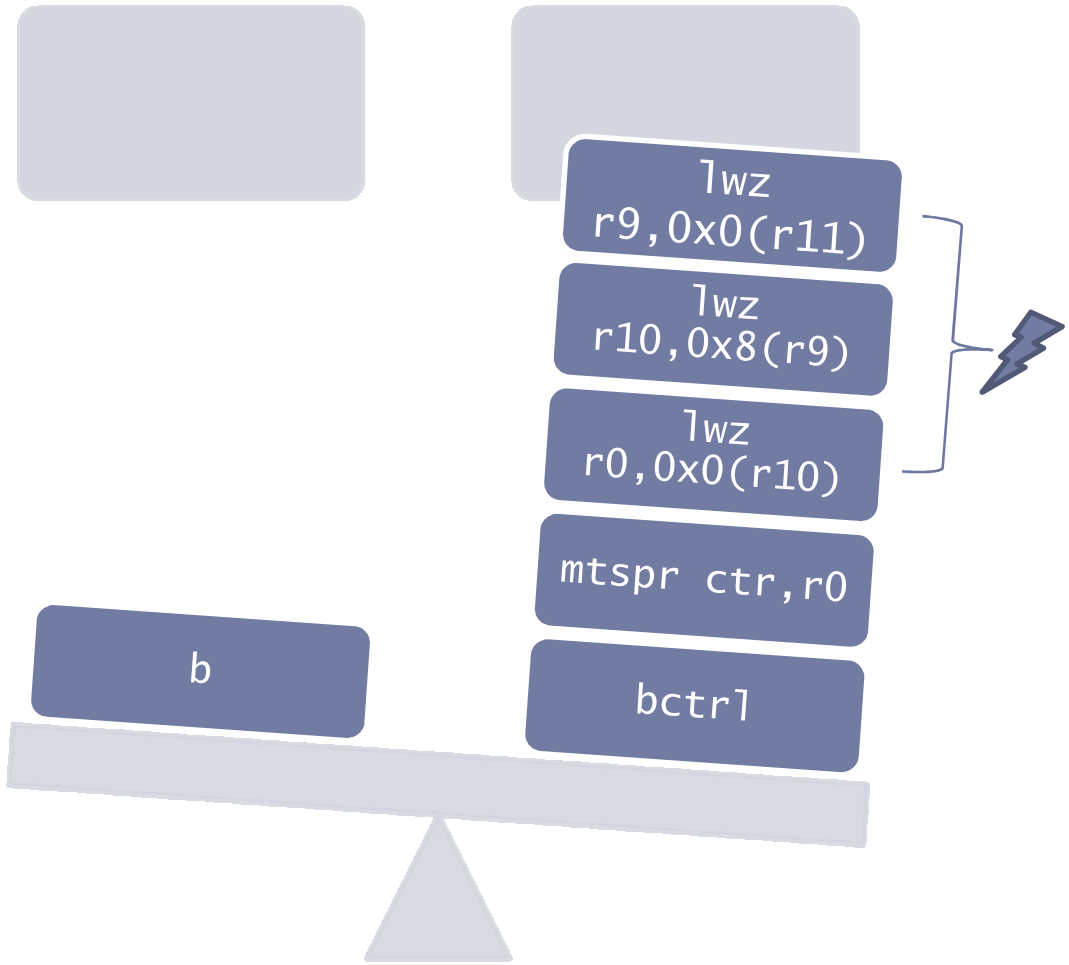
Virtual




```
struct V
{
   int m;
   virtual void f();
};

V v; v.f();
```

Virtual



Virtual

- Virtual *kan* också vara inlinekandidater
  - Bara om typen är välkänd
  - Anropet måste vara direkt till objektet, inte via pekare

```
struct T {
   virtal void f() const = 0;
};

struct V : public T {
   inline virtual void f() const { puts(”hello world”); }
};

V v; v.f();
```

Virtual

- Att ärva eller inte ärva..
  - Låt inte arv vara default
    - Public inheritence i C++ är en Is-a-relation
      - » Använd protected / private för implementationsdetaljer
      - » Mix-ins?
    - Lättare vara lat och komma åt implementationsdetaljer (downcast)
    - Kräver ofta en vtable (destruktor)
    - Fragile Base Class-problemet
  - Glöm inte bort komposition
  - ..eller statisk polymorfism

Templates

- Statisk polymorfism
  - Implicita interface
    - Oskrivna
      - » Uppfylls av giltiga uttryck istället för funktionssignaturer
    - Man får dekryptera kompilatorfel om de inte uppfylls
    - Högre inlärningströskel?
  - Koden väljs under kompilering istället för körning
    - Typsäkert
    - Snabbare
    - Duplicerar koden per typ
      - » Inlineas också potentiellt
      - » Håll koll på storleken!

Templates

- Globala, generiska funktioner
  - Knyter ihop designen för grupper av klasser
    - Alltid samma interface!
    - Bättre inkapsling (så länge dom inte är friend)
  - Som operatorer
  - Klassdesign går mot typdesign
  - normalize(x) oavsett matris / vektor / quaternion, ...
    - Type inference
  - Kuriosa: Stepanov designade från början STL med globala begin(), end() och size()-funktioner

Templates

```
template<class T> inline T abs(T a)
{
   return a < static_cast<T>(0) ? -a : a;
}

template<> inline int abs(int a)
{
   const int m = a >> 31;
   return (a ^ m) - m;
}

template<> inline float abs(float a)
{
   return __fabs(a);
}
```

Standard Template Library

- Förstå hur det är designat
  - “to write programs in the most general terms, and to write programs as efficient as the underlying hardware allows”
- Använd rätt verktyg för uppgiften
  - Tänk på minnesanvändningen
    - vector <> list
  - ..och komplexiteten
    - list::erase(it) – O(1)
    - list::remove(v) – O(n)

Standard Template Library

- <algorithm>, <functional>
  - Generiska funktioner:
    - for_each, find, find_if, find_first_of, replace, replace_if, remove, remove_if, copy, swap, transform, sort, stable_sort, merge, unique, ...
  - Högre ordningens funktioner
  - Stuvade funktioner (currying)
    - bind1st, bind2nd
    - Kompilatorn kan ibland optimera med specialinstruktioner på vissa arkitekturer, t.ex. reciprok:

```
reciprocal = bind1st(divides<float>(), 1.f);
y = reciprocal(x);     // y := 1/x
```

Standard Template Library

```
using namespace std;
set<T*> s;

// #1
for (set<T*>::iterator it = s.begin(); it != s.end();) {
   if (!(*it)->update()) { it = s.erase(it); }
   else { ++it; }
}

// #2
bool notUpdate(T* t) { return !t->update(); }
remove_if(s.begin(), s.end(), notUpdate);

// #3
remove_if(s.begin(), s.end(), not1(mem_fun(&T::update)));
```

Standard Template Library

- Specialisera containers med egna allokatorer
  - Allokera från pooler, ocachead minnesrymd, etc..

```
template<class T> struct SmallObjectAllocator {
   pointer allocate(
        size_t n, std::allocator<void>::const_pointer = 0) { ... }
...
};

template<class T> struct SmallObjectList
   : public std::list<T, SmallObjectAllocator<T> >
{ ... };
```

Minneshantering

- Hantera allokeringar själv
  - Malloc-implementeringar är ofta för generella
    - Slösar minne på overhead för blockhuvud och alignment
    - Ger ibland inte tillbaka minne till operativsystemet (sbrk)
  - Fragmentering
    - ”Failed to allocate 8KB, 4MB free”
    - Konsoller använder inte virtuellt minne
  - Tråd-contention
  - Analysera minnesåtgång per resurstyp, mäta allokeringar per-frame (churn), etc..

Minneshantering

- Ingen generell allokator löser alla problem
  - (På alla plattformar)
- Skapa separata heapar för separata problemområden
  - Flexibelt / tweakbart
  - Bättre cohesion – bättre performance
  - Enklare kod – enklare att bevisa / debugga
  - Välj allokeringspolicy själv
    - Fast storlek eller växdynamiskt? Lockless?
    - FIFO? LIFO? First-fit? Next-fit? Best-fit? Worst-Fit?

Minneshantering

- Överlagra ::new/::delete
  - Kan med fördel inlineas för performance
  - Kompilatorn kan optimera bort if-satsen då n är känd
  - Behåll en icke-inlinead kopia i en .cpp-fil
    - Extern inline är ”weak”-länkat på vissa targets (elf)

```
extern inline void* operator new(size_t n)
{
   return n <= 256 ?
        SmallObjectAllocator::allocate(n) :
        HeapAllocator::allocate(n);
}
```

Minneshantering: objektpool, ex

- Utnyttjar klass-new/delete
- Snabb, typsäker allokering
- Används t.ex. för viss trådkommunikation i Mirror's Edge
  - Känt antal allokeringar per typ per frame
  - Begränsad livslängd (0..1 frame)
  - Trådsäker (lockless)
    - Allokeras av en tråd, avallokeras av en annan
  - LIFO – håll cache-linorna varma

Minneshantering: objektpool, ex

```
template<class T> struct ObjectPool
{
   static void* allocate()
   {
        PooledObject<T>* o;
        do {
                if ((o = m_list) == 0) {
                        break;
                }
        } while (compareAndSwap(m_list, o, o->m_next) != o));
        return o;
   }
   ...
   static PooledObject<T>* m_list;
};
```

Minneshantering: objektpool, ex

```
template<class T> struct PooledObject
   : private Noncopyable
{
   static inline void* operator new(size_t)
   {
        return ObjectPool<T>::allocate();
   }

   static inline void operator delete(void* p)
   {
        ObjectPool<T>::deallocate(p);
   }

   PooledObject<T>* m_next;
};
```

Minneshantering: objektpool, ex

```
class ParticleVertexFactory
   : public PooledObject<ParticleVertexFactory>
{
   ...
};

...

vertexFactory = new ParticleVertexFactory;

...

delete vertexFactory;
```

Minneslokalitet

- Moore's lag gäller inte för minnen
  - Siliconteknikerna blir inte snabbare, bara billigare
  - RAM-hastigheter har rört sig från 50ns till 2ns (25x) över de senaste 30 åren
  - CPU-hastigheterna har stigit med närmare 60% per år de senaste ~20 åren (>10000x)

Minneslokalitet

- Stream-processorer (PS3)
  - Kräver större datamängder för att arbeta effektivt
    - Dubbel- eller trippelbuffra
    - Arbeta medans nästa inläsning sker
- SIMD-instruktioner konsumerar 2-8x mer data än vanliga instruktioner
  - Larrabee kommer ha en 512-bitars vektorenheter (16st 32-bitars flyttal)

Minneslokalitet

- Problemet lappas och lagas med mindre, lokala minnen
  - L1- och L2-cache
  - Scratchpad (PS2, PSP), Local Store (PS3)
- C/C++ ställer krav på programmeraren
  - Organisera (spatial lokalitet)
  - Reducera (mindre / färre / komprimera)
  - Återanvänd (temporal lokalitet)

Minneslokalitet

- Var cache-medveten
  - Gruppera medlemsvariabler som används ihop
  - Kopiera data som används ofta
  - Dela upp i varmt och kallt
    - Allokera kall data separat , i egen struktur
    - Mindre ”onödig” data reducerar cachemissar
- Låt datan styra
  - Arrays of Structures?
  - Structures of Arrays?

Minnesaliasing

- Aliasing innebär minnesreferenser inte nödvändigtvis är unika
- Vad returneras här?
    - Får man anta att a != b?

```
int f(int* a, int* b)
{
   *a = 1;
   *b = 2;
   return *a;
}
```

Minnesaliasing

- Kompilatorn kan bara optimera kod som den säkert vet inte är aliasad
  - Optimeringar får inte orsaka semantikfel
  - Förhindrar t.ex. effektivare instruktionsschedulering, loopoptimeringar, etc.
- Ineffektiv instruktionsschedulering slår hårdast på konsoller
  - Kärnor med in-order-exekvering
  - Orsakar t.ex. load-hit-store stalls

Minnesaliasing

- Problemet blir fort värre i C++
  - **this**-pekaren likvärdig med en global variabel
  - m_n är inte lokal, den kan vara aliasad av m_v och måste läsas om varje iteration:

```
struct
{
   int* m_v, m_n;
   void clear() {
        for (int i = 0; i < this->m_n; ++i)
                this->m_v[i] = 0;
   }
};
```

Minnesaliasing

- En lokal kopia tar bort pekaren från loopen..

```
struct
{
   int* m_v, m_n;
   void clear() {
        for (int i = 0, n = this->m_n; i < n; ++i)
                this->m_v[i] = 0;
   }
};
```

Minnesaliasing

- Kompilatorn kan utföra typbaserad aliasanalys för att reducera problemet
  - ANSI C/C++
    - Säger att en minnesarea bara associeras med *en* typ under dess livstid
    - Antar att aliasing bara finns mellan referenser av *samma* typ
  - Aktiveras i gcc med **–fstrict-aliasing**
- Kan göra mer nytta i C++ än i C
  - Templates, färre casts – enklare att utföra typanalys

Minnesaliasing

- Använd typkvalificeraren **restrict**
  - Ett löfte att minnet som refereras bara addresseras av den kvalificerade variabeln i det scope den deklarerats (eller kopior av den)
  - Aktiveras i gcc med **–std=c99**

```
void* memcpy(
       void*          restrict       s1,
       const void*    restrict       s2,
       size_t                        n);
```

Frågor?

malte.hildingsson@dice.se