Classes 2

- composition.
- const
- this
- static

Part 1: composition

Example: A car has 4 wheels

"Has a" relationship. The owner has a component,

The component has its existence managed by the owner (object).

Object pointer in a class: dynamic memory allocation

```
class Time
{
public:
  Time( int seconds );
private:
  int m_Seconds;
};
```

```
#include "Time.h"

Time::Time(int seconds)
: m_Seconds{seconds}
{
}
```

```
class Time; // 1

class Game
{
public:
  Game();
private:
  Time *m_pTime;
};
```

```
#include "MyClass.h"
#include "Time.h " // 2

Game::Game()
  : m_pTime{ new Time{ 15 } } // 3
{ }

Game::~Game()
{
  delete m_pTime; // 4
}
```

- The Game class has a Time object.
- A Time object pointer variable is added to the Game class. Using dynamic memory allocation, the memory address of an object is assigned to the pointer. It occupies heap memory.

Object pointer in a class: dynamic memory allocation

```
class Time
{
public:
  Time( int seconds );
private:
  int m_Seconds;
};
```

```
#include "Time.h"

Time::Time(int seconds)
: m_Seconds{seconds}
{
}
```

```
class Time; // 1

class Game
{
public:
  Game();
private:
  Time *m_pTime;
};
```

```
#include “Game.h"
#include "Time.h " // 2

Game::Game()
  : m_pTime{ new Time{ 15 } } // 3
{ }

Game::~Game()
{
  delete m_pTime; // 4
}
```

- The Game class has a Time object.
- A Time object pointer variable is added to the Game class. Using dynamic memory allocation, the memory address of an object is assigned to the pointer. It occupies heap memory.
  1. Add a class forward declaration in the header file.

Object pointer in a class: dynamic memory allocation

```
class Time
{
public:
  Time( int seconds );
private:
  int m_Seconds;
};
```

```
#include "Time.h"

Time::Time(int seconds)
: m_Seconds{seconds}
{
}
```

```
class Time; // 1

class Game
{
public:
  Game();
private:
  Time *m_pTime;
};
```

```
#include "MyClass.h"
#include "Time.h " // 2

Game::Game()
  : m_pTime{ new Time{ 15 } } // 3
{ }

Game::~Game()
{
  delete m_pTime; // 4
}
```

- The Game class has a Time object.
- A Time object pointer variable is added to the Game class. Using dynamic memory allocation, the memory address of an object is assigned to the pointer. It occupies heap memory.
  1. Add a class forward declaration in the header file.
  2. Add the #include statement in the cpp file
  3. The pointer is initialized in the member initialization list. A heap object is instantiated using the new operator. It returns the heap address of the created object that is used to initialize the pointer.
  4. The object is destroyed in the destructo

Object pointer in a class: dynamic memory allocation

```
class Time
{
public:
  Time( int seconds );
private:
  int m_Seconds;
};
```

```
#include "Time.h"

Time::Time(int seconds)
: m_Seconds{seconds}
{
}
```

```
class Time; // 1

class Game
{
public:
  Game();
private:
  Time *m_pTime;
};
```

```
#include "MyClass.h"
#include "Time.h " // 2

Game::Game()
  : m_pTime{ new Time{ 15 } } // 3
{ }

Game::~Game()
{
  delete m_pTime; // 4
}
```

- The Game class has a Time object.
- A Time object pointer variable is added to the Game class. Using dynamic memory allocation, the memory address of an object is assigned to the pointer. It occupies heap memory.
  1. Add a class forward declaration in the header file.
  2. Add the #include statement in the cpp file
  3. The pointer is initialized in the member initialization list. A heap object is instantiated using the new operator. It returns the heap address of the created object that is used to initialize the pointer.
  4. The object is destroyed in the destructo

Object pointer in a class: dynamic memory allocation

```
class Time
{
public:
  Time( int seconds );
private:
  int m_Seconds;
};
```

```
#include "Time.h"

Time::Time(int seconds)
: m_Seconds{seconds}
{
}
```

```
class Time; // 1

class Game
{
public:
  Game();
private:
  Time *m_pTime;
};
```

```
#include "MyClass.h"
#include "Time.h " // 2

Game::Game()
  : m_pTime{ new Time{ 15 } } // 3
{ }

Game::~Game()
{
  delete m_pTime; // 4
}
```

- The Game class has a Time object.
- A Time object pointer variable is added to the Game class. Using dynamic memory allocation, the memory address of an object is assigned to the pointer. It occupies heap memory.
  1. Add a class forward declaration in the header file.
  2. Add the #include statement in the cpp file
  3. The pointer is initialized in the member initialization list. A heap object is instantiated using the new operator. It returns the heap address of the created object that is used to initialize the pointer.
  4. The object is destroyed in the destructo

Classes 2

- composition.
- const
- this
- static

Const member function qualifier

```
void HandlePoint(const Point2f & p)
{
  p.x = p.y = 0;
}
```

- What if an object is passed as a const ref parameter?
- Compile error when the function tries to modify the const object!

Const member function qualifier

```
void HandlePoint(const Point2f & p)
{
  p.Print();
  p.Reset();
}
```

```
void Point2f::Reset()
{
  x = y = 0;
}
```

```
void Point2f::Print()
{
  std::cout << x << " " << y;
}
```

- What if the const object has member functions that modify the member variables…??? Is there a security breach?
- Are we allowed to call a member function that modifies the data members?
- No.
- It is not allowed to call any member function of a const object, unless it is absolutely sure that the function does not modify data members.

Const member function qualifier

```
void HandlePoint(const Point2f & p)
{
  p.Print();
  p.Reset();
}
```

```
void Point2f::Reset()
{
  x = y = 0;
}
```

```
void Point2f::Print()
{
  std::cout << x << " " << y;
}
```

- How can the compiler be sure that a member function does not modify data members?

DIGITAL ARTS & ENTERTAINMENT

Const member function qualifier

```
void HandlePoint(const Point2f & p)
{
  p.Print(); // const method -> ok!
  //p.Reset(); // not ok!
}
```

```
void Point2f::Reset()
{
  x = y = 0;
}
```

```
void Point2f::Print() const
{
  std::cout << x << " " << y;
}
```

- How can the compiler be sure that a member function does not modify data members?
- When the function is marked as "const"

```
class Point2f
{
public:
  Point2f( );
  void Reset()
  void Print() const;
  …
```

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

Const member function qualifier

```
void Point2f::Reset()
{
  x = y = 0;
}
```

```
void HandlePoint(const Point2f & p)
{
  p.Print(); // const method -> ok!
  p.Reset(); // not ok!
}
```

```
void Point2f::Print() const
{
  std::cout << x << " " << y;
}
```

- Calling a non-const member function from a const object results in a compile error:

```
game.cpp(27): error C2662: 'void Point2f::Reset(void)': cannot convert 'this' pointer from 'const Point2f' to 'Point2f &'
game.cpp(27): note: Conversion loses qualifiers
```

?

Const member function qualifier

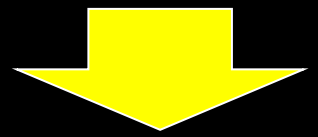
A const member function is a member function that guarantees it will not modify the object or call any non-const member functions (as they may modify the object).

Always use the const suffix to mark member functions as "inspectors" instead of "mutators" if they do not modify data members. They are called: "const member functions"

It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided. (GeeksforGeeks)

Const member functions: example

```
float Rectangle::GetArea()
{
  return ( m_X2 - m_X1 ) * ( m_Y2 - m_Y1 );
}
```



```
float Rectangle::GetArea() const
{
  return ( m_X2 - m_X1 ) * ( m_Y2 - m_Y1 );
}
```

- GetArea() does not modify any data members.
- We know this, but the compiler assumes it can happen!
- Appending the keyword "const" to the function declaration and definition turns it into a const member function.
- Modifying member vars in a const member function results in compile errors.

Const member functions: UML

- UML specs for a const member function: {query}
- GetArea() : int {query}

Classes 2

- composition.
- const
- this
- static

Let's return to the slide: "this"

```
void HandlePoint(const Point2f & p)
{
  p.Print(); // const method -> ok!
  p.Reset(); // not ok!
}
```

```
void Point2f::Reset()
{
  x = y = 0;
}
```

```
void Point2f::Print() const
{
  std::cout << x << " " << y;
}
```

```
game.cpp(27): error C2662: 'void Point2f::Reset(void)': cannot convert 'this' pointer from 'const Point2f' to 'Point2f &'
game.cpp(27): note: Conversion loses qualifiers
```

?? 'this' pointer ??

Memory areas

The memory a program uses can be roughly divided into a few different areas, called segments:

- The code segment (also called a text segment), where the compiled program (functions) sits in memory. The code segment is typically read-only.
- The (un)initialized data segment, where (un)initialized (const) global and local static variables are stored.
- The heap, where dynamically allocated variables are allocated from.
- The call stack, where function parameters, local variables, and other function-related information are stored.

DIGITAL ARTS & ENTERTAINMENT

Memory areas

- (Member)Functions and variables are stored in different memory areas!

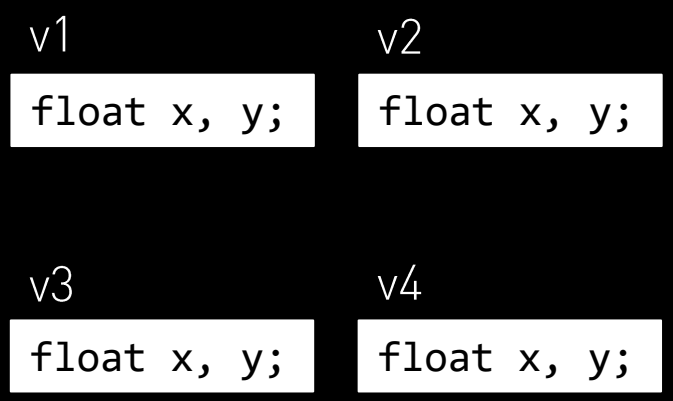
- Example: 4 objects of a class.
  - For each object, a different set of member variables is stored.
  - Only one set of member functions of that class is stored. (!)

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

Example: class Vector2f

```
Vector2f v1{ 10,15 }, v2{ 20,50 }, v3{}, v4{};
v1.x = 10; // ok
float l = v1.GetLength();
int s = sizeof(Vector2f); // s is 8
```

Several different object instances are stored in stack/heap memory



Only one copy of each member function is stored in the code segment memory,

```
Vector2f::Vector2f();
Vector2f::Vector2f(float, float);
float Vector2f::GetLength();
float Vector2f::DotProduct(const Vector2f&);
. . .
```

Hidden "this" pointer

1. Every instance of a class has its own set of member variables.
2. The member functions, however, are shared among all objects.

- Interesting question:
  - How does a member function know what object to operate on?

A hidden "this" pointer refers to the object

DIGITAL ARTS & ENTERTAINMENT

Hidden "this" pointer

```
int main()
{
  Vector2f p{ 1, 2 };
  p.SetX(5);
  std::cout << p.GetX() << endl;
}
```

```
void Vector2f::SetX(float x)
{
  m_X = x;
}
```

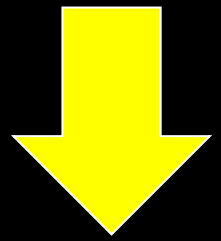
Let's visualize the hidden stuff

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

DIGITAL ARTS & ENTERTAINMENT

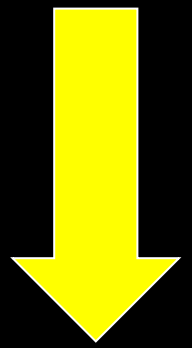
Hidden "this" pointer

```
int main()
{
  Vector2f p{ 1, 2 };
  p.SetX(5);
  std::cout << p.GetX() << endl;
}
```



```
int main()
{
  Vector2f p{ 1, 2 };
  Vector2f::SetX(&p, 5);
  std::cout << p.GetX() << endl;
}
```

```
void Vector2f::SetX(float x)
{
  m_X = x;
}
```



- The compiler converts the code into this. (see red)
- The compiler adds a hidden const this pointer parameter to all member functions. Except for static member functions.

```
void Vector2f::SetX(Vector2f* const this, float x)
{
    this->m_X = x;
}
```

const here indicates that the pointer variable can not be modified, the object "this" points to can be modified.

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

DIGITAL ARTS & ENTERTAINMENT

Hidden "this" pointer

"this" can be accessed in any member function:

```
float Rectangle::GetWidth() const
{
  return m_X2 - m_X1;
}
```

```
float Rectangle::GetWidth() const
{
  return this->m_X2 - this->m_X1;
}
```

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

DIGITAL ARTS & ENTERTAINMENT

Part 3: Const objects

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

Const objects

- Recap:
  - Fundamental data types (int, double, char, etc…) can be made const
  - const variables must be initialized at time of creation:
    - copy, direct or uniform initialization
  - Example:

```
const int number1 = 2;   // copy initialization
const int number3{ 6 };  // uniform initialization (C++11)
```

Const class objects

- Const objects must also be initialized at time of creation:

```
const Vector2f v1;          //init using default constructor
const Vector2f v2( 2, 3 ); //init using parameterized constructor
const Vector2f v3{ 4, 5 }; //init using parameterized constructor
```

- const member variable objects can be initialized in the constructor initializer list.

```
class Ball
{
public:
  Ball();
private:
  const Vector2f m_Gravity;
}
```

```
Ball::Ball() : m_Gravity{0,-9.81f}
{
}
```

Const class objects

```
class Vector2f
{
public:
  Vector2f(float x, float y);
  void SetX(float x);
  float GetX() const;
private:
  float m_X, m_Y;
};

int main() {
  const Vector2f v1{ 2,3 };
  v1.SetX(12);
  float y = v1.GetX();
  cin.get();
}
```

ERROR

- const objects may not be modified. Any attempt to do so will fail.
- Only const methods (member functions) can be called.

- v1 is const and initialized through the constructor
- v1 can NOT be modified
- v1.SetX(12) → error.

Classes 2

- composition.
- const
- this
- static

Static - introduction

- The static keyword can be used to declare variables, functions, class data members and class functions.

- Reference: https://msdn.microsoft.com/en-us/library/y5f6w579.aspx

- Static member variables
- Static member functions
- Static const member variables

recap: Memory areas

The memory a program uses can be roughly divided into a few different areas, called segments:

- The code segment (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.
- The (un)initialized data segment, where (un)initialized (const) global and local static variables are stored.
- The heap, where dynamically allocated variables are allocated from.
- The call stack, where function parameters, local variables, and other function-related information are stored.

Static member variables

- Non-static member variables: Each object has its own copy of these variables.

- Static member variables:
  - Are shared by all the objects of a class.
  - Are not on the stack, occupy the global memory area of the application.
  - Are also called: "Class variables".
  - Have the lifetime of the application.
  - Have global scope, limited by the access specifiers. (private, protected, public)

Static member variables

Declaration:

- When you declare a static data member in a class declaration, the static keyword specifies that one copy of the member is shared by all instances of the class.

```
class Sprite
{
  Sprite();
  static int m_InstanceCounter;
};
```

DIGITAL ARTS & ENTERTAINMENT

Static member variables

- Initialization:
  - Not possible in the header file (!)
  - At file scope → in the cpp file:
  - Is obligatory

```
#include "Sprite.h"

int Sprite::m_InstanceCounter{ 0 };

Sprite::Sprite()
{

}
```

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

Static member variables

- Access to static data members
  - You can access static data members through any instance of that class using the member select operators "." and "->"
  - If public, they can even be accessed via the class name using the scope resolution operator "::"
    - PS: public data members only when working with structs!
- The rules for access specifiers do apply. (private, public)

DIGITAL ARTS & ENTERTAINMENT

Static member functions

- Can be called without making an object first.
- Restriction: Static methods can only access other static elements, because they do not have a hidden "this" pointer.

```
class Sprite
{
public:
  Sprite( );
  ~Sprite( );

  void Draw(const Point2f& position);
  static int GetNumberOfInstances();
private:
  static int m_InstanceCounter;
};
```

```
int Sprite::m_InstanceCounter{ 0 };

int Sprite::GetNumberOfInstances()
{
   return m_InstanceCounter;
}

Sprite::Sprite()
{
  ++m_InstanceCounter;
}

Sprite::~Sprite()
{
  --m_InstanceCounter;
}
```

Static member functions

- You use them by calling them on the name of the class and the scope operator(can also be called on objects).

```
MyClass::SomeStaticMethod();
```

- or by using the member select operators

```
m_MyObjectPtr->SomeStaticMethod();
```

```
int main()
{
  int number = Sprite::GetNumberOfInstances();
  std::cout << number; // 0 is printed

  m_pSprite = new Sprite();
  number = m_pSprite->GetNumberOfInstances();
  std::cout << number; // 1 is printed
}
```

DIGITAL ARTS & ENTERTAINMENT

Static uml representation

- Anything static is underlined

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

static const variables

- static variables can be const
  - The rules for access specifiers apply as normal:
    - If public (is OK), then the variable can even be accessed via the class name.
  - Only integral data members that are declared as const static can have an initializer in the header file.

```
private:
  static const int m_ArraySize { 10 };
  static double m_Pi;
```

static const variables

- integers that are static const can be initialized in the header.
- All non int types MUST be initialized in the cpp file:

```
#include "Sprite.h"

const double Sprite::m_Pi = 3.14159265359;

Sprite::Sprite()
{

}
```

modifier static and static arrays

- "The size of an array must be known at compile time"

- What if an object has a static array as member?
  - And a "const int" is used to define the number of elements.
  - The object is created at runtime. → The const is created at runtime
  - Conclusion: the size is not known at compile time.
  - Solution: make it "static const int"
  - Can be public if needed (is const)
  - Initialize it in the header file.

```
class MyClass
{
public:
  MyClass();
private:
  static const int m_Size{ 10 };
  int m_Numbers[m_Size];
};
```

references

- https://msdn.microsoft.com/en-us/library/y5f6w579.aspx
- https://isocpp.org/wiki/faq/const-correctness
- http://www.learncpp.com/cpp-tutorial/79-the-stack-and-the-heap/
- https://en.wikipedia.org/wiki/Data_segment

Bonus: Hidden "this" pointer used for chaining

```
class Vector2f
{
  public:
  Vector2f(float x, float y) : m_X{ x }, m_Y{ y }
  {}
  Vector2f& Add(const Vector2f& other)
  {
    this->m_X += other.m_X;
    m_Y += other.m_Y;
    return *this;
  }
private:
  float m_X, m_Y;
};

int main() {
  Vector2f v1{ 2,3 }, v2{ 5, 8 }, v3{ 3, 1 };
  Vector2f vSum = v1.Add(v2).Add(v3);
  cin.get();
}
```

- by returning a reference to the object Add was called upon, the Add calls can be "chained" together.

```
// Static array of pointers
Time* pTimes[4]{};
pTimes[0] = new Time{ 14 };
pTimes[0]->AddHours(10);
delete pTimes[0];
```