

The background is a screenshot from the video game Battlefield 2: Modern Combat. It shows a snowy, mountainous landscape with several buildings, some of which are on fire. A large explosion is visible on the left side. In the foreground, there are silhouettes of soldiers in combat gear. The overall scene is hazy and atmospheric.

Avancerad programmering i C++

KTH 2007-11-21

Måns Vestin, Technical Director

EA DICE

BATTLEFIELD 2
MODERN COMBAT

Innehåll

- Objektorientering
- Beroenden
- Minneshantering
- Templates
- Standard Template Library
- Typsäkerhet
- Run-time type information
- Ägandeskap
- Multitrådning

BATTLEFIELD 2
MODERN COMBAT

Objektorientering

- Objekt = kod + data
- OOP handlar också om att lägga kod och data på rätt ställe
- Tänk alltid: "Vem bestämmer ...?"
 - Vem bestämmer vad som händer när ett fordon krockar?
 - Vem bestämmer hur fienden rör sig?

BATTLEFIELD 2
MODERN COMBAT

Objektorientering, exempel

```
void Game::update()
{
    Vec3 impact;
    if (m_world->collides(m_player, impact))
        m_player->decreaseHealth(impact.length());
    ...
}
```

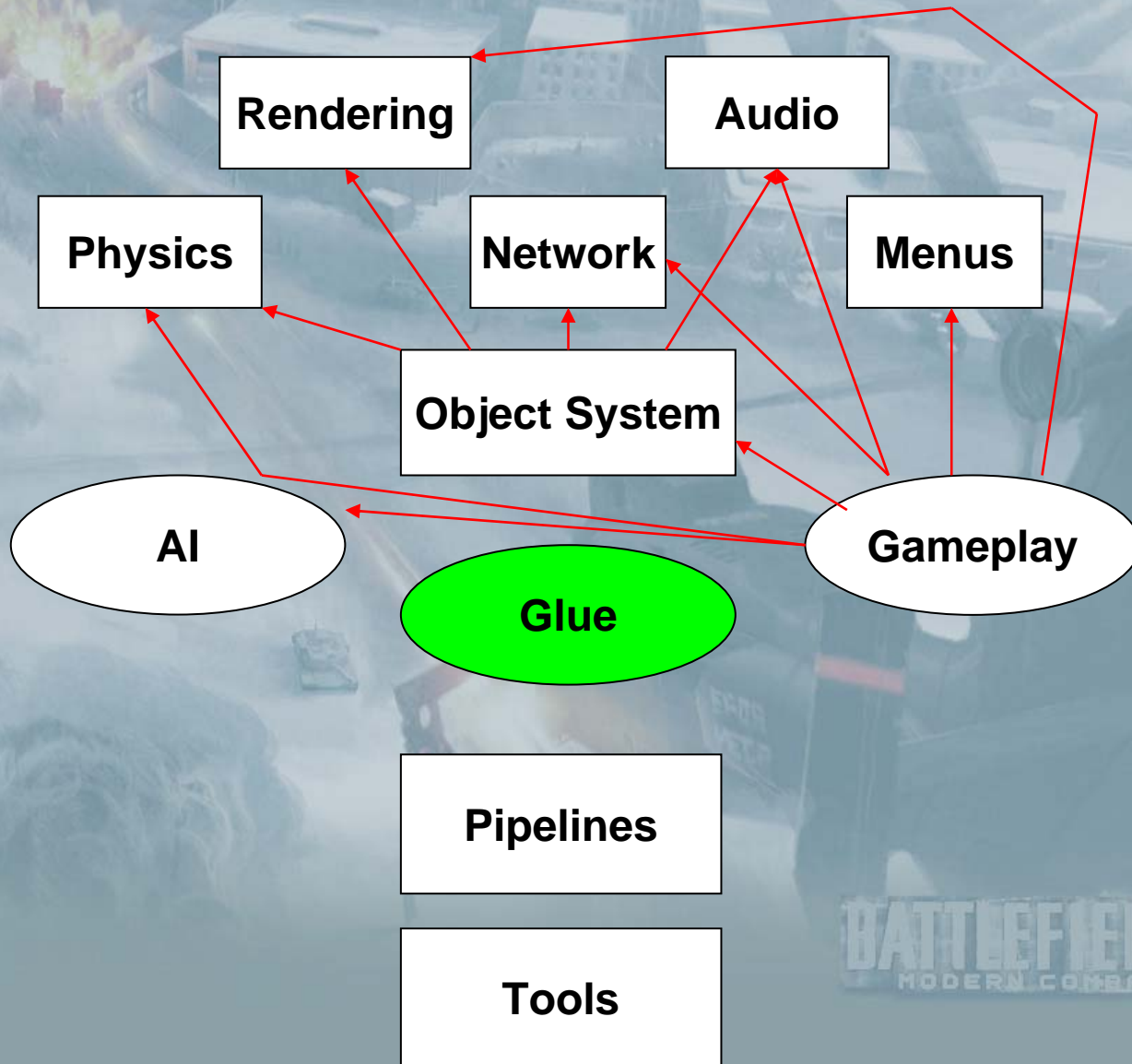
```
void Game::update()
{
    m_collisionManager->update();
    ...
}
```

```
virtual void Player::onCollision(const Vec3& impact)
{
    decreaseHealth(impact.length());
}
```

Objektorientering, virtual

- Virtuella funktioner
 - Syfte?
 - Att kunna lägga logiken på rätt ställe
 - Vad kostar ett dynamiskt anrop jämfört med ett statiskt?
 - V-Table lookup
 - När ska man skriva virtual?
 - När någon jobbar mot det abstrakta interfacet!
 - När man vill ha olika beteende i subklasser
 - Undvik att göra virtuella funktioner som inte alla nedärvda klasser är intresserade av!
 - `Object::onExplode()`

Beroenden



Beroenden

- Varför är beroenden dåliga?
 - Komplexitet
 - Kompileringstider
 - Minneslokalitet
 - Det är lättare att återanvända en modul än en motor

BATTLEFIELD 2
MODERN COMBAT

Forwarddeklarationer

```
#include <Physics/RigidBody.h>

namespace physics {
    class RigidBody;
}

class Vehicle
{
    ...
    physics::RigidBody* m_body;
};
```

Smala interface

```
class Vehicle : public Collidable
{
    // From Collidable
    virtual void onExploded() {
        Renderer::getInstance()->addMesh(m_wreckMesh);
        g_physics_manager->add_rigid_body(m_wreckMesh->body());
        m_callback->onExploded(this);
    }
    class ExplodedCallback {
        virtual void onExploded(Vehicle*) = 0;
    };
    Vehicle(ExplodedCallback*);
    ExplodedCallback* m_callback;
};

class OnVehicleExploded : public Vehicle::ExplodedCallback
{
    void onExploded(Vehicle* v) {
        Renderer::getInstance()->addMesh(v->wreckMesh());
        g_physics_manager->add_rigid_body(v->wreckMesh()->body());
    }
};
```

Delade interface

- Återanvänd interfacet om det förekommer på många ställen

```
void Vehicle::onExplode() {
    m_callback->createMesh(m_model, transform());
}

class CreateMeshCallback
{
    virtual void createMesh(Model*, const Transform&) = 0;
};
```

Old implementation

```
class Level
{
    void addObject(Object*);

    SceneGraph<Object*> m_objects;
    physics::body_list* m_rigidBodies;
    class LevelImpl;
    LevelImpl* m_implementation;
};

class LevelImpl
{
    void addObject(Object*);
    SceneGraph<Object*> m_objects;
    physics::body_list* m_rigidBodies;
};

void Level::addObject(Object* object)
{
    m_implementation->addObject(object);
}
```

Dynamisk bindning

```
class Level
{
    void update(float t);
    list<Vehicle*> m_vehicles;
    list<StaticModel*> m_buildings;
    Terrain* m_terrain;
    list<Object*> m_objects;
};

void Level::update(float t)
{
    for (list<Object*>::const_iterator it = m_objects.begin();
        it != m_objects.end(); ++it) {
        (*it)->update(t);
    }
}
```

Kollision 1: Interface

```
class CollisionHandler
{
    void addCollisionObject(Collidable*);
    void testCollisions();
};

class Collidable
{
    virtual void const BBox& boundingBox() = 0;
    virtual void onCollision(const Vec3& impact) = 0;
};

class Vehicle : public Object, public Collidable
{
    virtual void onCollision(const Vec3& impact);
    ...
};
```

BATTLEFIELD 2
MODERN COMBAT

Design patterns

- Sök efter design patterns
 - Class Factory
 - Generell create-callback
 - State Machine
 - Uppmanar till separata klasser för varje state
 - Singleton
 - Tänk efter om du verkligen vill ha en Singleton eller bara en global variabel!
- Hjälper dig att lösa generella problem
- Gör koden lättläslig

Beroenden, sammanfattning

- Beroenden kostar!
- Inkludera så lite som möjligt
- Tänk på beroenden från början

BATTLEFIELD 2
MODERN COMBAT

Minneshantering

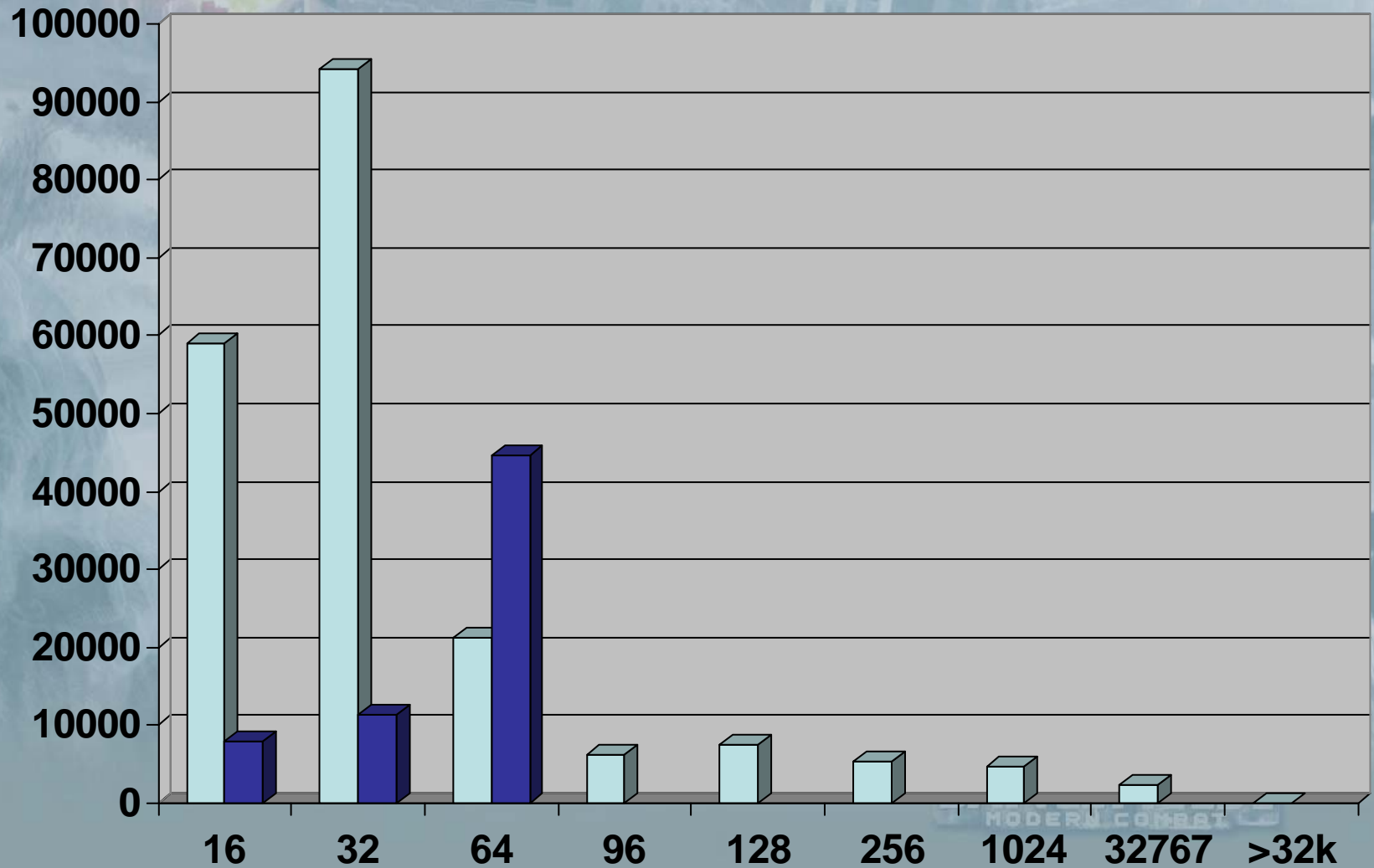
- Tänk på när minne allokeras ...
 - ... men framför allt på VAR det allokeras!
- Cache-missar kostar mer än allokeringar
- Fragmentering
 - Tillräckligt med minne men inte på rätt ställe
- Lös problemet globalt, inte lokalt!
 - Om varje klass har en egen pool så blir kostnaden för stort

Minneshantering

- New i runtime?
 - Vet du vad som händer när new anropas?
 - New är snabb men minne är långsamt
 - Align-overhead per allokering
 - Minnet blir utspritt i heapen
 - Battlefield 2: Modern Combat (PS2):
 - 2 500 - 50 000 dynamiska allokeringar per frame
 - 200 000 öppna allokeringar
 - Skriv en egen alloikator som löser de problem du har (när du identifierat dem)

BATTLEFIELD 2
MODERN COMBAT

Minnesanvändning



Överlagring av new

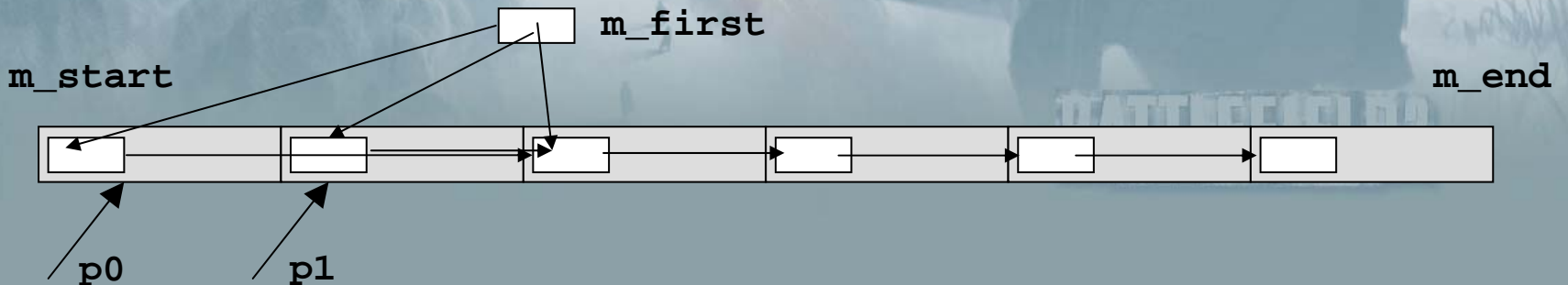
- Problem: Programmet allokerar väldigt många små block.

```
void* operator new(size_t s) {
    if (s <= 64) {
        void* P = SmallBlockAllocator::allocate();
    }
    return ::malloc(s);
}

void operator delete(void* p)
{
    if (!SmallBlockAllocator::free(p))
        ::free(p);
}
```

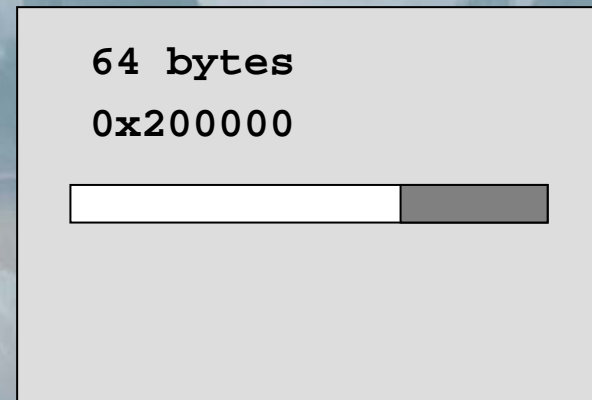
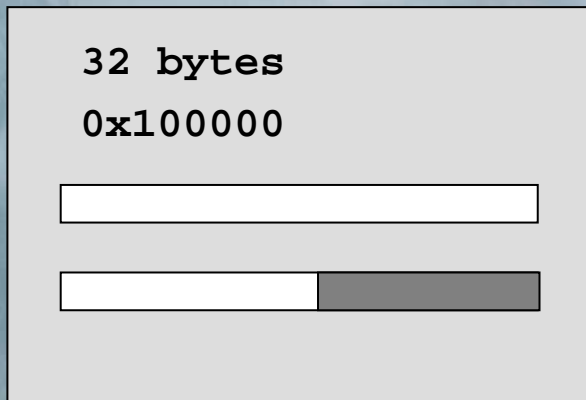
Minneshantering, SmallBlocks

```
void* SmallBlockAllocator::allocate() {  
    void* newMem = m_first;  
    m_first = (u32*)(*m_first);  
    return newMem;  
}  
  
bool SmallBlockAllocator::free(void* p) {  
    if (p < m_start || p >= m_end)  
        return false;  
    *((u32*)p) = (u32)m_first;  
    m_first = p;  
    return true;  
}
```



Minneshantering, forts

- Varje SmallBlockAllocator kan ha flera block, men de ligger i samma virtuella adressrymd
- På så sätt kan vi snabbt slå upp vilken allocator en pekare kommer ifrån



Kollision 2: Minneslokalitet

```
class CollisionHandler
{
    class Collidable
    {
        virtual void onCollision(const Vec3& impact) = 0;
    };

    class CollisionInfo
    {
        Vec3 m_impact;
        Collidable* m_object;
    };

    void addCollisionObject(BBox boundingBox, Collidable* cookie);
    void testCollisions(std::vector<CollisionInfo>& result);
};

for (it = result.begin(); it != result.end(); ++it)
{
    it->m_object->onCollision(it->impact());
}
```

Minneshantering, sammanfattning

- Tänk på var minnet hamnar!
- Lös bara de problem du har!

BATTLEFIELD 2
MODERN COMBAT

Templates

- Ger typsäker kod
- Flyttar beräkningar från run-time till compile time
- Kan användas istf dynamisk bindning
- Ger mycket inlinead kod
- Jobbigare att forward-deklarera
- Kan bli svårläst/svårdebuggat

BATTLEFIELD 2
MODERN COMBAT

Kollision 3: Templates

```
template<class Collidable>
class CollisionHandler
{
    class CollisionInfo
    {
        Vec3 m_impact;
        Collidable* m_object;
    };
    void addCollisionObject(BBox boundingBox, Collidable* object);
    void testCollisions(std::vector<CollisionInfo>& result);
};

CollisionHandler<Vehicle> m_collisionHandler;

for (it = result.begin(); it != result.end(); ++it)
{
    it->m_object->onCollision(it->impact());
}
```

Standard Template Library

- Ska man använda STL?
 - Förstå hur stl är implementerat!
 - Förstå orsaken till implementationen!
 - Vad skulle du implementera annorlunda?
 - Läs Stroustrups förord till "Design and evolution of C++" – The emergence of STL
 - Undvik buggar!

BATTLEFIELD 2
MODERN COMBAT

STL, forts

- Hur ska man använda STL?
 - `list::erase(it)` $O(1)$
 - Om du har en iterator
 - `list::remove(v)` $O(n)$
 - Om du har ett värde
 - `vector::push_back(v)` $O(1)$
 - Ammorterad tid! Max $O(n)$
 - `vector::erase(it)` $O(n)$
 - Byt plats med sista elementet i stället!
 - `string::push_back(v)` $O(n)$
 - Allokerar om minnet varje gång!

STL, forts.

- Hur hanterar stl minne?
 - Hur ser minnet ut i respektive container?
 - Node allocator engine för små block
 - Fragmenteringsproblem

BATTLEFIELD 2
MODERN COMBAT

vector

vector

T* begin

T* end

T* capacity

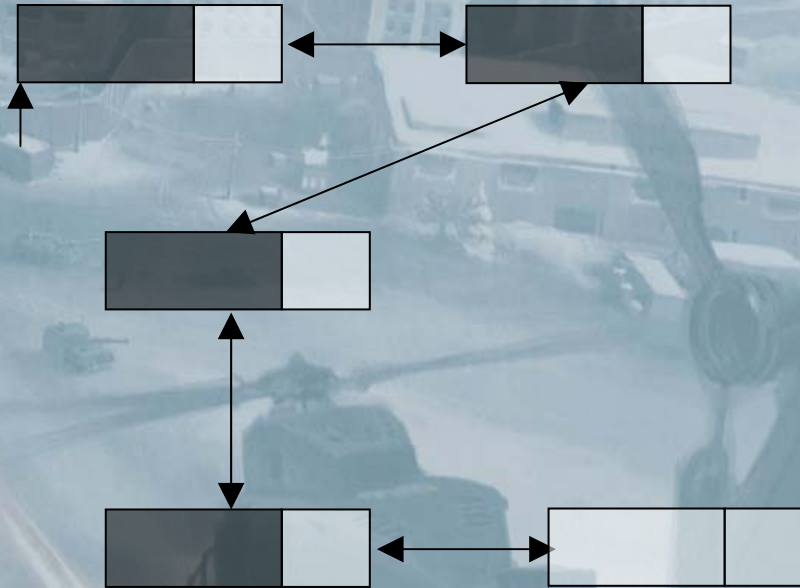


- Måste flytta alla element till en ny array när den gamla blir full
- Måste flytta alla bakomvarande element om man tar bort/lägger till
- vector ÄR en array!
- String ser likadan ut MEN har ingen extra capacity

BATTLEFIELD 2
MODERN COMBAT

list

```
list
class Node
    T value
    Node* next
    Node* prev
Node* begin
Node* end
```



- Behöver aldrig flytta element
- Elementen blir utspridda i minnet
- Overhead per element
- `size()` kan ta lång tid!

STL, forts.

- Traversering
 - Iteratorer möjliggör flera simultana traverseringar
 - Tänk på vilka iteratorer som förstörs vid add/remove!

```
for (iterator it = v.begin(); it != v.end(); ++it) {  
    if (!v->update())  
        it = v.erase(it);  
}
```

```
for (iterator it = v.begin(); it != v.end(); ) {  
    if (!v->update())  
        it = v.erase(it);  
    else  
        ++it;  
}
```

BATTLEFIELD 2
MODERN COMBAT

STL, <algorithm>

- find, find_if, find_first_of
- sort, stable_sort, merge, unique
- lower_bound
- min/max_element
- remove_if
 - No need to worry about iterator invalidation etc!
- stable_partition
- transform

BATTLEFIELD 2
MODERN COMBAT

STL, <algorithm>

- Sortera på värden:

```
std::vector<int> v;  
sort(v.begin(), v.end());
```

- Sortera med en funktion:

```
bool compare(Object* a, Object* b) {  
    return a->getName() < b->getName();  
}
```

```
std::vector<Object*> v;  
sort(v.begin(), v.end(), compare);
```

STL, <functional>

- Functional innehåller funktioner som returnerar funktioner
- <functional> hjälper dig att använda <algorithm>:
 - Skicka med extraparametrar
 - Konvertera C-funktioner
 - Konvertera medlemsfunktioner
 - Hjälpfunktioner: **not**, **and**, **plus** etc.

STL, exempel

```
set<Foo*> s;
for(set<Foo*>::iterator it = s.begin(); it != s.end();) {
    if (!(*it)->update(deltaTime))
        it = s.erase(it);
    else
        ++it;
}

bool update(Foo* f) {
    return !f->update(g_deltaTime);
}
g_deltaTime = deltaTime;
remove_if(s.begin(), s.end(), update);

remove_if(s.begin(), s.end(), not(bind2nd(mem_fun(&Foo::update),
    deltaTime)));
```

STL, sammanfattning

- Tänk på hur containrarna är implementerade!
- Tänk på vad det får för konsekvenser!
 - Komplexitet
 - Minne

BATTLEFIELD 2
MODERN COMBAT

Typsäkerhet

- Typsäkra program ger kompileringsfel i stället för runtime-fel
- Undvik att hacka runt typkontrollen!
- Cast-operatorer:
 - C-style cast: $a = (A^*)b$
 - `static_cast`
 - `dynamic_cast`
 - `const_cast`
 - `reinterpret_cast`

BATTLEFIELD 2
MODERN COMBAT

Typsäkerhet, forts.

- C-cast castar vad som helst utan varning!
- `static_cast` castar nedåt i arvshierarkier
- `dynamic_cast` castar nedåt i arvshierarkier med run-time check
- `const_cast` castar bort `const`
- `reinterpret_cast` castar icke-relaterade typer

```
Vehicle* v = (Vehicle*)object;  
Vehicle* v = (Vehicle*)soldier;  
  
Vehicle* v = static_cast<Vehicle*>(object);  
  
if (dynamic_cast<Vehicle*>(object)) ...  
  
Vehicle* v = const_cast<Vehicle*>(constVehicle);  
  
Vehicle* v = reinterpret_cast<Vehicle*>(soldier);  
Vehicle* v = reinterpret_cast<Vehicle*>(256);
```

RTTI

- Run-Time Type Information (Rtti)?
 - Två användningsområden
 - getInstance(Type) - factory
 - getType(Instance) - type testing
 - Global eller lokal typinfo?
 - Performance
 - Logik hamnar utanför objekten

BATTLEFIELD 2
MODERN COMBAT

Rtti - exempel

```
class Node;
class ModelNode : public Node;
class TransformNode : public Node;

void SceneGraph::load(Stream* s)
{
    type_info t;
    s >> t;
    m_rootNode = factory->create(t);
    m_rootNode->load(s);
}

void ModelNode::load(Stream* s)
{
    s >> m_model;
    s >> m_materials;
}
```

Rtti, exempel

```
void SceneGraph::traverse(Node* node)
{
    renderer->push();
    if (typeid(*node) == typeid(ModelNode))
    {
        ModelNode* m = static_cast<ModelNode*>(node);
        renderer->draw(m->mesh());
    }
    else if (typeid(*node) == typeid(TransformNode))
    {
        TransformNode* t = static_cast<TransformNode*>(node);
        renderer->setTransform(t->transform());
    }
    for_each(node->begin(), node->end(),
            bind1st(mem_fun(&SceneGraph::traverse), this));
    renderer->pop();
};
```

Rtti, exempel

```
void SceneGraph::traverse(Node* node)
{
    renderer->push();
    node->render(renderer);
    for_each(node->begin(), node->end(),
             bind1st(mem_fun(&SceneGraph::traverse), this));
    renderer->pop();
};
```

```
void TransformNode::render(Renderer* r)
{
    r->setTransform(m_transform);
}
```

```
void ModelNode::render(Renderer* r)
{
    r->draw(m_mesh);
}
```

BATTLEFIELD 2
MODERN COMBAT

Typsäkerhet och rtti, sammanfattning

- Tänk alltid efter en extra gång innan du skriver en cast!
- Se till att koden ligger på rätt ställe!

BATTLEFIELD 2
MODERN COMBAT

Ägandeskap

- Håll reda på vem som äger objekt
 - Level eller Parent?
- Skilj på ägande och referenser
 - Använd gärna olika pekartyper
- Håll reda på när objekt tas bort
 - Weak pointer eller callbacks
- RAI – Resource Acquisition Is Initialization
 - Betyder att det ska finnas ett objekt som äger varje resurs
 - Objektet ska ha samma scope som resursen
 - `std::auto_ptr<T>` för dynamiskt minne

Ägandeskap, forts

- `scoped_ptr`
 - Delete i destruktorn
- `auto_ptr`
 - Transfer of ownership
- `shared_ptr`
 - Flera ägare av samma objekt
 - Referens-räkning
- `weak_ptr`
 - Observer till `shared_ptr` – ökar inte referensräkningen
 - Nollställs när `shared_ptr` dör

BATTLEFIELD 2
MODERN COMBAT

Multitrådning

- Datorerna har fler och fler processorer
 - Xbox 360: 3 CPU:s, 6 trådar
 - PS3: 1 CPU, 2 trådar, 6 SPU:s
 - PC: 1-8 CPU:s, 2-16 trådar
 - Trådarna blir fler men långsammare!
- Varje klockcykel där vi inte gör 6-8 olika saker samtidigt är bortkastad

BATTLEFIELD 2
MODERN COMBAT

Multitrådning, forts.

- Hitta saker som kan göras parallellt
 - På hög nivå
 - Spellogik och rendering
 - Server och klient
 - På låg nivå
 - Kollisionstester – Raycasts
 - Animationer

BATTLEFIELD 2
MODERN COMBAT

Multitrådning, problem

- Datalokalitet
 - Kod som delar data måste låsas
 - Critical sections kring data-access
 - Det gör koden långsam igen
 - Svårt att veta vad som behöver låsas
 - Dubbelbuffring av data kräver mer minne
 - En buffer som skrivs, en som läses
 - Visa CPU:er kräver helt lokal data
 - PS3 SPU:s

Multitrådning, exempel

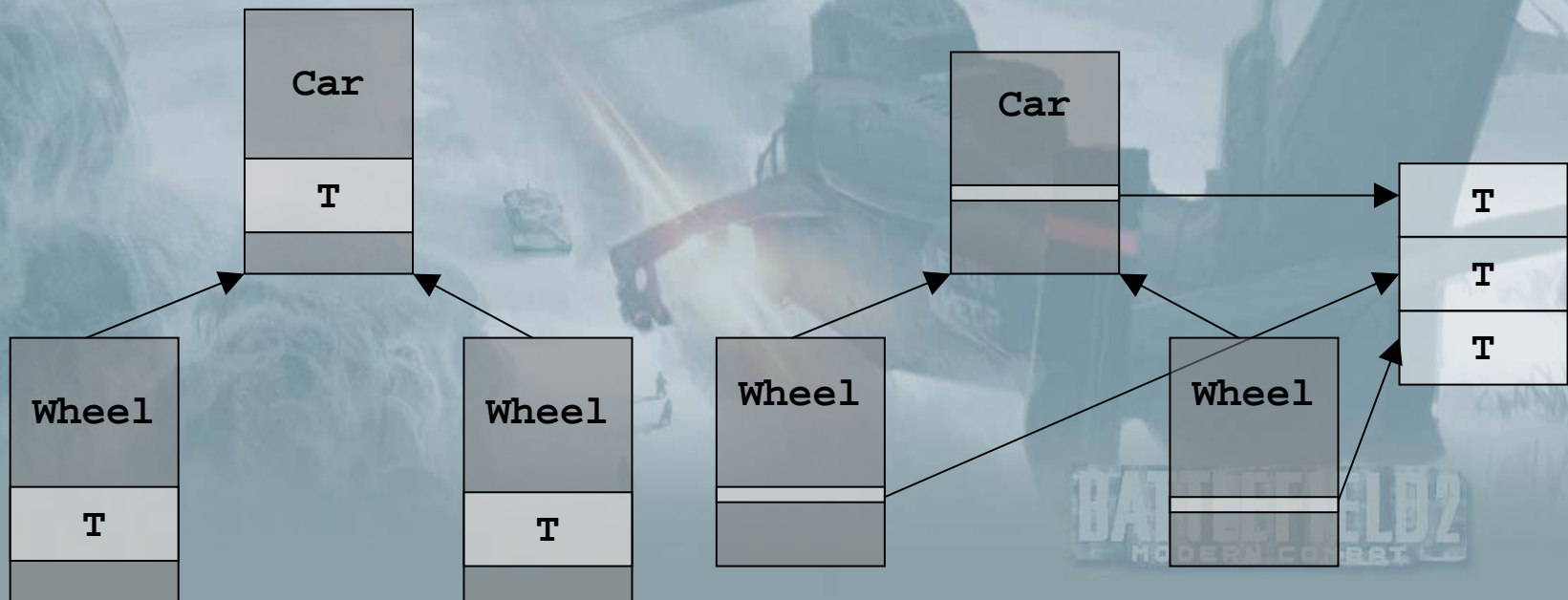
- Uppdatera olika typer av objekt
 - Svårt att hålla reda på vad update() gör
 - Vi måste sortera objekt efter typ för att få större kontroll

```
class Level
{
    void update(float t);

    list<Object*> m_objects;
    map<Type, list<Object*>> m_objects;
};
```

Multitrådning, exempel

- Uppdatering av transformationer
 - Trädhierarki av objekt som flyttas
 - Minnet ligger utspritt
 - Bryt ut Transformationerna till en egen lista



Multitrådning, sammanfattning

- Leta efter kod som kan köras parallellt
- Var försiktig med data-access
- Tänk på data-lokalitet
- Var beredd på att designa om din kod

BATTLEFIELD 2
MODERN COMBAT

A screenshot from the video game Battlefield 2: Modern Combat. The scene is a snowy, mountainous landscape. In the foreground, a soldier in a dark uniform is seen from behind, aiming a rifle. A bright orange and yellow explosion is visible in the mid-ground on the left. In the background, there are several buildings, some of which appear to be damaged or on fire. The overall atmosphere is hazy and overcast.

Frågor?

BATTLEFIELD 2
MODERN COMBAT