Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #3

Advanced OO, STL, compilation and programming

Inheritance

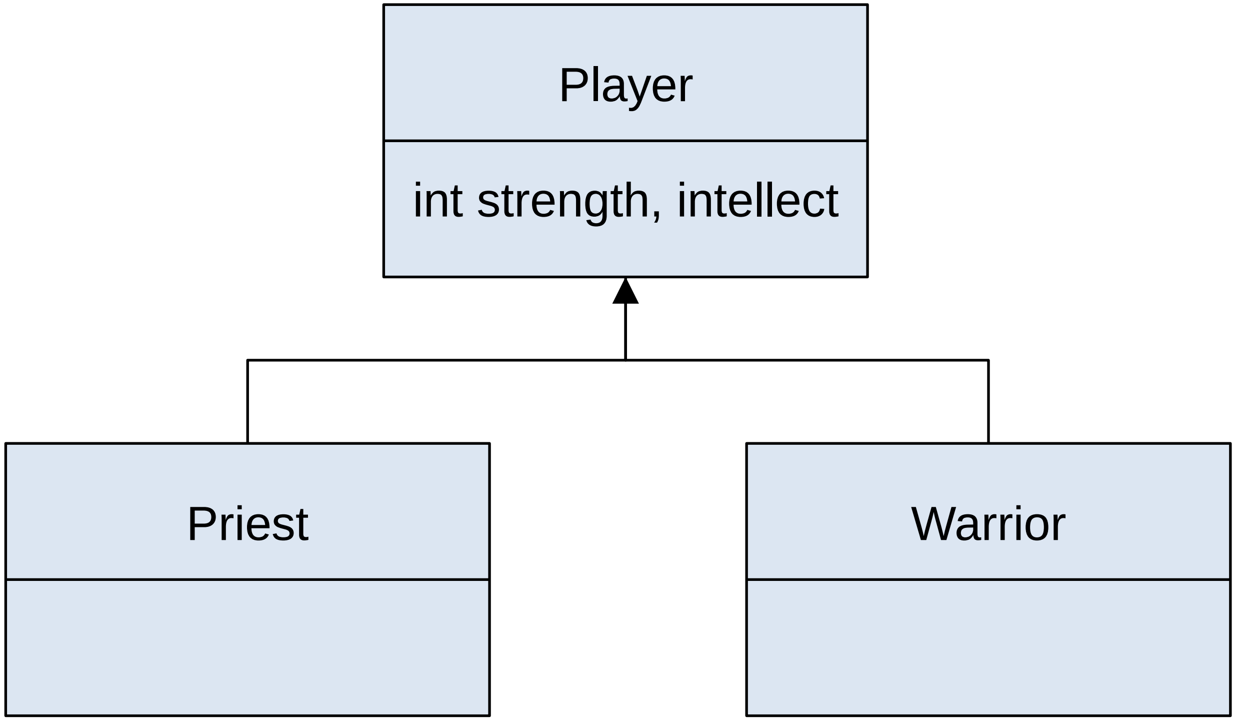
- Allows to create classes which are derived from other classes
  - automatically include some of its parent's members *(plus its own)*

```
class derived_class: access_specifier base_class {
    // class declaration
};
```

  - access specifier (public, protected and private) represents the most accessible level for the members inherited

Inheritance

- Example



Inheritance

- Class Player

```
class Player {                                                    Player.h
   protected:
         int strength, intellect;
   public:
         int level;
         void setAttributes(const int, const int);
};
```

```
                                                                Player.cpp
Player::setAttributes(const int newStrength, const int newIntellect) {
   strength = newStrength;
   intellect = newIntellect;
};
```

Inheritance

- Class Priest

```
class Priest: public Player {                                     Priest.h
   public:
        int castSpell () const;
        int meleeAttack () const;
};
```

```
                                                                Priest.cpp
int Priest::castSpell() const {
   return (intellect * level);
};

int Priest::meleeAttack() const {
   if (level > 10) return (strength * level);
   else return 1;
};
```

Inheritance

- Class Warrior

Warrior.h

```
class Warrior: public Player {
   public:
         int castSpell () const;
         int meleeAttack () const;
};
```

Warrior.cpp

```
int Warrior::castSpell() const {
   if (level > 10) return (intellect * level);
   else return 1;
};

int Warrior::meleeAttack() const {
   return (strength * level);
};
```

Inheritance

- Main program

Main.cpp

```
int main () {
   Priest player1;
   Warrior player2;

   player1.level = 4;
   player2.level = 11;

   player1.setAttributes(2,20);
   player2.setAttributes(40,12);

   cout << player1.castSpell() << “ ” << player1.meleeAttack() << endl;
   cout << player2.castSpell() << “ ” << player2.meleeAttack() << endl;

   return 0;
};
```

Inheritance

- What is inherited from the base class?
  - everything except constructor, destructor, operator= and friends
- Calling the base constructor from the derived class
  - syntax

```
derived_constructor (parameters) :
   base_constr(parameters) {
   // body of derived class constructor
}
```

Inheritance

- Class Player

```
class Player {                                                          Player.h
   protected:
        int level;
   public:
        Player ();
        Player (int);
};
```

```
                                                                    Player.cpp
Player::Player() {
   level = 0;
   cout << “Player newbie! ”;
};

Player::Player(int newLevel) {
   level = newLevel;
   cout << “Player created with level ” << level << “. ”;
};
```

Inheritance

- Class Priest

Priest.h

```
class Priest : public Player {
   public:
        Priest (int);
};
```

Priest.cpp

```
Priest::Priest(int newLevel) {
   cout << “Priest (lvl ” << level << “ )” << endl;
};
```

Inheritance

- Class Warrior

Warrior.h

```
class Warrior : public Player {
   public:
         Warrior(int);
};
```

Warrior.cpp

```
Warrior::Warrior(int newLevel) : Player (newLevel) {
   cout << “Warrior (lvl ” << level << “ )” << endl;
};
```

Inheritance

- Main program

```
                                                                   Main.cpp
int main () {
   Priest player1 (3);
   Warrior player2 (5);
   return 0;
};
```

- Output

```
Player newbie! Priest (lvl 0)
Player created with level 5. Warrior (lvl 5)
```

- Because

```
Priest(int newLevel) // nothing specified: calls default parent
Warrior(int newLevel) : Player (newLevel) // calls specific constructor
```

Virtual members

- Imagine we want

```
int main () {
   Player * player1 = createRandomPlayer(); // Priest or Warrior
   Player * player2 = createRandomPlayer();
   cout << “Damage done by player1 : ” << player1->castSpell()  << endl;
   cout << “Damage done by player2 : ” << player2->castSpell()  << endl;
   return 0;
};
```

- We should add castSpell() function to Player
- But Priest and Warrior classes use different implementations of the castSpell() function
  - Virtual members

Virtual members

- Class Player

Player.h

```
class Player {
   protected:
        int strength, intellect;
   public:
        int level;
        void setAttributes(const int, const int);
        virtual int castSpell() const;
        virtual int meleeAttack() const;
};
```

Player.cpp

```
int Player::castSpell() const {
   return 0;
};
int Player::meleeAttack() const {
   return 0;
};
```

Virtual members

- Main program

```
                                                                 Main.cpp
int main () {
    Player * player1 = new Priest();
    Player * player2 = new Warrior();
    Player * player3 = new Player();

    player1->level = 1;
    player2->level = 1;
    player3->level = 1;

    player1->setAttributes(10,20);
    player2->setAttributes(10,20);
    player3->setAttributes(10,20);

    cout << player1->castSpell() << endl;
    cout << player2->castSpell() << endl;
    cout << player3->castSpell() << endl;

    delete player1; delete player2; delete player3;

    return 0;
};
```

Virtual members

- Resulting output

```
20
1
0
```

- If castSpell() was not declared virtual

```
0
0
0
```

  - because they are created as Player instances
- The effect of automatically calling the method from the derived class is called **polymorphism**
- If a function could be overridden, it should be declared as virtual
  - induce a small performance overhead (lookup table)

Abstract base classes

- In abstract base classes virtual member functions do not need implementation at all
  - by appending = 0 (equal zero) to the declaration
  - called pure virtual function

```
virtual int castSpell() const = 0;
virtual int meleeAttack () const = 0;
```

- A class containing at least one pure virtual function is called abstract base class
  - instances of an abstract base class are impossible
  - but pointers to it can be created
  - and pure virtual functions can be called from the abstract base class

Abstract base classes

Player.h

```
class Player {
    protected:
           int strength, intellect;
    public:
           int level;
           void setAttributes(const int, const int);
           virtual int castSpell() const = 0;
           virtual int meleeAttack () const = 0;
           int bestAttack() { return max(this->castSpell(),this->meleeAttack()); }
};
```

Main.cpp

```
int main () {
    Player * player1 = new Priest();  // Player player1; forbidden
    Player * player2 = new Warrior(); // Player * player2 = new Player(); forbidden

    player1->level = 1; player2->level = 1;

    player1->setAttributes(10,20); player2->setAttributes(10,20);

    cout << player1->bestAttack() << “ ” << player2->bestAttack() << endl;

    delete player1; delete player2;
    return 0;
};
```

Multiple inheritance

- C++ allows a class to inherit members from more than one class
  - by simply separating the different base class names with commas in the derived class declaration

```
class derivedClass :
    access_specifier baseClass1,
    access_specifier baseClass2, ... {
    ...
};
```

- Multiple inheritance is often used to inherit from multiple abstract base classes
- But there are some problems
  - Ambiguity
  - Topography
  - and more

Multiple inheritance

- Problem 1: ambiguity
  - base classes having the same member

```
...
if (derivedClass->CommonMember()) { // Compiler error!
...
```

  - solution by prefixing the class name
    - in derived class

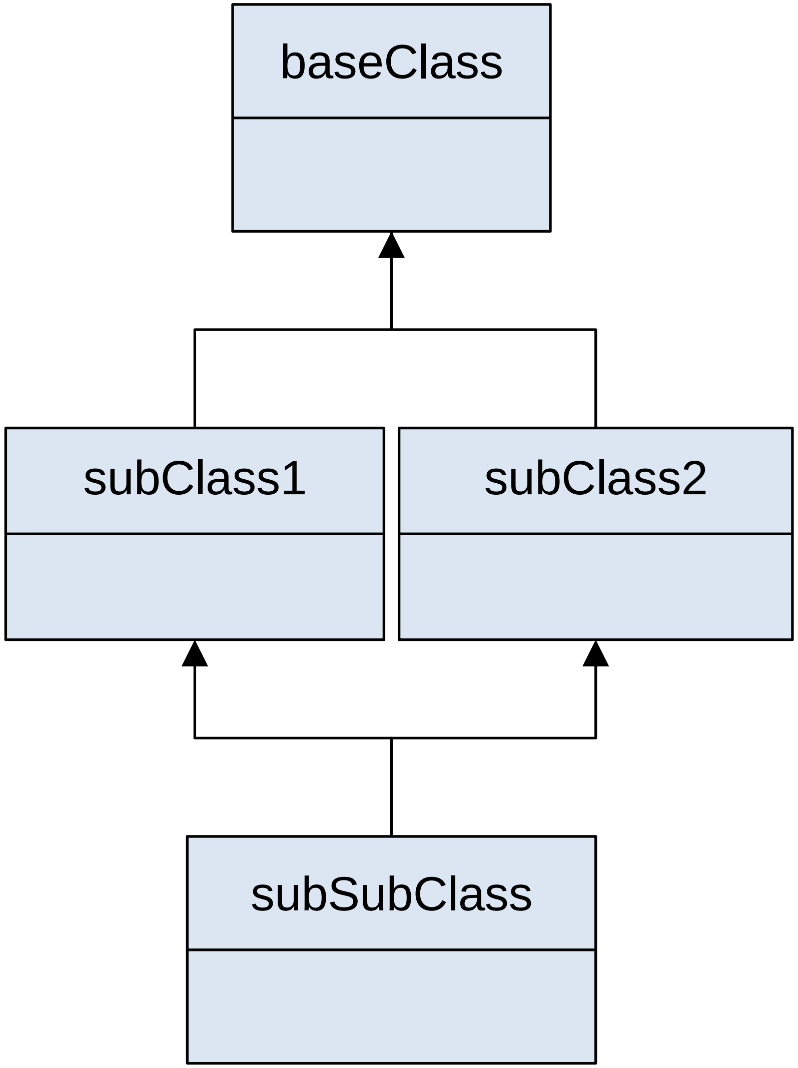
```
...
if (baseClass1::CommonMember()) {
...
```

    - outside derived class (required to know the parents) ⚠

```
...
if (instance.baseClass1::CommonMember()) {
...
```

Multiple inheritance

- Problem 2: topography
  - Diamond of Death (DoD)
  - Content of baseClass appears twice in subSubClass



Multiple inheritance

- Problem 2: topography
  - Ambiguity issues
    - Using members of baseClass
    - Creating baseClass * b = new subSubClass()
  - Solutions
    - Inheritance path and intermediate cast everywhere (see problem 1)
    - Virtual inheritance (space and performance cost)
      - class subClass1 : public virtual baseClass
      - class subClass2 : public virtual baseClass

Type casting

- Implicit and explicit conversion

```
int lvlInt = 1;
float lvlFloat1 = lvlInt;         // implicit conversion, compiler warning
float lvlFloat2 = (float) lvlInt; // explicit conversion (c-like)
float lvlFloat3 = float (lvlInt); // another explicit syntax (functional)
```

- ⚠ Careful with conversion between pointers

```
Team t;
Player * ptrPlayer;
ptrPlayer = (Player*) &t;
cout << ptrPlayer->level; // read data member level on Team memory space
```

  – No compiler error but wrong memory state

Type casting

- C++ has four specific casting operators
  - dynamic_cast
  - reinterpret_cast
  - static_cast
  - const_cast
- Syntax is

```
cast_type <data_type> (expression);
```

- Example

```
dynamic_cast <float *> (positionX);
```

Type casting

- dynamic_cast
  - used only with pointers and references
  - checks the compatibility at run-time
  - ensures that the result of the type conversion is a valid complete object of the requested class
  - always successful when casting a class to one of its base classes
  - result
    - success: returns a new pointer or reference
    - fail: returns NULL or throws bad_cast exception

```
class CBase { };
class CDerived: public CBase { };

CBase b; CBase* pb;
CDerived d; CDerived* pd;

pb = dynamic_cast<CBase*>(&d);     // OK: derived-to-base
pd = dynamic_cast<CDerived*>(&b);  // wrong: base-to-derived
```

Type casting

- static_cast
  - conversions between pointers to related classes
    - from the derived class to one of its bases
    - from a base class to one of its derived classes
  - no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type
    - the overhead of the type-safety checks of dynamic cast is avoided

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
// valid, but b points to an incomplete object of the class and
// could lead to runtime errors
```

Type casting

- reinterpret_cast
  - converts any pointer type to any other pointer type, even of unrelated classes
    - binary copy of the value from one pointer to the other
    - neither the content pointed nor the pointer type itself is checked

- ⚠ Use sparingly and only when other types of casts are not enough

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
// valid but pointless as B points to an object of an
// incompatible class
```

Type casting

- const_cast
  - to manipulate the constness of an object
    - to set and to remove
  - only use if absolutely necessary
  - if you need it, you probably have to rethink the design of your class

```
const char * c = "text to print";
printAString(const_cast<char *>(c));
// Does printAString really need a non-const object?
```

typeid operator

- C++ allows to check the type of an expression with the typeid operator

```
typeid (expression);
```

  – returns a reference to a constant object of type type_info
    - can be compared with another one
    - can serve to obtain the data type or class name

```
#include <typeinfo>
...
Player * player1 = new Warrior();
Player * player2 = new Player();
cout << “player1 is: ” << typeid(player1).name(); // Player *
cout << “*player1 is: ” << typeid(*player1).name(); // Warrior
cout << “*player2 is: ” << typeid(*player2).name(); // Player
```

Operator overload

- Operators (+,&,--,<<,...) manipulating objects can also be changed (not the primitive type)
- Same as regular function using the syntax

```
type operator operator_symbol (parameters) {
...
}
```

- Example

```
class Player {
    public:
        int level;
        bool operator > (const Player& player) const {
                return (level > player.level);
        }
        friend ostream& operator << (ostream& os, const Player& player);
}
```

STL

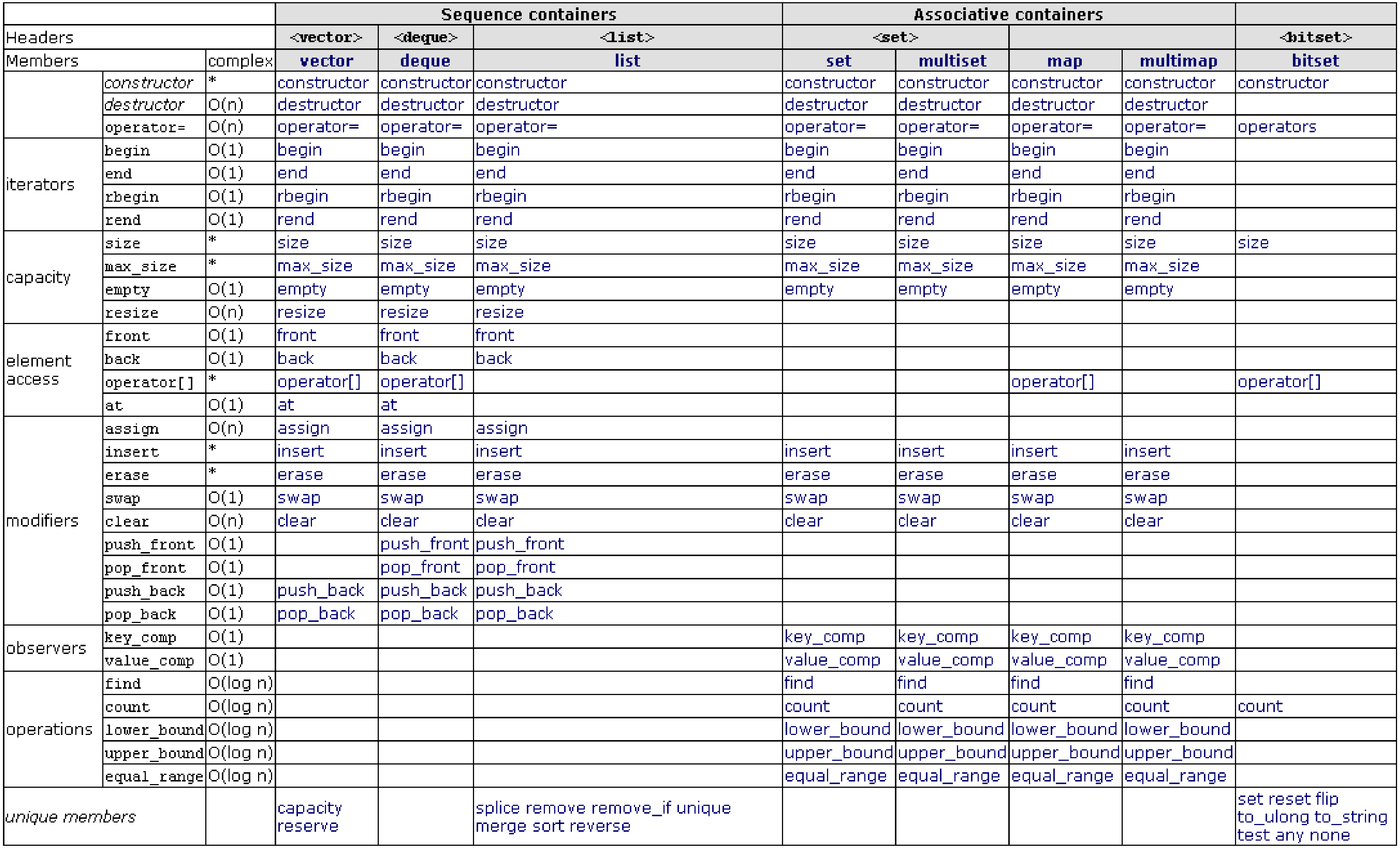
- “Standard Template Library”
  - containers, iterators and algorithms
  - implemented as class template (more later)
- The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators
  - Improved array implementation for C++
  - Automatic memory management when adding and deleting elements
- The algorithms library is a collection of functions especially designed to be used on ranges of elements

STL

- Part of the ANSI/ISO C++ since 1994
  - – everything inside “std” namespace
  - – provides useful data structures and algorithms
  - – easy integration to your classes (templates)
  - – robust, optimized, stable and widely used

STL

- Mainly two types of containers
  - Sequence containers: elements are stored in a specific order
  - Associative containers: order of elements is not preserved
- Iterators allow to access the different elements
  - begin() returns the iterator to the first element
  - end() returns the iterator *past* the last element
- STL contains a set of standard algorithms that can be applied to containers and iterators
  - Finding elements, copying, reversing, sorting, etc.



			Sequence containers			Associative containers				
Headers			<vector>	<deque>	<list>	<set>				<bitset>
Members		complex	**vector**	**deque**	**list**	**set**	**multiset**	**map**	**multimap**	**bitset**
	constructor	*	constructor	constructor	constructor	constructor	constructor	constructor	constructor	constructor
	destructor	O(n)	destructor	destructor	destructor	destructor	destructor	destructor	destructor	
	operator=	O(n)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operators
iterators	begin	O(1)	begin	begin	begin	begin	begin	begin	begin	
	end	O(1)	end	end	end	end	end	end	end	
	rbegin	O(1)	rbegin	rbegin	rbegin	rbegin	rbegin	rbegin	rbegin	
	rend	O(1)	rend	rend	rend	rend	rend	rend	rend	
capacity	size	*	size	size	size	size	size	size	size	size
	max_size	*	max_size	max_size	max_size	max_size	max_size	max_size	max_size	
	empty	O(1)	empty	empty	empty	empty	empty	empty	empty	
	resize	O(n)	resize	resize	resize					
element access	front	O(1)	front	front	front					
	back	O(1)	back	back	back					
	operator[]	*	operator[]	operator[]				operator[]		operator[]
	at	O(1)	at	at						
modifiers	assign	O(n)	assign	assign	assign					
	insert	*	insert	insert	insert	insert	insert	insert	insert	
	erase	*	erase	erase	erase	erase	erase	erase	erase	
	swap	O(1)	swap	swap	swap	swap	swap	swap	swap	
	clear	O(n)	clear	clear	clear	clear	clear	clear	clear	
	push_front	O(1)		push_front	push_front					
	pop_front	O(1)		pop_front	pop_front					
	push_back	O(1)	push_back	push_back	push_back					
	pop_back	O(1)	pop_back	pop_back	pop_back					
observers	key_comp	O(1)				key_comp	key_comp	key_comp	key_comp	
	value_comp	O(1)				value_comp	value_comp	value_comp	value_comp	
operations	find	O(log n)				find	find	find	find	
	count	O(log n)				count	count	count	count	count
	lower_bound	O(log n)				lower_bound	lower_bound	lower_bound	lower_bound	
	upper_bound	O(log n)				upper_bound	upper_bound	upper_bound	upper_bound	
	equal_range	O(log n)				equal_range	equal_range	equal_range	equal_range	
unique members			capacity reserve		splice remove remove_if unique merge sort reverse					set reset flip to_ulong to_string test any none

Universiteit Utrecht

source: cplusplus.com

STL vector

- Most commonly used container
- Random element access
- Insertion and deletion
  - efficient at the end, less otherwise
  - element can be added/deleted everywhere
- Always better than C arrays

STL vector

```
#include <vector>

int main() {
    std::vector<int> PlayerPerTeam;
    std::vector<float> AverageKillsPerPlayer;
    PlayerPerTeam.push_back(2);
    PlayerPerTeam.push_back(1);
    AverageKillsPerPlayer.push_back(10.3);
    AverageKillsPerPlayer.push_back(8.4);
    AverageKillsPerPlayer.push_back(15.9);
    std::cout << “Game has ” << PlayerPerTeam.size() << “ team(s).” << endl;
    std::cout << “Team 1 has ” << PlayerPerTeam[0] << “ player(s).” << endl;
    std::cout << “Player 3 has ” << AverageKillsPerPlayer[2] << “ AK.” << endl;
    PlayerPerTeam.clear();
    AverageKillsPerPlayer.clear();
    return 0;
}
```

STL deque

- Double-ended queue
- Fast insertion/deletion at the beginning as well as the end of the sequence
- Use several memory blocks
- Useful for FIFO-like structures (buffers)
- Do not use in small memory reserve and expensive memory usage programs

STL list

- No random access to elements
- Double-linked list of elements (each element has two pointers, one for each neighbor)
  - No penalty for inserting/deleting in the middle
  - Costly to transverse the list (no contiguous in memory)
  - Algorithms efficient as no copy (pointer update)
- Use when you need to apply algorithms and add/delete operations on all elements

STL set/multiset

- Mathematical set
  - not ordered elements
  - no duplicate in set, allowed in multiset
  - operator < between elements should be defined
- Implemented as binary search tree
  - cost of $O(\ln n)$ for search and comparison
  - useful only for large structures to keep track of processing

STL map/multimap

- Key-based set (instead of value)
  - can be seen as array with index as object
  - provides the direct access [] operator ($O(\ln n)$)
- Useful for non index-based look-up table or dictionary
- Create default element if access out of boundary
- Same implementation and performance as set

STL iterator

- Several types of iterators
  - const and non-const
  - forward, bidirectional and direct access
- Iterators have operators (==, != ,++, ...)
- Accessing the elements with * operator

```
vector<string> PlayerNames;
PlayerNames.push_back(“John”); ...

vector<string>::iterator it;
for (it = PlayerNames.begin(); it != PlayerNames.end(); ++it) {
   cout << “Player name : ” << *it << endl;
}
```

Building the code

- Preprocessor
  - – Evaluate macros and includes
- Compiler
  - – Create object files (.obj) from C++ code (h+cpp)
- Linker
  - – Resolve the links between different parts of code, for example include libraries
  - – Create
    - executable (.exe on Windows) if main program
    - library (.dll/.lib on Windows) otherwise

Preprocessor

- Preprocessor directives
  - lines included in the code that are not program statements but directives for the preprocessor
  - preceded by a hash sign (#)
  - executed before the compilation of code begins
- C++ has several types of directives
  - Macro definitions *(#define, #undef)*
  - Conditional inclusions *(#ifdef, #ifndef, #if, #endif, #else, #elif)*
  - Error directive *(#error)*
  - Source file inclusion *(#include)*
  - and more...

Preprocessor: macro

- Tells the preprocessor to do a text replace in the code
  - useful for constants used everywhere

```
#define identifier replacement
...
#undef identifier
```

  - useful for context-independent short functions

```
#define max(x,y) x>y?x:y
```

- Two special operators (# and ##)
  - the operator # replaces a parameter by a string
  - the operator ## concatenates two parameters

Preprocessor: condition

- Allows to include or discard part of the code of a program if a condition is met
  - To use at the beginning of a class declaration to prevent multiple loading
  - Useful to write platform independent and modular programs

```
#ifndef CLASSNAME_H_
#define CLASSNAME_H_

class className {
...
};

#endif
```

Preprocessor: error

- Aborts the compilation process when it is found
  - generates a compilation error that can be specified as its parameter
  - useful to raise problems during checking environment, compatibility...

```
#ifndef __cplusplus
#error A C++ compiler is required!
#endif
```

Preprocessor: inclusion

- Replaces the directive by the entire content of the specified file

```
#include “localfile”
   // 1st search in working directory then standard header directory

#include <standard_library>
   // search directly in standard header directory
   // platform / environment dependent
```

Programming in C++

- Organizing the code
  - use a directory structure to group related classes

```
#include <iostream>
#include <string>

#include “GameEngine/Graphics/Renderer.h”
#include “GameEngine/Graphics/3DObject.h”
#include “GameEngine/Network/SendData.h”

#include “Character/Team.h”
#include “Character/Player.h”
#include “Character/AI/PathPlanning.h”
#include “Character/AI/GroupBehavior.h”
```

Programming in C++

- namespace
  - allows to group entities like classes, objects and functions under a name

```
namespace identifier {
    entities
}
```

- When developing a toolkit / library, use a single namespace for all classes
  - Usage:

```
using namespace identifier;
```

- Only put using statements in definitions (.cpp) and not in headers (.h)

Programming in C++

- Comments
  - serve to clarify code and provide additional information to users
- Provide comments for
  - class descriptions
  - all constructors/methods and the destructor
  - all functions with parameters, in/out and return values
  - description of class attributes

Programming in C++

- Hungarian notation
  - Invented by Charles Simonyi from Microsoft
  - Helps as a reminder of the type in the name
  - Extended to include scope information
  - Example: `static std::string * s_pName;`

scope prefix	description
m_	class member variable
s_	class static variable
g_	global variable

type prefix	description
b	boolean variable
i	integer variable
f	float variable
p	pointer variable

Remarks about const

- Use const instead of #define
  - type safe compiler
  - available in the debugger
- Useful for non modifiable function
  - control over updating methods
  - ⚠ const function cannot call non-const functions
  - the mutable keyword on data member
    - to allow data member modification from a const function

Remarks about references

- Commonly used in function parameters, but also as returned object
  - no local copy
- References vs. Pointers
  - more control as never NULL and fixed owner
  - but impossible to change ownership and object pointed
  - NULL can be useful
  - no arithmetic in references

More tips

- Give explicit member name (not a, b, hfyw)
- Indent the code to indentify the scopes
- Create functions instead of copy/paste
- Use inheritance and containment
- Make the class as simple as possible
- Double check destruction of heap variables
- Make the program working, then optimize

End of lecture #3

Next lecture

Game engine architecture