Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #11

Optimization and Advanced STL

Memory

- Memory consists of a number of slots
  - each slot has a fixed size (8 bits or 1 byte)
  - each slot has a unique identifier in a “list”
  - we can read and write bytes from each slot by referring to the identifier
- Objects and variables takes up a number of slots in the memory
  - for example, float = 4 bytes
  - the location of objects in memory is determined by the identifier of the first slot they occupy
- A memory address = identifier of a slot in the memory

Heap vs. Stack

- Stack
  - parameters
  - return address (where to begin execution when function exits)
  - pointer to current instance (this)
  - return value (where to put the return value)
  - local variables
- Function calls
  - function call - push stack frame
  - function exit - pop stack frame
- Storage objects on stack are de-allocated when the stack frame is popped

Heap vs. Stack

- Heap
  - is a list of free space: freelist
  - on allocation: memory manager finds space and marks it as used changing the freelist
  - on de-allocation: memory manager marks space as free changing the freelist
  - memory fragmentation: memory fragments into small blocks over lifetime of program
  - garbage collection: combine fragments, possibly moving objects
    - HeapCompact in <windows.h>

Function calls

- Functions form the basic components of programming
- C++ offers different types of functions with a specific overhead
  - Normal (global) functions
  - Static functions
  - Non-virtual member functions
  - Virtual member functions

Function calls

- Normal (global) functions

```
int amnoleft = Shoot(int amno, Enemy e);
```

  - cost: jumping to a different memory location (almost negligible)
- General rule: forget about performance overhead and use functions whenever it seems logical
  - Implementation in smaller sub-steps
  - More readable code
  - Easier to maintain code
  - Encourages reuse (same sub-steps can be reused in other problems)

Function calls

- Static functions

```
static int Player::Shoot(int amno, Enemy e) { /* ... */ }
// ...
int amnoLeft = player.Shoot(10,e1);
```

  - Static calls are handled in a similar way as global function calls by the compiler
  - Therefore the same (negligible) performance overhead as global function calls
  - Provides a way to group related functions under a class for better understanding of code

Function calls

- Non-virtual member functions

```
int Player::Shoot(int amno, Enemy e) { /* ... */ }
// ...
int amnoLeft = player.Shoot(10,e1);
```

  - Functions associated with a particular instance of a class
  - Function address is determined at link time, but type of object is known at compile time

Function calls

- Non-virtual member functions
  – Implemented by having a hidden 'this' parameter that points to the object being called

```
Player player;
int amnoLeft = __PlayerClass__Shoot(&player,10,e1);
```

  – Additional cost to regular function is just passing the extra instance pointer as a parameter
    - No real final difference in performance

Function calls

- Virtual member functions
  - Have potential to be expensive
  - Occur when we invoke a method on polymorphic objects
    - example

```
Player * player1 = new Warrior();
Player * player2 = new Priest();
player1->castSpell();
player2->meleeAttack();
```

Function calls

- Virtual member functions
  - Performance overhead for dereferencing the look-up table (called v-table)
  - This cost might become an issue if the function is often called (*e.g.* 1000 times per frame)
  - But the depth of the inheritance hierarchy has no influence on performance
    - each class has its own look-up table
  - With multiple inheritance
    - Append look-up table of base classes
    - Using a large multiple inheritance tree could result in a large lookup table

Function calls

- Virtual member functions
  - When a virtual function is invoked directly on the object, same performance hit as normal function call
    - Normal hit

```
Player player1;
player1.castSpell();
```

    - Virtual look-up table overhead

```
Player * player1 = new Player();
player1->castSpell();
```

Function inline

```
class Player {
   public:
        bool isMoving() const;
   private:
        bool moving;
        float lastDisplacement;
};
```

```
bool Player::isMoving() const {
   return moving;
}
```

- A non-virtual function overhead occurs every time the function isMoving() is called
  - quite a pity for a so trivial function

Function inline

- The additional performance cost can be avoided by making the member public
  - But the user needs to know about the implementation
  - And it's getting worse if the function uses more members

```
bool Player::isMoving() const {
   return (moving || lastDisplacement < 10);
}
```

    - both members need to be public
    - and the whole code needs to be updated!

Function inline

- Solution: make it an inline function

```
class Player {
   public:
        bool isMoving() const;
   private:
        bool moving;
        float lastDisplacement;
};

inline bool Player::isMoving() const {
   return moving;
}
```

```
class Player {
   public:
        inline bool isMoving() const {
                return moving;
        };
   private:
        bool moving;
        float lastDisplacement;
};
```

Inline functions must be declared in the header file, not the body file

Function inline

- The compiler replaces the function with the code in the executable
  - No function calling overhead, so more efficient execution
- So why not always use inline?
  - Executable size increases because of code duplication
    - Consumes more memory
    - Poor use of code cache, lower program performance
  - Some includes move to the header files, resulting in longer compilation times
  - You lose the declaration/implementation separation
- Avoid using inline when developing, but add inline later when you are optimizing/finalizing your code
- Useful for small, frequently used methods like get-set methods

STL

- Containers
  - Sequence containers (vector, deque, list)
  - Associative containers (set, map)
- Iterators
- Functors
- Algorithms

Functors

- To pass functions as parameters

```
bool lessThanAbsoluteValue(float a, float b) {
   return abs(a) < abs(b);
}

bool (*mycomparison)(float, float);
mycomparison = &lessAbsoluteValue;

// Now we can pass mycomparison to any function that takes
// function pointers of that type
```

Functors

- A function that uses the mycomparison pointer might look like this

```
void sort(bool(*cmpfunc)(float, float),std::vector<float>) {
    // ...
}
```

  – but quite difficult to read

Functors

- STL provides function objects: the functors
- A functor is simply any object of a class that provides a definition for the operator ()
- The class is a normal class with constructor, destructor, data and function members

Functors

- Example of a simple functor

```
class lessThanAbsoluteValue {
   public:
        bool operator()(float a, float b) const {
               return abs(a) < abs(b);
        }
};

// Using the sort function of STL:
sort(unSortedData.begin(),
     unSortedData.end(),
     lessThanAbsoluteValue()); // lessThanAbsoluteValue instance
```

Functors

- Multiple functors can be defined with different signatures

```
class lessThanAbsoluteValue {
   public:
       bool operator()(float a, float b) const {
              return abs(a) < abs(b);
       }
       bool operator()(int a, int b) const {
              return abs(a) < abs(b);
       }
};
// ...
sort(unSortedData.begin(),unSortedData.end(),
     lessThanAbsoluteValue());
// uses the operator with type of unSortedData elements
```

Functors

- Often, we would like to pass a member function
  - Passing a pointer to this function does not work!

```
class Player {
private:
    int _life;
public:
   Player(int life) : _life (life) {}
   void PrintLife() { std::cout << _life << " " ; }
};
// ...
std::vector< Player > team ;
team.push_back(Player(9));
team.push_back(Player(3));
std::for_each(team.begin(),team.end(),&Player::PrintLife);
// Does not compile!
```

Functors

- But we can write a wrapper functor that calls the member function

```
class Player {
private:
    int _life;
public:
   Player() : _life (0) {}
   Player(int life) : _life (life) {}
   void PrintLife() { std::cout << _life << " " ; }
   void operator() (Player p) {p.PrintLife();}
};

};
// ...
std::for_each(team.begin(),team.end(),Player());
```

Functors

- Or STL also provides functor adaptors
  - mem_fun for member functions through a pointer
  - mem_fun_ref for member functions through an object or a reference
  - ptr_fun for global functions through a function pointer
- Here mem_fun_ref as vector of Player

```
std::for_each(team.begin(),team.end(),
              std::mem_fun_ref(&Player::PrintLife));
```

Functors

- Examples

```
vector<Player> team;
// assuming: bool Player::CompPlayer(Player);
sort(team.begin(),team.end(),mem_fun_ref(&Player::CompPlayer));
```

```
vector<Player *> team;
// assuming: bool Player::CompPlayer(Player *);
sort(team.begin(),team.end(),mem_fun(&Player::CompPlayer));
```

```
vector<Player> team; // resp. Player *
// assuming: bool CompPlayer(Player, Player); // resp. Player *
sort(team.begin(),team.end(),ptr_fun(CompPlayer));
```

Algorithms

- STL defines many different algorithms
- Functions of common complex operations on containers
- Four categories of algorithms
  - Non-modifying operations
  - Modifying operations
  - Sorting and operations on sorted range
    - Binary search
    - Merge
  - Min/max

Non-modifying container operations

operation	description
for_each	apply function to range
find	find element in range
find_if	find element in range satisfying condition
find_end	find last subsequence in range
find_first_of	find element from set in range
adjacent_find	find equal adjacent elements in range
count	count appearances of element in range
count_if	return number of elements in range satisfying condition
mismatch	return first position where two ranges differ
equal	test whether the elements in two ranges are equal
search	find subsequence in range
search_n	find succession of equal elements in range

Non-modifying container operations

- Example: find
  - Iterates through the elements in a container and looks for specific element

```
vector<string> playersName;
// names construction ...

// check if new name does not already exist
if (find(playersName.begin(), playersName.end(), newName) !=
    playersName.end()) {
   // name already exists ...
}
else {
   // name does not exist yet ...
}
```

Non-modifying container operations

- Example: for_each
  - Executes a function on a range of elements
    - Very different from a regular for statement

```
vector<string> playersName;
// names construction ...

// print the names, assuming: void printName(string s) { ... }
for_each(playersName.begin(), playersName.end(), printName);
```

```
void update (Player& player) {
  player.update();
}
// update all players
for_each(players.begin(), players.end(), update);
```

Universiteit Utrecht

Non-modifying container operations

- Example: count
  - Counts the amount of elements matching a particular element

```
// count players named "God"
int nbGod = count(playersName.begin(), playersName.end(), "God");
```

```
Player player1 ("Azimux",10,4.5,6.7);

// Creation of vector<Player> players ...
// assuming: bool Player::operator == (const Player& p) const

// count players similar to player1
int nbP1 = count(players.begin(), players.end(), player1);
```

Non-modifying container operations

- Example: count_if
  - Same as count but evaluates a predicate instead of using operator ==

```
class isSamePlayer {
public:
   bool operator()(const Player& player) const {
        return (player.getName().compare("God") == 0);
   }
};

// Creation of vector<Player> players ...

// count players with name "God"
int nbGod = count_if(players.begin(), players.end(),
   isSamePlayer());
```

Non-modifying container operations

- Example: count_if

```
bool isSamePlayer (const Player& player1, const Player& player2) {
   return (player1.getName().compare(player2.getName()) == 0);
}

Player myPlayer (“Azimux”,10,4.5,6.7);

// Creation of vector<Player> players ...

// count players similar to myPlayer
int nbP1 = count_if(players.begin(), players.end(),
   bind1st(ptr_fun(&isSamePlayer),myPlayer));
```

Function adaptors

- As count_if is expecting a unary function to compare the elements, we need to convert the binary function:

```
bool isSamePlayer (const Player& player1, const Player& player2)
```

into a unary function applied on our particular element

  - here the first parameter player1 will always be myPlayer

```
bind1st(ptr_fun(&isSamePlayer),myPlayer)
// bind2nd is also possible
```

Modifying container operations

operation	description
copy	copy range of elements
copy_backward	copy range of elements backwards (start copying from end)
swap	exchange values of two objects
swap_ranges	exchange values of two ranges
iter_swap	exchange values of objects pointed by two iterators
transform	apply function to range
replace	replace values in range
replace_if	replace elements in range satisfying condition
replace_copy	copy range replacing value
replace_copy_if	copy range replacing element satisfying condition
fill	fill range with value
fill_n	fill sequence with value
generate	generate values for range with function
generate_n	generate values for sequence with function

Modifying container operations

operation	description
remove	remove values from range
remove_if	remove elements from range satisfying condition
remove_copy	copy range removing values
remove_copy_if	copy range removing elements satisfying condition
unique	remove consecutive duplicates in range
unique_copy	copy range removing duplicates
reverse	reverse range
reverse_copy	copy range reversed
rotate	rotate elements in range
rotate_copy	copy rotated range
random_shuffle	re-arrange elements in range randomly
partition	partition range in two
stable_partition	partition range in two with stable ordering

Modifying container operations

- Example: copy
  - Copies all elements in a specified range [first,last) to another range

```
vector<string> playersName;
// names construction ...

// copy the first three names
// assuming there is at least 3 names
vector<string> podium;
copy(playersName.begin(), playersName.begin()+3, podium.begin());
```

Modifying container operations

- Other examples

```
// reverse the podium
reverse(podium.begin(), podium.end());

// shuffle the player names
random_shuffle(playersName.begin(), playersName.end());

// remove non-god player names
// assuming: bool NonGod(string s) { ... }
remove_if(playersName.begin(), playersName.end(), NonGod);
```

Sorting

operation	description
sort	sort elements in range
stable_sort	sort elements preserving order of equivalents
partial_sort	partially sort elements in range
partial_sort_copy	copy and partially sort range
nth_element	sort element in range

Sorting

- Sort
  - Sorts all the elements in a range (based on quicksort algorithm)
  - Uses operator < or a functor to order elements

```
class Player {
public:
   bool operator < (const Player& p) {
        return this->level_ < p.level_;
   }
};

vector<Player> players; // ...
sort(players.begin(), players.end());
```

Sorting

- Sort
  - Sorts all the elements in a range (based on quicksort algorithm)
  - Uses operator < or a functor to order elements

```
bool isPlayerLessThan(const Player& p1, const Player& p2) {
   return p1.level_ < p2.level_;
}

vector<Player> players; // ...
sort(players.begin(), players.end(), ptr_fun(isPlayerLessThan));
```

Sorting

- Sort is not “stable”
  - Two elements with the same order might end up in different relative positions
- If a stable sort is required, use the stable_sort algorithm instead
  - However, this runs slower than a normal sort
  - Relative order of equal elements will be preserved
- In case of a large number of elements and we only need the top X elements sorted, use partial_sort
  - [first,middle) contains the smallest elements of the entire range sorted in ascending order (not stable)
  - *[middle,end)* contains the remaining elements without any specific order

Sorting

- Example

```
class Player {
   public:
        bool operator < (const Player& p) {
                return this->level_ < p.level_;
        }
};

vector<Player> players; // ...

// sort players by level
// two players with same level -> unknown relative location
sort(players.begin(), players.end());

// -> always the same relative location (the original)
stable_sort(players.begin(), players.end());

// partial sort, to get only ordered podium
partial_sort(players.begin(), players.begin() + 3, players.end());
```

Binary search

operation	description
lower_bound	return iterator to lower bound
upper_bound	return iterator to upper bound
equal_range	get subrange of equal elements
binary_search	search element in range

Binary search

- Example

```
class Player {
   public:
        bool operator < (const Player& p) {
              return this->level_ < p.level_;
        }
};

vector<Player> players; // ...

// sort player by level
sort(players.begin(), players.end());

// find same as player1 by binary search
// players container is supposed to be sorted
bool found = binary_search(players.begin(), players.end(), player1);
```

Merge

operation	description
merge	merge sorted ranges
inplace_merge	merge consecutive sorted ranges
includes	test whether sorted range includes another sorted range
set_union	union of two sorted ranges
set_intersection	intersection of two sorted ranges
set_difference	difference of two sorted ranges
set_symmetric_difference	symmetric difference of two sorted ranges

Merge

- Examples

```
// sort players by level
sort(RedTeam.begin(), RedTeam.end());
sort(BlueTeam.begin(), BlueTeam.end());

// merge (and sort) in allPlayers
merge(RedTeam.begin(), RedTeam.end(),
      BlueTeam.begin(), BlueTeam.end(),
      allPlayers.begin());

// check if levels of blue team players are also
// levels of red team players
bool in = includes(BlueTeam.begin(), BlueTeam.end(),
                   RedTeam.begin(), RedTeam.end());
```

Min/max

operation	description
min	return the lesser of two arguments
max	return the greater of two arguments
min_element	return smallest element in range (iterator)
max_element	return largest element in range (iterator)
lexicographical_compare	lexicographical less-than comparison
next_permutation	transform range to next permutation
prev_permutation	transform range to previous permutation

Min/max

- Examples
  - Uses operator < or a functor to compare elements

```
cout << "Best player between " <<
          player1.getName() <<
          " and " <<
          player2.getName() <<
         " is " <<
         (max(player1,player2)).getName() ;

cout << "Total player higher level is " <<
      (*max_element(players.begin(),players.end())).getLevel();
```

Algorithms

- Usefulness of these algorithms can be increased by overloading operators <, ==, etc.
  - But do this within the rules!
- In order to use the algorithms, the following include is necessary

```
#include <algorithm>
```

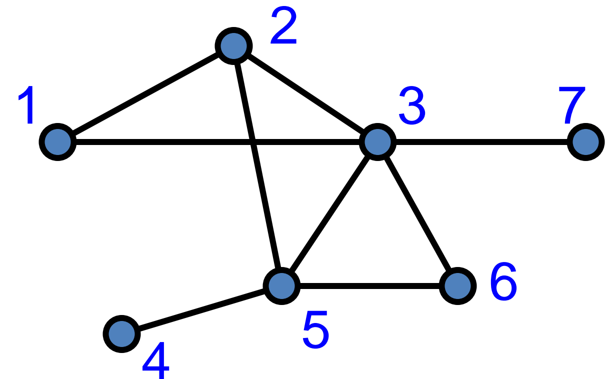
- All algorithms are in the std namespace
- Use STL design to create your own containers and algorithms

More containers

- The Boost library offers extensions to STL
  - including more containers such as
    - array
    - bidirectional map
    - circular buffer
    - disjoint set
    - graph
    - tree
    - ...
  - http://www.boost.org

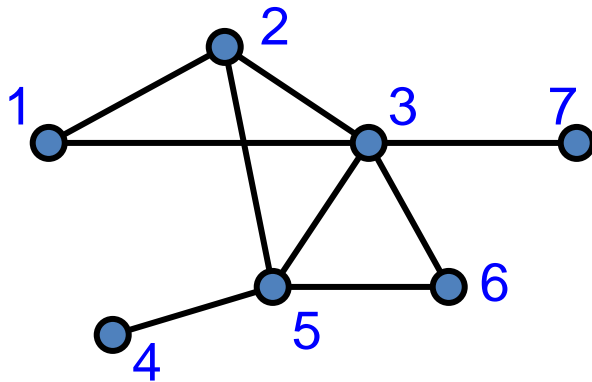
Graph

- A graph $G = (V, E)$ consists of a set V of vertices and a set E of edges
- Abstractly speaking, vertices are elements and edges are pairs of elements
- Example
  - $V = \{1, 2, 3, 4, 5, 6, 7\}$
  - $E = \{(1,2), (1,3), (2,3), (2,5), (3,5), (4,5), (3,6), (3,7), (5,6)\}$



Graph

- A common representation of a graph is the **adjacency matrix**, a *n x n* matrix of zeroes and ones with a one at *(i,j)* if and only if *(i,j)* is an edge in *E*

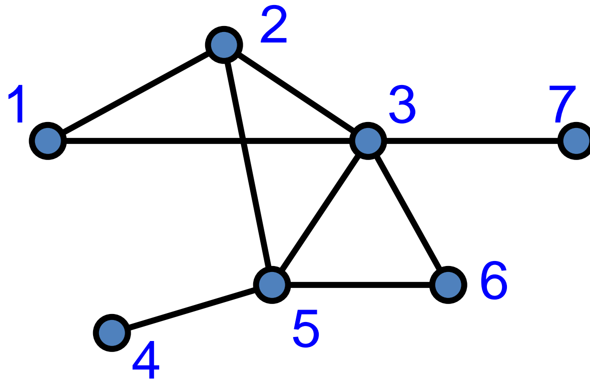


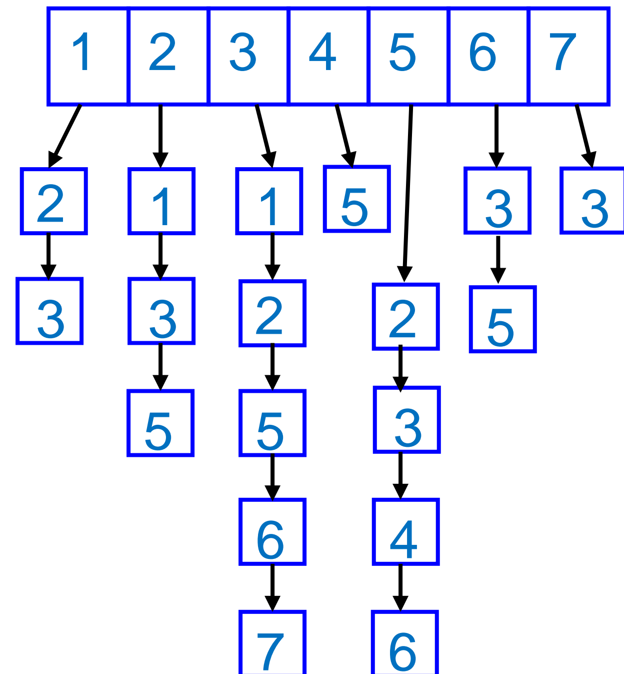
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

  - $V = \{1, 2, 3, 4, 5, 6, 7\}$
  - $E = \{(1,2), (1,3), (2,3), (2,5),$
    $(3,5), (4,5), (3,6), (3,7), (5,6)\}$

Graph

- A different common representation for graphs is the **adjacency list** representation
- It consists of an array $A[1 \cdots n]$, with one entry for each vertex, with access to a list of neighbors of that vertex





- $V = \{1, 2, 3, 4, 5, 6, 7\}$
- $E = \{(1,2), (1,3), (2,3), (2,5), (3,5), (4,5), (3,6), (3,7), (5,6)\}$

Boost Graph Library

- Boost proposes the two implementations
  - Classes adjacency_list and adjacency_matrix
  - Template classes describing the type of graph
    - 1st type represents the STL container used to store the edges (vecS for vector, listS for list *etc.*)
    - 2nd type represents the STL container used to store the vertices (vecS for vector, listS for list *etc.*)
    - 3rd type represents type of edges among bidirectional, directed or undirected
- Edges are then filled in the graph
  - Function add_edge

Boost Graph Library

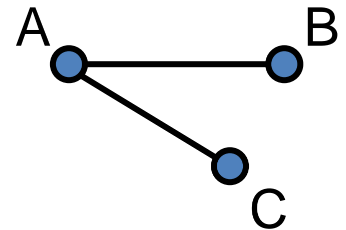
```
// create a typedef for the Graph type
typedef adjacency_list<vecS, vecS, bidirectionalS> Graph;

// Make convenient labels for the vertices
enum { A, B, C };
const int num_vertices = 3;

// writing out the edges in the graph
typedef std::pair<int, int> Edge;
Edge edge_array[] = { Edge(A,B), Edge(C,A) };
const int num_edges = 2;

// declare a graph object
Graph g(num_vertices);

// add the edges to the graph object
for (int i = 0; i < num_edges; ++i)
  add_edge(edge_array[i].first, edge_array[i].second, g);
```



Algorithms in Boost Graph Library

- Basic operations
- Core searches
- Other core algorithms
- Shortest paths / Cost minimization algorithms
- Minimum spanning tree algorithms
- Random spanning tree algorithm
- Connected components algorithms
- Maximum flow and matching algorithms
- Minimum cut algorithms
- Sparse matrix ordering algorithms
- Graph metrics
- Graph structure comparisons
- Layout algorithms
- Clustering algorithms
- Planar graph algorithms
- Miscellaneous algorithms

End of lecture #11

Next lecture

Game performance tuning