Game Engine Architecture
Spring 2017
1. Modern C++ (11/14) for Game Engines

Juha Vihavainen
University of Helsinki

[Gregory, Ch. 3 *Fundamentals of SE for Games*]
[Josuttis, Ch. 3 *New Language Features*]
[Lakos, *Large-Scale C++ Software Design*]

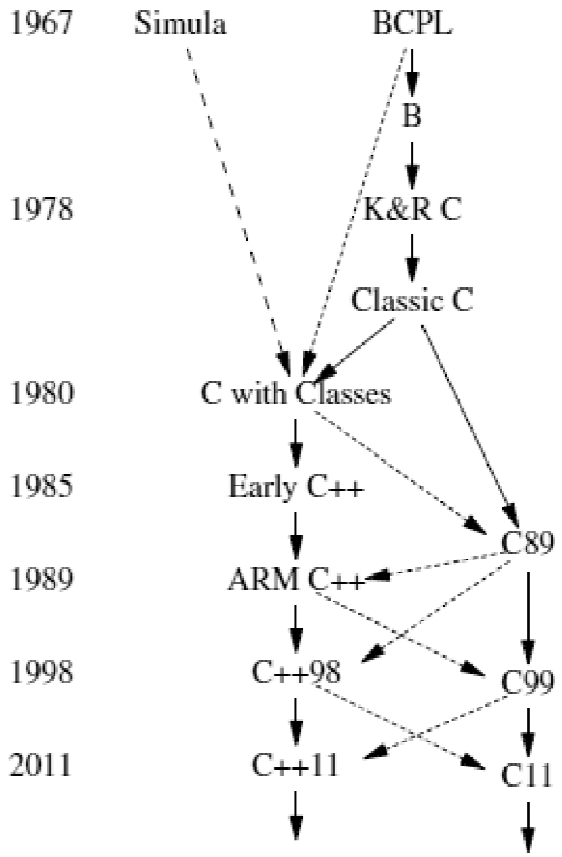
On the development of C++

1979 **=>** B.Stroustrup: ***Adding Classes to C****. in Software Practice and Experience,* Feb, 1983, pp. 139-161.

- 1998 ISO C++ Standard officially adopted. 776 p.
  RTTI, namespaces, **export**, **bool**, new cast syntax, many template extensions, exception specifications, etc., **std::string**, *STL*
- 2003 ISO standard revision (bug fix release for C++98)
- 2003 *Library* TR1: likely components, *hash* map, regex, smart pointers..
- 2006 *Performance* TR: A report on C++ performance issues
- 2009: Selected “C++0x” features became commonly available
- 2011 ISO Standard ("C++0x" => "C++11/14"). 1353 p.

  auto, **static_assert**, **constexpr**, *uniform initialization*, *move semantic, lambda expressions, threads, futures, locks*, spec. of "machine model", template aliases, variadic templates, **extern** ("do not instantiate here")
  - no "*concepts*" for generics (yet); removed **export**, **auto_ptr**, and old version of exception specifications (that had proved to be bad ideas)

A (simplified) family tree of C++ (Stroustup, 2014)



modelling experience with Simula (PhD thesis) => how to efficiently utilize classes and inheritance

.. other influences: Clu, Algol68, Ada: exceptions, generics, overloading..

On C++ type safety (refreshment)

C++ is **not** guaranteed to be statically or dynamically type safe

- "a programming language designed for general and performance critical systems programming with the ability to manipulate hardware cannot be type safe" [Stroustrup]
- the goal of preserved *C compatibility* assures such "unsafety", too

Some problems (mostly features inherited from C)

- untagged unions (binary data reinterpreted as the current variant)
- explicit (and some implicit) unsafe type conversions (casts)
- pointers and arrays without (guaranteed) checks
- ability to deallocate a free store (heap) object while holding on to its "dangling" pointer (allowing post-allocation access)
- deallocating an object not allocated, double deallocations..

C++ application areas

- C++ is used in many key areas of the software industry
  - computer games and entertainment industry
  - virtual machines and translator tools (compilers)
  - audio/video processing
  - computer device drivers
  - control systems
  - telecommunications systems
  - embedded software systems
  - simulation systems
  - medical imaging
- Its implementation and runtime systems make it suitable to its application areas
- C++ has many technicalities, subtleties, and features

Challenges of C++ (modified from Stroustrup)

- preventing potential misuses of manual resource handling (mainly, **delete**)
- designing sound class hierarchies, and using obscure design patterns such as the *Visitor* pattern
- difficulty of generic programming and templates ("template meta-programming")
- combining object-oriented and generic programming
- making exceptions suitable for hard-real-time: eliminating overheads from code unrelated to error handling (?)
- providing higher-level concurrency models as libraries (?)
- *coming*: type case, new forms of asserts for prog-by-contract

New syntax and semantics (2011)

- new uniform *initialization syntax*
  - old C++ provided many *different* ways to initialize objects
  - now, one new way **{ . . . }** works similarly for (most) situations
- new keywords: **static_assert**, **constexpr**, **noexcept**
- new *contextual* keywords: **final override** (ids but in special context)
- *meaning changed* (*extended*)**: auto**, **delete**, **default**
- *move semantics*: can efficiently *move over* a whole (internal) data representation
- *lambda* expressions, with various kinds of *captured variables*
- *template aliases*, **extern** & *variadic templates*
- support for using *user-defined* "*literals*"..
- *threads*, *futures*, *locks*, specification of "*machine model*" (memory)

C++11 features to prefer

- **auto** (type derivation for variables, and range loops)
- **nullptr** (replacing 0 or NULL; for smart pointers, too)
- range-based-**for**-loops (**for (int & e : container) . . .**)
- "*override*" and "*final*" for **virtual** functions overrides (~ Java/C#)
- *scoped* enums (qualified by enum name, no integral conversions)
- *smart pointers* (resource handling, and exception safety)
- *lambdas* (can be used for, e.g., *STL algorithms* and *concurrency*)
- non-member **begin () and end ()** (work with any type)
- **static_assert** (compile-time check) and *type traits* (at compile-time, to query or modify the properties of types)
- *move semantics* (efficiently construct/copy a value)

New **noexcept** specification

- A **noexcept** specification enables a programmer to inform the compiler that a function *should* not **throw** exceptions
  - an improved (with less overhead) version of the old C++03 "**throw()**" specification which is deprecated in C++11
- The compiler can use this information to enable certain optimizations on such non-throwing functions
  - for example, containers such as **std::vector** will move their elements if the elements' *move* constructor is **noexcept**, and copy otherwise (- needed to support exception safety)
- If a function marked **noexcept** allows an uncaught exception to escape at runtime, => *fatal error*, and **std::terminate** is called immediately

Motivation: how to return data efficiently?

New C++11 solution: use *move semantics* to "steal the rep"

- To support the convential ("natural") algorithm and notations:

```
Vector operator + (Vector const& a, Vector const& b) {
    Vector r;
    // for each i, copy a [i] + b [i] into r [i]
    return r;
}  . . .
Vector res = a + b;                    // clear and natural
```

a temporary object is created here (recognized by the compiler)

- To provide an efficient implementation, the **Vector** type needs to provide a new special *move constructor*

Copying state vs. moving state

- *Moving* is a key new C++11 idea
  - usually an optimization of copying
- C++ has always supported copying object state:
  - *copy constructors, copy assignment operators*
- C++11 now adds support for requests to *move object state*:

```
Vector w1;    . . .                    // STL-style vector
Vector w2 (w1);                        // (1) copy w1's state to w2
Vector w3;    . . .
Vector w4 (std::move (w3));            // (2) move w3's state to w4
```

Note: Above, the **w3** continues to exist in a valid state after creation of **w4** (generally: *valid* but *indeterminate* state!); here *valid* means that the object can be destructed (by the compiler)

C++11: *move* constructors

- Often, the constructor gets a temporary that *can* lose its value
- Implement the transfer of internal representation

```
class Vector  {  . . .          // illustrative pseudocode, only
    Vector (Vector&& a)  noexcept {  // no const here
        rep = a.rep;            // *this gets (steals) a's elements
        a.rep = nullptr;        // a becomes empty
    }    . . .
private:
    Rep * rep;                  // some hypothetical representation Rep
};    . . .

Vector res = a + b;             // now, the temporary result is
                                // efficiently passed on
```

lvalues vs. *rvalues* (simplified)

- An *lvalue* is an expression that refers to a *memory location*, and, e.g., allows us to take the address (&) of that location
- Essentially, an *rvalue* is any expression that is not an *lvalue* (often, a *temporary* result); examples of *lvalues* and *rvalues*:

```
int i = 42;  i = 43;                  // ok, i is an lvalue
int * p = &i;                         // ok, i is an lvalue
int& foo ();    foo () = 42;          // ok, foo () is an lvalue
int * p1 = &foo ();                   // ok, foo () is an lvalue
int foobar ();   int j = foobar ();   // ok, foobar () is an rvalue
int * p2 = &foobar ();                // error: address of an rvalue
```


Rvalue reference (&&) type

- If **X** is any type, then **X&&** is called an *rvalue reference* to X
  - *implemented* much like the ordinary reference **X&** (pointers)
  - the ordinary reference **X&** is now called an *lvalue reference*
- Rvalue reference parameters allow a new kind of function overloading resolution

```
void fn (X&& x);          // rvalue reference overload
. . .
fn (X ());                // an rvalue: calls fn (X&&)
```

- This kind of overload is *usually* used only for constructors and assignment operators, to achieve *move semantics*

Summary: handling results efficiently

- Temporary objects are prime candidates for moving:

```
Vector createVector ();  // produce a Vector (func prototype!)
Vector v;                                  . . .
v = createVector ();     // in C++98, copies value to v
```

- C++11 now (often) turns such copy operations into *move* requests:

```
Vector v;                                  . . .
v = createVector ();     // implicit move request in C++11
```


A (very simplified) STL-*style* container *Seq*

```
class Seq {                                  // a hypothetical container for strings
public:
    Seq ();                                  // default ctor: make empty Seq
    explicit Seq (int n);                    // ctor: initialize to n elements
    Seq (std::initializer_list <string>) ;   // ctor: initialize with a list { . . . }
    ~Seq ();                                 // dtor: deallocate elements
    int size () const;                       // number of elements
    String& operator [] (int i);             // access the i'th element
    void pushBack (String const& x);         // add a new item at the end
    String * begin ();                       // first element
    String * end ();   . . .                 // one-beyond-last element
private:
    int sz;                                  // number of elements
    String * elem;                           // pointer to sz elements
};
```

Note

What is missing?

Defining *coping* and *moving* within a class

```
class Seq {
public:
    // . . .
    Seq (Seq const&);                    // copy constructor
    Seq (Seq &&) noexcept;               // move constructor
    Seq& operator = (Seq const&);        // copy assignment
    Seq& operator = (Seq &&) noexcept;   // move assignment
    // . . .
};
```


Writing *move* member functions

- Write your move overloads in such a way that they cannot throw exceptions
  - that is often both natural and trivial, because move semantics typically do no more than manipulate pointers and resource handles between two objects
- If you succeed in *not throwing* any exceptions from your overloads, then
  - make sure to advertise that fact using the new **noexcept** keyword (see the previous slide)

Move constructor and move assignment

```
Seq::Seq (Seq&& v) noexcept                  // no const here
   : sz { v.sz },  elem { v.elem } {         // grab v's elements
     v.elem = nullptr;  v.sz = 0;            // and make v empty
}

Seq& Seq::operator = (Seq&& v) noexcept  {  // no const here
     delete elem;                            // dtors can't throw
     elem = v.elem;  sz = v.sz;              // grab v's elements
     v.elem = nullptr;  v.sz = 0;            // make v empty
     return *this;
}
```

- Now, (depending on the context) the compiler can "automatically" use appropriately either copy or move operations

A move *request* by **std::move**

- In C++11, there is an standard library function called **std::move** that can be used to suggest/enforce a move (e.g., inside an implementation)
- It is a function that turns its given argument into an *rvalue reference* without doing anything else
- *Essentially*, for a type **T**, it's **std::move** function is defined

```
T&& move (T& a)  noexcept  {   // from template std::move
     return static_cast <T&&> (a);
}
. . .
std::string s = "Hello";                . . .
std::vector <std::string> v;            . . .
v.push_back (std::move (s));            // the contents of s is moved
```

Some STL details/technicalities: when to *move*

- Depending on circumstances, C++11 (generally) turns copy operations on *rvalues* into *move* operations, but not always
  - some operations (e.g., **std::vector::push_back**) offer the *strong exception*-safety guarantee, so moving can replace copying *only* if the move operations *are known* not to throw (e.g., by declaring them **noexcept**)
- Moving a whole container (generally) requires that the container's allocator be movable, which need not (necessarily) be the case
- If the allocator is not movable, the elements of the container must be individually copied, unless the element type's move constructor is known not to throw, in which case they may be moved
  - a utility **std::move_if_noexcept** gives an *rvalue reference* for a **noexcept** move constructor, otherwise an *lvalue reference*

C++11 provides *move* constructors

- The standard-library containers now support *move* constructors and *move* assignments, *for example*

  std::vector

  std::list

  std::forward_list (singly-linked list)

  std::map

  std::unordered_map (hash table)

  std::set

  std::string . . .

- Of course, recommended for any new custom data structures, too

Sometimes, moving allowed – but *not* copying

- Generally, most standard types in C++11 are *move-enabled*
  - they support move requests
  - e.g., STL containers
- But some types are actually *move-only*: copying is prohibited, but moving is allowed
  - often kind of abstractions that are (or represent) somehow "unique" values and don't have a copy operation that would make any sense
    - e.g., **stream** objects, **std::thread** objects, **std::unique_ptr**, etc.
  - only allowed overloads are provided

Lambda expressions

Using lambdas with of STL libraries and types

- **for**-loop vs . **for_each** () + lambda
- can give identical performance on several compilers

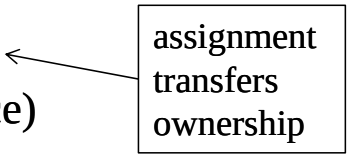
```
std::vector <int> v = {  35, 92, 68 };     //  init with new syntax
sum = 0;
1. for (std::vector <int>::size_type i = 0; i < v.size (); ++i)
     sum += v [i];
2. //   using STL algorithm + lambda
   for_each (v.begin (), v.end (), [&sum](int x) { sum += x; } );

3. for (auto x : v) sum += x;    //  range for; type deduced from v
```

old style C++ container-for-loop; or could use iterators

here, explicitly captured by reference

Resource management (revisited)

- Remember the RAII: *resource acquisition is initialization*
- Many classes include resource management as part of their fundamental semantics
  - e.g., **std::vector**, **std::ostream**, **std::thread**..
- "*Smart pointers*" can be used to address many of the remaining problems of destruction and leaks
  - **std::unique_ptr** for (unique) ownership
    - practically zero cost (in time and space)
  - **std::shared_ptr** for shared ownership
    - maintains *use counts* (some options provided..)
  - but still using *any* pointers may result in unwanted sharing – even in a single-threaded program



Issues in architecture of C++ programs

The basic principles of *object-oriented programming*

- *programming to an interface*, class hierarchies, frameworks..
- *design patterns*: problem solutions (inheritance & composition)
  - sample patterns in C++:
    Template Method, *Singleton*, *Bridge*, *Factory*

Large-scale C++ programming (game engines, virtual machines,..)

- what if software is "big" (> 10000) or "huge" (>100000)
- we should take modularization very seriously!

How to physically organize a *large* C++ program into files

- modularity, header files, and breaking unwanted dependencies
- logical vs. physical dependencies between program units
- proper use of *namespaces*

On modularity

- Important goals in software construction are *maintainability* and *reusability*
  - both are supported by *modularity*: minimizing the dependencies between program units or subsystems (components/engines/layers)
- Background and ideas:
  - interfaces (services) are often more stable than implementations
  - try to make client code as independent as possible from changes in implementations (by providing *representation independence*)
  - C++ classes (as *abstract data types*) support a kind of logical modularity (with its **public** vs. **private** parts)
    - but **private** members still come along as a part of the class definition, and *often* create unnecessary dependencies
  - in C++, try to pass *minimal* information to other program units - preferably include *no headers* at all (replace by *name stubs*)

Headers and physical dependencies

- In C++, the actual *physical dependencies* between program units are manifasted in *header files* (since they are shared around)
  - a class definition and its implementation (**private** data members) are strongly connected, and changes in the **private** part cause recompilation of the client's code (since sizes may have changed)
  - also, concrete object creation builds strong dependencies
    - again, code that *creates* objects compiles only if the compiler has access to the *definitions* and *the sizes of the types* of the data members (even if they are **private**)
- The *Bridge* design pattern can be used to completely separate interface hierarchies from implementation structures
- Using *Factory* patterns can break dependencies and decouple program units (discussed with design patterns)

Problem: unnecessary header dependencies

```
// file: "data.h"
class Data  { . . . };          // some data definition
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// file: "client.h"  -  a header file that needs Data
#include "data.h"  . . .      // get the Data class definition
class Client {                // compiles OK, but bad style
    Data query () const;  . . .
private:
    Data * ptrData_;
};
```

- Now, any changes to **Data** propagate to **Client**-related source code
- The physical dependence (file include) may create maintenance problems - or, sometimes, force long-lasting recompilations

Required class information?

What information is actually required from the class **X** in order to compile client code? For example:

```
    X obj;      // here compiler needs to know instance size
. . . new X;    //    to allocate space for the object
```

However, this information is not required for:

(1) members with pointers or references, e.g., **X ***, **X&**, or **X&&**

(2) function *declarations* even with value parameters or results

X getX () or **void print (X par)** need only *name declaration* of **X**

- the *caller* of the operations needs the definitions to determine the required sizes - to actually to create/pass values (within a *.cpp* file)
- thus, often header files don't actually require full class definitions in order to define their own services and interfaces

Breaking unwanted dependencies

```
// file "client.h" - better modified header file
class Data;                    // forward declaration only (name stub)
class Client {
public:
  Data query () const; . . .   // OK: no implementation needed (yet)
private:
  Data * pData_; . . .         // OK: no implementation needed (yet)
};
```

- Only source code that actually creates objects needs to include appropriate header files
  - e.g., "*client.cpp*" usually needs to include "*data.h*"
  - but *no problem* since *it* is an *isolated* program *unit*

Problems with templates and headers

- Not all classes can be forwarded with names only (with a name stub)
  - **std::string** is a **typedef** of a template instance (not named classs)
- The standard library provides a special header **<iosfwd>,** with
  - minimal declarations for stream templates and their standard *typedef*s, such as **std::ostream** => optimize compilation process
  - unfortunately, no such forward header file exists for **std::string**
  - similar practice recommended for user-defined headers: **myfwd.h**
    - collect the *minimal name info* (name stubs) into a header
- Note that C++ implementations sometimes **#include** extra header files along system headers (for internal purposes), making code nonportable
  - in another platform, such "missing" headers may break compilation
  - minimize dependencies on nested **#include**s by always including system headers as the last ones

Idiom: Pimpl (*P*ointer to *impl*.)

- Also known as the *Handle-Body* idiom (or "compilation firewall", or "Cheshire Cat" technique: "leaving only the smile behind")
- The whole class definition, with its **private** and **protected** parts, is unnecessarily carried along when compiling the client's code
  - this physical coupling hinders maintenance
- To completely separate abstraction and its implementation, replace an object/data by a pointer to it (and omit all data definitions)

```
class Abstraction  {                    // the handle part
    . . .
private:                                // the private part is replaced
    struct Impl * body_;                //   by a pointer to the body
};
// the struct Impl is defined and hidden in a .cpp file
```

the pointer size doesn't change

C++ global name space

- *Namespace* defines a *scope*, and anything that can be globally defined can also be defined in a namespace
- All declarations that are not explicitly placed in a named namespace become part of the one *global namespace* (a bad thing!)

Rule

- *Don't ever corrupt the global namespace with user-defined names* (*but a limited set of top-level namespace names*)

Historically, ***String*** classes.

Accessing the global namespace:

- inside a namespace, in definitions of its members, you can refer to a name in the outermost global namespace using the scope operator **::**

```
::GlobalName  // refer to a file-scope name, from anywhere
```

Namespaces

- A namespace definition is *open*:
  - can add members to the existing namespace by writing later a new namespace section **{ . . . }** with the *same name*: means extension
  - for example, assuming an earlier definition of *Company*, you can add an additional member to this namespace:

```
namespace Company {        // extending an existing namespace
    class Volunteer { . . . };   // adds a new member
}
```

- Now, we can distribute a namespace into multiple header files, each defining some closely related set of specific services
  - e.g., needed for defining the standard namespace *std* (lately, the use of just a *single* namespace *std* has been criticized)
- No keywords to specify the *accessibility of individual* members
  - but we *can* hide them inside an *unnamed namespace*; see later

Namespaces (cont.)

- A namespace may contain only the *declaration* of a member (in a *.h*)
- The corresponding *definitions* can be provided *outside* of the namespace (in the *.cpp* file); must use *Namespace*::*Name* to specify the member

```
namespace Company {                    // .h
    void foo ();                       // declare a function
    class Employee {  . . .  };        // plus: declare operations
}  // . . .
```

```
void Company::foo () { . . . }         // .cpp: definition of operation
void Company::Employee::op () { . . . } // parameters must match
```

- This strategy makes it completely explicit in which namespace the name is originally defined (and may prevent and expose potential mixups)

Unnamed namespaces

- To limit the scope and accessibility of a variable, function, or class to a *single* file, place it in an *unnamed namespace*
- Replaces C's global (file-scope) **static** declarations (=> internal name)

```
namespace {          //  a hypothetical "<UNNAMED>" namespace
    int value_;      //  private variable (zero by default)
    void SetValue (int value) {    //  also private
        value_= k;
    }
}  //  if the same file continues,  we have ..
//  here implicit: "using namespace <UNNAMED>"
SetValue (77);    //  compiler sees:  <UNNAMED>::SetValue
```

- Members of unnamed namespaces can be used without qualification but *only in the same file*

using declarations

- Use fully qualified names (*std*::*cout*), *or* import often used names with **using** *declarations*

```
using Company::print;          //  adds a local declaration
print ();                      //  means: Company::print ()
```

- Since the **using** declaration *adds a declaration to the scope*, it can prevent and reveal dangerous conflicts (name collisions):

```
void goo () {
    int value;                 . . .
    using NamespaceX::value;   //  error: trying redeclare
    using Company::print;      //  OK: can declare locally
    print ();                  //  Company::print
    ::print ();        //  access some (odd legacy) global print
}
```

Namespace *aliases*

- We should use *descriptive* and *unambiguous* names for our namespaces
  - but this may lead to long and cumbersome symbols
- To simplify the use of long names, we can introduce one or more *local* aliases:

```
namespace CompanyFromSomewhere { . . . }
namespace Company = CompanyFromSomewhere;   // alias
Company::Employee e;              //  OK: uses alias
```


Style rules for namespaces

- Use namespaces to express logical structure, and don't define components outside namespaces
  - place related classes, interfaces, etc., in a common namespace
- *Don't ever use a* **using** *directive*: local declarations may then silently (accidentally) override imported names
  - tutorials often use **using** directives, but *it's bad style*
  - Stroustrup: a transient strategy (needed only for outdated compilers to make namespaces work)
- *A header file may not contain even* **using** *declarations*:
  - use *fully qualified* names to make all header files independent of modifications of namespaces (system/own)
- Use the *Namespace*::*Member* notation when *defining* (implementing) namespace members (see slide 36)

C++ application architecture: Summary

- Use namespaces, and **using** *declarations* (never *directives*)
- Always write *full namespace paths* in header files
- Don't reveal implementation details in header files, and include only those headers that are needed to make your source to compile
  - prefer using *class forward declarations* (*name stubs*)
  - supported by writing special name-fwd headers ("*myFwd.h*")
- Use the *Pimpl* idiom to separate abstraction and its implementation
- Hide those implementation parts inside an *unnamed namespace* within implementation units (*.cpp* files)
- First include user-defined (custom) header files
- Use *design patterns* to make some aspects (structure or behavior) of software separate and thus easily manageable/changeable
  - remember basic patterns: *Singleton, Bridge, Factory, Strategy*..