Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #2

Array, pointer, dynamic memory, string and OO basics

Arrays

- Sequence of elements of the same type placed in contiguous memory locations
- Individually referenced by adding an offset to the identifier (from 0, not 1)

```
type name [elements] ;
```

`elements` is constant

- Example

```
float position [3] ;
int orientation [3] ;
```

Initialization and access

- Initialization by enclosing the values in braces

```
float position [3] = {1.2, 3.5, 9.4};
int orientation [] = {45 90 0};
```

- Access with bracket [ ] operator

index out of range compiles!

```
if (position[0] == 1.2) orientation[2] = 10;
else {
   int orientY = orientation[1];
   position[1] = 3.6;
}
```

More features

- Multidimensional arrays can be described as "arrays of arrays“

```
type name [elements_dim1] [elements_dim2] ... ;
```

  - Example

```
float TransformationMatrix [4][4];
```

- Array as function parameter

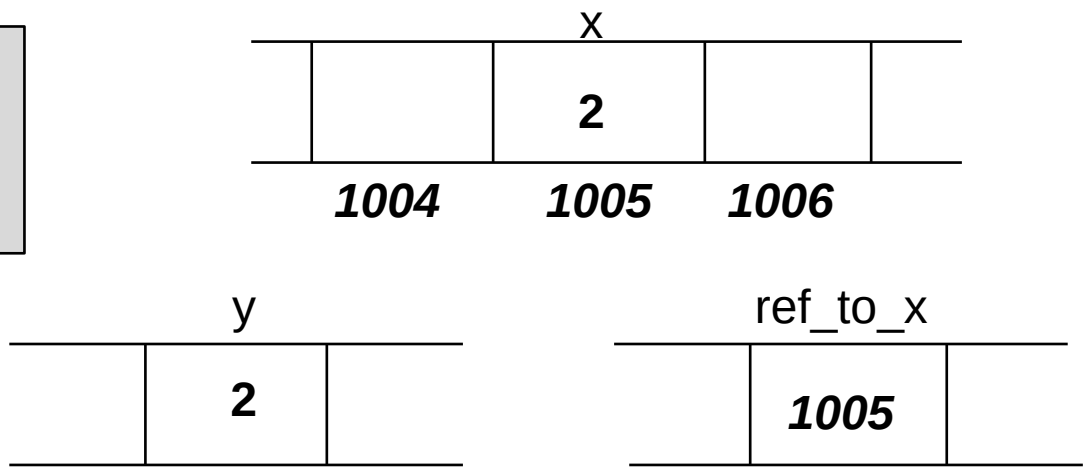
```
void printKills (int kills[], int nb_kills) {
   for (int n=0; n < nb_kills; n++) cout << kills[n] << " ";
}
int main () {
   int KillsPlayer1[] = {5, 10, 15}; int KillsPlayer2[] = {2, 4, 6, 8, 10};
   printKills (KillsPlayer1,3); printKills (KillsPlayer2,5);
   return 0;
}
```

Pointers

- Reminder
  - If x is a variable then &x is a reference to that variable (memory address)
  - Example

```
x = 2 ;
y = x ;
ref_to_x = & x ;
```

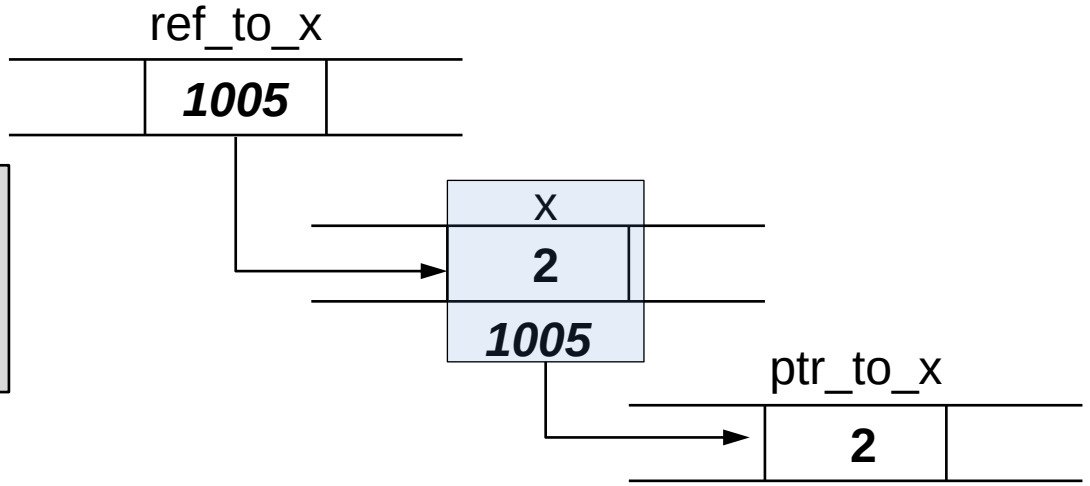
in memory



Pointers

- A variable which stores a reference to another variable is called a pointer
  - to directly access the value stored in the variable which it points to
  - by preceding the pointer's identifier with an asterisk (*), which acts as dereference operator
- Example

```
x = 2 ;
ref_to_x = & x ;
ptr_to_x = * ref_to_x ;
```



Pointers

- Pointers vs. references
  - References need to be assigned when declared
  - Pointers can point to nothing (NULL)
- Declaration also with * operator

```
type * name;
```

- Examples

```
int * ptrInteger ;
char * ptrCharacter ;
float * ptrNumber ;
```

Pointers

```
#include <iostream>
using namespace std;

int main () {
   int firstvalue = 5, secondvalue = 15;
   int * p1, * p2;
   p1 = &firstvalue;   // p1 = address of firstvalue
   p2 = &secondvalue;  // p2 = address of secondvalue
   *p1 = 10;           // value pointed by p1 = 10
   *p2 = *p1;          // value pointed by p2 = value pointed by p1
   p1 = p2;            // p1 = p2 (value of pointer is copied)
   *p1 = 20;           // value pointed by p1 = 20
   cout << "firstvalue is " << firstvalue << endl;
   cout << "secondvalue is " << secondvalue << endl;
   return 0;
}
```

More features

- Multiple pointers can point to the same variable

```
int x = 1;
int * ptrX1 = & x;
int * ptrX2 = & x;
* ptrX1 = 2;
cout << x << " " << *ptrX1 << " " << *ptrX2;
```

- Pointer to pointer

```
int x = 1;
int * ptrX = & x;
int ** ptrptrX = & ptrX;
cout << ** ptrptrX;
```

- void pointers, null pointer, pointer to functions, pointer as parameter ...

Pointers

- Dangling pointer

```
char x = 'x';
char *cp = &x;
if (x == 'x') {
   char y = 'y';
   cp = &y;
}
// now cp is a dangling pointer
```

- NULL address is used to indicate that pointer is not pointing to data

```
int * p = NULL;
```

  - Checking for pointer validity in if statement

```
if (p != NULL) { // or just (p)
   // you can use variable p
}
```

Dynamic memory

- When we need a variable amount of memory that can only be determined during runtime
- Operators new and delete

```
type * pointer = new type ;
delete pointer;
```

```
type * pointer = new type [number_of_elements] ;
delete [] pointer;
```

- number_of_elements does not required to be constant ☺ !

Dynamic memory

```
#include <iostream>
using namespace std;

int main () {
   unsigned int nb_char,n;
   char * name;
   cout << "How many characters has your name?";
   cin >> nb_char;
   if (nb_char == 0) cout << "Error: no character in your name";
   else {
        name = new char [nb_char];
        for (n=0; n<nb_char; n++) {
                cout << "Enter character: ";
                cin >> name[n];
        }
        cout << "You have entered: ";
        for (n=0; n<nb_char; n++) cout << name[n] ;
        delete[] name;
   }
   return 0;
}
```

Dynamic memory

- Arrays (summary)
  - Syntax for declaring array (stack)

```
type array_name [array_size];
```

  - Syntax for declaring array (heap)

```
type * array_name = new type [array_size];
delete [] array_name;
```

  - Example: an array of integers, size 10

```
int array1 [10];
int * array2 = new int [10];
delete [] array2;
```

    - In both cases, the type is int [10]

Dynamic memory

- Arrays
  - Creating and destroying multi-dimensional heap arrays

```
int ** array2D = new int * [10];
for (int i=0; i<10; i++) array2D[i] = new int [5];
```

```
for (int i=0; i<10; i++) delete [] array2D[i];
delete [] array2D;
```

String in C++

- Basic string as array of char
  - char [ ]
  - string ends with character ‘\0’

```
#include <iostream>
using namespace std;
int main() {
   char question[] = “What is your name? ”;
   char greeting[] = {'H','e','l','l','o',' ','\0'};
   char playerName[80];
   cout << question;
   cin >> playerName;
   cout << greeting << playerName << endl;
   return 0;
};
```

String in C++

- Not possible to do easily with basic strings
  - String comparison
  - Assignment
  - Concatenation and insertion
  - Determination of the length

String in C++

- Whenever possible, use the string class of the std library
  - defined as
  - in library

```
typedef basic_string<char> string;
```

```
#include <string>
```

- Implement various functionalities
  - assignment
  - size manipulation
  - iterator and direct access to char
  - concatenation, insertion, replacement
  - search and comparison, and more
  - automatic conversion from / to char *

String in C++

```
#include <string>
using namespace std;

int main() {
    string playerName;
    cout << "What is your name? ";
    cin >> playerName;
    cout << "The length of your name is: " << playerName.length() << endl;
    if (!playerName.empty())
        cout << "Your initial is: " << playerName[0] << endl;
    if (playerName.compare("God") == 0)
        cout << "You have the same name as me!" << endl;
    else {
        playerName = "Mr./Ms. " + playerName;
        cout << "Goodbye, " << playerName;
    }
    return 0;
}
```

Object-oriented basics

- Class
  - extended concept of data structure
  - hold data and functions, called members
- Object
  - instance of a class

```
class class_name {
   access_specifier_1:
        member1;
   access_specifier_2:
        member2;
   ...
} object_names;
```

Classes: member protection

- access specifier can be
  - private (default): access only within own class
  - public: access from any class
  - protected: access from own class and derived
- a friend function or class has access to the private/protected/public members

```
friend type functionName (parameters); // in friend class
```

```
friend class className;                // in friend class
```

Classes: declaration and definition

```
#include <iostream>
using namespace std;

class Player {
          float posX,posY;
          int amno;
    public:
          int shoot() {amno --; return amno;}
          void setPosXPosY(float,float);
          void setAmno(int a) {amno = a;}
};

void Player::setPosXPosY(float newX, float newY) {
    posX = newX; posY = newY;
}

int main() {
    Player player1;
    player1.setPosXPosY(2.3,8.0);
    player1.setAmno(10);
    cout << "Amno left: " << player1.shoot();
    return 0;
}
```

Classes: declaration and definition

- Declarations in header file (Player.h)

```
#ifndef PLAYER_H_
#define PLAYER_H_

class Player {
        float posX,posY;
        int amno;
   public:
        int shoot();
        void setPosXPosY(float,float);
        void setAmno(int);
};

#endif
```

Classes: declaration and definition

- Definitions in body file (Player.cpp)

```
#include “Player.h”

int Player::shoot() {
    amno --;
    return amno;
}

void Player::setPosXPosY(float newX, float newY) {
    posX = newX;
    posY = newY;
}

void Player::setAmno(int newAmno) {
    amno = newAmno;
}
```

Classes: declaration and definition

- Program in main file (program.cpp)

```
#include <iostream>
using namespace std;

#include "Player.h"

int main() {
    Player player1;
    player1.setPosXPosY(2.3,8.0);
    player1.setAmno(10);

    Player player2;
    player2.setPosXPosY(4.5,8.5);
    player2.setAmno(4);

    cout << "Player 1 amno left: " << player1.shoot();
    cout << "Player 2 amno left: " << player2.shoot();

    return 0;
}
```

Classes: constructor and destructor

- Objects generally need to initialize variables or assign dynamic memory
- Class includes a special function called constructor
  - automatically called whenever a new object of this class is created
  - has the same name as the class
  - has no return type
- Class includes a special function called destructor
  - automatically called when an object is destroyed when
    - its scope of existence has finished
    - released using the operator delete
  - has the same name as the class preceded with a tilde sign (~)
  - has no return type
- Especially suitable when an object assigns dynamic memory during its lifetime
- Constructors and destructors are called in cascade

Classes: constructor and destructor

```
#include <iostream>
using namespace std;

class Player {
          float * posX, * posY;
          int amno;
    public:
          Player (float,float,int);
          ~Player ();
          int shoot() {amno --; return amno;}
};

Player::Player (float newX, float newY, int newAmno) {
    posX = new float; posY = new float;
    *posX = newX; *posY = newY; amno = newAmno;
}

Player::~Player () {
    delete posX; delete posY;
}

int main() {
    Player player1(2.3,8.0,10);
    cout << "Amno left: " << player1.shoot();
    return 0;
}
```

Classes: constructor and destructor

- Default constructor
  - If no constructor is specified the compiler assumes the class to have a default constructor with no arguments
  - If a constructor is specified, the compiler no longer provides an implicit default constructor. All objects have to be created with that constructor
- Copy constructor
  - copy all the data contained in an object to the data members of the current object

```
className::className (const className& copiedObject)
```

```
className object1 (dataMember1,dataMember2);
className object2 (object1);
```

Static members

- Static data
  - shared over all instances of a class
  - only prototype in class declaration

```
class Team {
public:
    static int nbTeams;
    Team () {nbTeams++;};
    ~Team () {nbTeams--;};
};
```

```
int Team::nbTeams = 0;
```

```
#include <iostream>
using namespace std;
#include "Team.h"

int main() {
    Team redTeam;
    Team blueTeam;
    cout << redTeam.nbTeams;
    return 0;
}
```

- Static functions
  - can only refer to static members (as not associated to one instance)

Keyword “const”

- const has various usages
  - After a method declaration, means that the method does not change data members

```
int getCurrentAmno() const;
```

  - In front of a variable, variable is treated like a constant (even as a function parameter)

```
const int maxPlayer = 99;
maxPlayer ++; // Error: variable is constant!
```

```
int Player::getTotalAmno(const int grenades,
                         const int bullets) const {
   return (grenades + bullets);
}
```

Creating instances

```
#include <iostream>
using namespace std;

class Player {
        float posX, posY;
        int amno;
   public:
        Player (float newX, float newY, int newAmno);

        ~Player ();

        void shoot() {amno --;}

        int getAmno() const { return amno; }

        void Move(const float offsetX, const float offsetY){
           posX += offsetX; posY += offsetY;
        }
};
```

Creating instances

- Constructor

```
Player::Player (float newX, float newY, int newAmno) {
    posX = newX; posY = newY; amno = newAmno;
}
```

or *(optimized)*

```
Player::Player (float newX, float newY, int newAmno) :
    posX(newX),
    posY(newY),
    amno(newAmno) {
    // Rest of creation
}
```

Creating instances

- Data structure on the stack
  - Creation

```
Player player1 (2.0,5.6,10);
```

  - Usage

```
player1.shoot();
```

- Data structure on the heap
  - Creation

```
Player * player1 = new Player (2.0,5.6,10);
```

  - Usage

```
player1->shoot(); // or (*player1).shoot();
```

Keyword “this”

- Represents a pointer to the object whose member function is being executed (pointer to the object itself)
  - needs -> to access the instance members

```
void Player::AddAmno(int moreAmno) {
    int currentAmno = this->getAmno();
    this->amno = currentAmno + moreAmno;
}
```

Default values

- Default values of function parameters can be specified in the function declaration

```
void move(const float offsetX = 0.1, const float offsetY = 0.1);
```

```
void Player::move(const float offsetX, const float offsetY) {
    posX += offsetX; posY += offsetY;
}
```

```
player1.move();              // uses both default values
player1.move(0.5,0.5);       // uses parameters
player1.move(0.5);           // uses default offsetY

// no way to use default offsetX and user offsetY:
// player1.move(,0.5) is not a valid call
```

Default values

-  Be careful!
  - All default parameters must be the rightmost parameters

```
void move(const float offsetX = 0.1, const float offsetY);
// This is not allowed
```

  - The leftmost default parameter should be the one most likely to be changed by the user

```
void move(const float offsetX, const float offsetY = 0.1);
```

```
player1.move();              // Error: missing offsetX
player1.move(0.5,0.5);       // OK: uses both user parameters
player1.move(0.5);           // OK: uses default offsetY
```

Overloading

- Allow to create similar functions with different parameter signatures

```
class Player {
        float posX, posY;
        int amno;
   public:
        Player (float newX, float newY, int newAmno);
        ~Player ();
        void shoot();
        void setAmno(const int newAmno);
        void setAmno(const float newAmno);
};
```

Overloading

```
#include <iostream>
using namespace std;
#include "Player.h"

int main() {
    Player player1 (2.0,3.0,10);
    int amnoInt = 5;
    float amnoFloat = 5.0;
    player1.setAmno(amnoInt);       // calls setAmno(int)
    player1.setAmno(amnoFloat);     // calls setAmno(float)
    return 0;
}
```

Overloading

- Very handy to create multiple constructors for user convenience
  - Constructor for creating a default instance
  - Constructors for creating instances according to different parameters
  - Constructor for reading instance from a file
  - etc.
-  A class can only have **one destructor!**

End of lecture #2

Next lecture

Advanced OO, STL, compilation and programming