Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #8

Template and Serialization

Lecture #8

Part I : Template

Introduction

- Suppose that we want a function for calculating the maximum of two numbers

```
float getMax(float a, float b) {
   if (a < b)
        return b;
   else
        return a;
   // equivalent to return (a < b) ? b : a
}
```

Introduction

- Suppose that we want a function for calculating the maximum of two integers
  - write another function

```
int getMax(int a, int b) {
   if (a < b) return b;
   else return a;
}
```

  - or call with type casting

```
int a, b, result;
a = 10;
b = 4;
result = (int) getMax((float) a, (float) b);
```

Introduction

- Suppose that we want a function for calculating the maximum of two objects
- We cannot avoid the typecasting problem by using void pointers

```
void * getMax(void * a, void * b) {
   if (*a < *b)    // illegal indirection
       return b;
   else return a;
}
```

  – we need to do a typecast anyway to existing type inside the function

```
if (*((Player *)a) < *((Player *)b)) //...
```

Introduction

- Only way is to rewrite the getMax function for your own class

```
Player getMax(Player a, Player b) {
   if (a < b) return b;
   else return a;
}
```

- Or use inheritance from a special object
  - but all objects need to inherit from it, and we need to write wrappers for primitives types

Introduction

- Use inheritance from a special object

```
class CompObj {
public:
   virtual bool operator < (CompObj &) const = 0;
}

class Player : public CompObj {
public:
   bool operator < (const CompObj & o) const {
        return level < (static_cast<const Player&>(o)).level;}
// ...
}

const CompObj& getMax(const CompObj& a, const CompObj& b) {
   if (a < b) return b;
   else return a;
}
```

Introduction

- Use inheritance from a special object

```
int main () {

   Player p1 (10);
   Player p2 (2);

   // assignment to pointer
   Player * result1 = (Player *) &getMax(p1,p2);

   // assignment to object
   Player result2 = static_cast<const Player&>(getMax(p1,p2));

   return 0;
}
```

Introduction

- The solution: generic programming
- C++ supports programming using types as parameters through compile-time techniques
- Template classes and functions can be parameterized by types

```
template <typename T> class GenericClass;
template <typename T> T& genericFunction(T&, T&);
```

  – or (typename vs. class)

```
template <class T> class GenericClass;
template <class T> T& genericFunction(T&, T&);
```

Function templates

- Defining the getMax function as a template

```
template <typename T>
const T& getMax(const T& a, const T& b) {
   if (a < b)
        return b;
   else
        return a;
}
```

Function templates

- Using the getMax function in your code

```
int main() {
   float a, b, c;
   a = 3.0; b = 4.0;
   c = getMax<float>(a,b);

   double x, y, z;
   x = 8.78; y = 6.45;
   z = getMax<double>(x,y);

   Player p1 (10);
   Player p2 (2);
   Player result = getMax<Player>(p1,p2);
   cout << "Max level: " << result.level << endl;

   return 0;
}
```

Function templates

- The type T is used to *instantiate* the getMax template function
  - it can be used anywhere within the function to declare objects of that type

```
template <typename T>
const T getMax(const T& a, const T& b) {
   T result;
   result = (a < b) ? b : a;
   return result;
}
// cannot return const T& as local (temporary) variable
```

Function templates

- At compile-time, the C++ template automatically generates the function(s) where each appearance of T is replaced by the type passed as parameter
  - This process is automatically performed by the compiler and is invisible to the programmer
  - The compiler then creates the appropriate instantiation of the function
  - The type name T is commonly used but any identifier is valid

Function templates

- Any class T needs to implement the operators used in the template!
  - Example: all classes T used to create a getMax<T> need to implement the operator <

Function templates

- In case the type T is used as parameter type, the compiler can automatically determine the appropriate instantiation

```
float a, b, c;
a = 3.0; b = 4.0;
c = getMax(a,b);
// as T parameter of getMax:
// template <typename T> const T& getMax(const T& a, const T& b)

double x, y, z;
x = 8.78; y = 6.45;
z = getMax(x,y);
// as T parameter of getMax:
// template <typename T> const T& getMax(const T& a, const T& b)
```

Function templates

- Template instantiation for a specific type is only done once

```
double a, b, c, d;
a = 3; b = 4;
c = getMax<double>(a, b);
d = getMax<double>(a, b+10);
```

  - In this case, the compiler only creates one instance of getMax<double>

Function templates

- But not in case of separate compilation in multiple files

```
double a, b, c;            file1.cpp
a = 3.9; b = 4.2;
c = getMax(a, b);
```

```
double x, y, z;            file2.cpp
x = 10.4; y = 2.1;
z = getMax(x, y);
```

  – production of two instances of the same getMax<double> function => code duplication

Function templates

- The following is not possible

```
double a, c; a = 3.0;
float b; b = 4.0f;
c = getMax(a,b); // compiler error: a and b different types
```

  – as a and b must have the same type T according to the signature of getMax

- Solution: multi-type in template

```
template <typename T, typename U>
T getMax(const T& a, const U& b);
```

  – but return type cannot be ref. const (internal cast)
  – careful in the choice of the return type

Templates vs. macros

```
template <typename T>
const T& getMax(const T& a, const T& b) {
   if (a < b) return b;
   else return a;
}
```

- vs.

```
#define getMax(a,b) ((a < b) ? b : a)
```

Templates vs. macros

Property / Code	Template	Macro
C++ compliance	Part of the C++ language	Not part of the C++ language
Syntax	Compilation with full syntax check	Direct text substitution without syntax checking
Type	Full semantic control with type checking	No type checking
`char * p;` `getMax(p,1000);`	Compiler error: template requires arguments of the same type	No compiler error. Result is unknown type (unreliable and not portable)
`getMax(++x,++y);`	The values are changed once (normal behavior)	Either x or y will be incremented twice (unexpected behavior)

Universiteit Utrecht

Class templates

- A type-parameterized class is not a fully defined class but a *type generator* that can be used to produce new user-defined types
- A class template is *instantiated* to particular class types by providing the missing type as parameter
  - example: STL containers

```
std::vector<int> playersLevel;
std::vector<std::string> playersName;
std::vector<Player *> podium;
```

Class templates

- A template class can have members that use template parameters as types

```
template <typename T>
class Pair {
        T values [2];
   public:
        Pair (T first, T second) {
                values[0]=first;
                values[1]=second;
        }
};
```

Class templates

- Instances of Pair template class

```
Pair<float> positionXY (10.2,-4.5);
Pair<std::string> fullName ("John","God");
Pair<Player> duel (player1,player2);
Pair<Player*> duelPtr (&player1,&player2);
```

- Function definition in class declaration

```
public:
   T getFirst() const { return values[0]; }
};
```

- Function definition outside class declaration

```
template <typename T>
T Pair<T>::getFirst() const {
   return values[0];
}
```

Template specialization

- Different implementations can be specified for any specific types

```
// Generic template class:
template <typename T> class className { ... };

// Specific template class:
template <> class className <specificType> { ... };
```

- All members have to be re-defined in the specialization
  - even those that are exactly the same
  - there is no inheritance from the generic class

Template specialization

- Example: Pair of Player
  - ‘same’ constructor is re-defined
  - new member duel specific to Pair of Player

```
template <> class Pair <Player> {
        Player values [2];
   public:
        Pair (Player first, Player second) {
                values[0]=first;
                values[1]=second;
        }
        Player duel() const {
                if (values[0].amno > values[1].amno)
                        return values[0];
                else return values[1];
        }
};
```

Universiteit Utrecht

Template parameter

- Template classes can have regular typed parameters (must be compile-time constant)

```
template <typename T, int N> class array {
        T values [N];
   public:
        void setValue (int i, T value) {values[i]=value;}
        T getValue (int i);
};

template <typename T, int N>
T array<T,N>::getValue (int i); {
   return values[i];
}
```

Template parameter

- Template classes can have regular typed parameters (must be compile-time constant)

```
int main () {
   array<float,3> positionXYZ;
   positionXYZ.setValue(1,2.0);
   // ...
   array<Player,10> readTeam;
   readTeam.setValue(0,player1);
   readTeam.setValue(5,player2);
   // ...
   cout << "First player: " << readTeam.getValue(0).getName();
   return 0;
}
```

Template parameter

- It is possible to set default values and types
  - example
    - array default template class as array of ten players

```
template <typename T=Player, int N=10> class array {
 // ...
};
```

    - used to create the following default object

```
array<> arrayOfTenPlayers;
```

    - is equivalent to

```
array<Player,10> arrayOfTenPlayers;
```

Template parameter

```
template <typename T=Player, int N=10> class array { ... };
```

```
array<float,2> a1;              // OK
array<>        a2;              // OK: array<Player,10>
array<>*       a3;              // OK: array<Player,10> *
array          a4;              // Error: need template <>
array<float>   a5;              // OK: array<float,10>
array<3>       a6;              // Error: 3 not a typename
array<3,float> a7;              // Error: both types mismatched
array<,6>      a8;              // Error: wrong syntax
```

Template generation

- Templates are compiled on demand
  - the code of a template is not compiled until the appropriate instantiation
    - not (yet) required when creating a pointer
  - then the compiler generates a function / class specifically for those arguments
  - the implementation of a template class or function must be in the same file as its declaration (header file)
    - both declaration and implementation are included in any file that uses the templates (for linker)

Template generation

- The instantiated template can be used wherever regular class names can

```
typedef array<float,10> arrayOfTenFloat;
Player getBestPlayer(const array<Player,3> & podiumPlayers);
```

- The implicit constraints (*e.g.* operators defined on T) are required only if the template becomes instantiated (at compile time)
  - And all templates are instantiated only when really needed (object created or function called)

Why use templates?

- One template class can handle different types of parameters
- Compiler generates classes for only the used types
- Templates reduce the effort on coding for different data types to a single set of code
- Testing and debugging efforts are reduced

Why not use templates?

- Some compilers still have poor support for templates
  - might result in less portable code
- Less readable error messages when errors are detected in template code
  - this can make templates difficult to debug
- Each use of a template may cause the compiler to generate extra code
  - code over-creation
- Providing template definition in the header might result in more headers being included
  - increased compile time and object file size

Lecture #8

Part II : Serialization

Serialization

- Ability to store an object into some medium and restore it at a later time
  - The medium can be any type of data storage, *e.g.* (networked) memory or hard drive
  - Essential component of games (loading and saving levels, object states, user profiles, preferences ...)
- C++ does not offer built-in serialization

Serialization

- Game Entities vs. Game Resources
  - Entities are objects that represent information about specific features in the game
    - Highly dynamic, will change during gameplay, needs to be saved
      - examples: user profile, state of other characters, *etc.*
  - Resources are parts that make up the game entities
    - Remain mostly static, do not change during gameplay
      - examples: meshes, sounds, textures, sprites, *etc.*

Serialization

- Serialization sounds trivial
  1. Go through every entity in the world, save the object to disk
  2. Then, when needed, load the data from the disk into memory again
- A few problems occur here
  – What about pointer values? If we save memory address, nothing ensures it will still be available
  – How to restore the objects? How do we know we are currently dealing with a camera or a player?
  – Only filling object with data is not enough (managers initialization, resources loading ...)

Serialization

- Expected functionalities
  - to save instances, not only type
  - to be able to choose the data to save
    - do we need to save all particles of a fireplace?
  - to allow for different saving formats
    - binary and ascii (*e.g.* release vs. debug)
  - to store to and load from different media types
  - to keep saved game size under control

Streams

- Streams are sequences of bytes that can be read from and written to
- Streams allow to abstract out the media that we use for serialization
- Standard C++ STL streams (iostream, fstream ..) can be used

Streams

```
class AbstractStream {
   public:
         virtual ~AbstractStream() {};
         virtual void reset() = 0;
         virtual int read(int bytes, void* buffer) = 0;
         virtual int write(int bytes, void* buffer) = 0;
         // ...
};

class StreamFile : public AbstractStream {
   public:
         StreamFile(const std::string& filename);
         virtual ~StreamFile();
         // ...
};

class StreamMemory : public AbstractStream {
   public:
         StreamMemory();
         virtual ~StreamMemory();
         // ...
};
```

Streams

- For convenience, we should also define read and write functions for primitive types

```
int readInt(AbstractStream & stream) {
   int n = 0;
   stream.read(sizeof(int),(void*)&n);
   return n;
}
float readFloat(AbstractStream & stream) { ... };
// ...

bool writeInt(AbstractStream & stream, int n) {
   int i = stream.write(sizeof(int),(void*)&n);
   return (i == sizeof(int));
}
bool writeFloat(AbstractStream & stream, float f) { ... };
// ...
```

Streams

- Serializing an object of user type Priest

```
bool Priest::write(AbstractStream & stream) const {
   // the parent class (Player) needs to write its attributes
   // e.g. life, position, orientation, inventory ...
   bool success = Player::write(stream);

   // write the Priest specific attributes
   success &= writeBool(stream, _minionAlive);
   success &= writeInt(stream, _numHealingRing);

   return success;
}
```

Streams

- Deserializing an object of type Priest
  - NB: order is critical!

```
void Priest::read(AbstractStream & stream) {
   // the parent class (Player) reads its attributes
   Player::read(stream);

   // read the Priest specific attributes
   _minionAlive = readBool(stream);
   _numHealingRing = readInt(stream);

}
```

Streams

- Such an approach works fine but
  - The saved data is unreadable to humans (byte by byte write/read), not practical for debugging
  - If a class changes (*e.g.* add a data member)
    - Serialization / deserialization need update
    - Previous saved streams become unreadable
- Possibility to save it in a more readable format such as XML
  - More flexible but takes more space
  - Additional file parsing required

Streams

- Human-readable (“text”) format vs. non-human-readable ("binary") format
  - Text format can be opened with a text editor to see if it looks right *etc.*
  - Binary format typically uses fewer CPU cycles
  - Text format lets you ignore programming issues like sizeof and little-endian vs. big-endian
  - Binary format lets you ignore separations between adjacent values

Serialization of objects

- Serializing the pointer (memory address) does not make much sense as the address will most certainly change
  - location not available anymore
- The object referenced is serialized instead

Serialization of objects

- Objects or pointers in serialized class

```
bool Priest::write(AbstractStream & stream) const {
   // the parent class (Player) needs to write its attributes
   // ex: life, position, orientation, inventory ...
   bool success = Player::write(stream);

   // write the Priest specific attributes
   // primitive types ...
   success &= writeBool(stream, _minionAlive);
   success &= writeInt(stream, _numHealingRing);

   // ... and user-defined types
   success &= _spellBook.write(stream);       // Object
   success &= _potionRecipes->write(stream); // Pointer

   return success;
}
```

Serialization of objects

- Objects or pointers in serialized class

```
void Priest::read(AbstractStream & stream) {
   // the parent class (Player) reads its attributes
   Player::read(stream);

   // read the Priest specific attributes
   // primitive types ...
   _minionAlive = readBool(stream);
   _numHealingRing = readInt(stream);

   // ... and user-defined types
   _spellBook.read(stream);            // Object
   _potionRecipes->read(stream);       // Pointer
}
```

Serialization of objects

- Pointers to object have to be initialized before the call `pointer->read(stream);`
  - usually in constructor but not necessarily
  - or object destruction called between write/read
    - by an explicit delete call
    - by exiting and starting up the program again

```
void Priest::read(AbstractStream & stream) {
// ...
if (_potionRecipes == NULL) _potionRecipes = new Recipes();
_potionRecipes->read(stream);
```

- but constructor parameters not necessarily known, they have to be accessible *e.g.* as data members

Serialization of objects

- Use of read and write

```
Priest * player1 = new Priest();

// ... setting of Player attributes (life etc.)
// ... setting of Priest attributes (_numHealingRing etc.)

StreamFile file (“savedFile.txt”);
player1->write(file);

// after update or delete of player1
// or ‘load saved game’ routine ...

Priest * player2 = new Priest();
player2->read(file);
```

Serialization

- But objects can contain pointers created by another class (not the owner)
  - we do not want to create a new instance
- As memory address is unique for each object
  - we also store memory location with each object we save
  - then, we can construct a translation table for going from the old memory address to the new one (called only once)
  - translation should happen after all entities have been loaded

Serialization

- Class Serializable defining this behavior for serializable objects

```
class Serializable {
  public:
      virtual ~Serializable() {};
      virtual bool write(AbstractStream& stream) const = 0;
      virtual void read(AbstractStream& stream) = 0;
      virtual void fixup() = 0;
};
```

Serialization

- Add address in translation table at reading
  - Example: if Priest is owner of _potionRecipes

```
void Priest::read(AbstractStream & stream) {
   // ...
   if (_potionRecipes == NULL) _potionRecipes = new Recipes();
   void* pOldAddress = (void*)ReadInt(stream);
   AddressTranslator::AddAddress(pOldAddress,_potionRecipes);
   _potionRecipes->read(stream);
   // ...
}
```

Serialization

- Now we only need to implement the *fixup* method for each object pointing to _potionRecipes that is not its owner
  – this method only works for pointers that were explicitly saved

```
void Enemy::fixup() {
   _priestPotionRecipes = (Recipes*)
    AddressTranslator::TranslateAddress(_priestPotionRecipes);
}
```

- In case the translation failed, the user should be warned (*e.g.* exception thrown)

Serialization

- The Boost library provides a serialization mechanism (see documentation)

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
std::stringstream ss;

class Priest : public Player {
   // ...
   friend class boost::serialization::access;
   template <typename Archive>
   void serialize(Archive &ar, Priest &p, const unsigned int ver) {
        ar & p.dataMember;
   }
}
```

Serialization

- The Boost library provides a serialization mechanism (see documentation)

```
void save() {
  boost::archive::text_oarchive oa(ostream);
  Priest p; // or Priest *
  oa << p;
}

void load() {
  boost::archive::text_iarchive ia(istream);
  Priest p; // or Priest *
  ia >> p;
}
```

End of lecture #8

Next lecture

Interfacing