

Overview course Programming 1

1. Variables 1
2. Variables 2
3. Variables 3
4. Conditionals
5. Iterations
6. Functions 1
7. Functions 2
8. Arrays
9. Strings – Game
10. **Classes 1 – Encapsulation**
11. Classes 2 – Static const

Object Orientation

A new way of thinking.

Object orientation

- Example: Water cooker

- Properties:

- width, height, color, volume
 - on/off state

- Behavior:

- When on, it heats its content
 - When a temperature is reached, it switches off automatically

Not using object orientation:

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

struct Vector2f
{
    float x, y;
};

void PrintVector2f(const Vector2f& v)
{
    cout << "(" << v.x << ", " << v.y << ")" << endl;
}

int main()
{
    Vector2f v1, v2;
    v1.x = 21.0f;
    PrintVector2f(v1);
    PrintVector2f(v2);


    cin.get();
}
```

- the data and function are separated
- the function needs the data as parameter

Using object orientation:

```
struct Vector2f
{
    float x, y;
    void Print()
    {
        std::cout << '(' << x << ", " << y << ")" << '\n';
    }
};

int main()
{
    Vector2f v1, v2;
    v1.x = 21.0f;
    v1.Print();
    v2.Print();
}
```

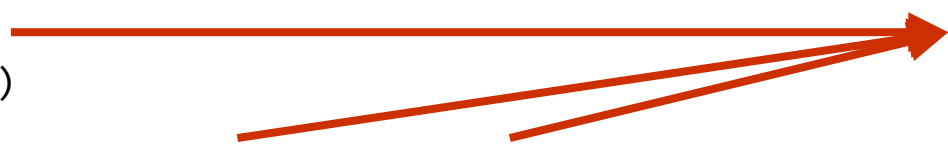


- Functions are added to the struct.

Using object orientation:

```
struct Vector2f
{
    float x, y;
    void Print()
    {
        std::cout << '(' << x << ", " << y << ")" << '\n';
    }
};

int main()
{
    Vector2f v1, v2;
    v1.x = 21.0f;
    v1.Print();
    v2.Print();
}
```

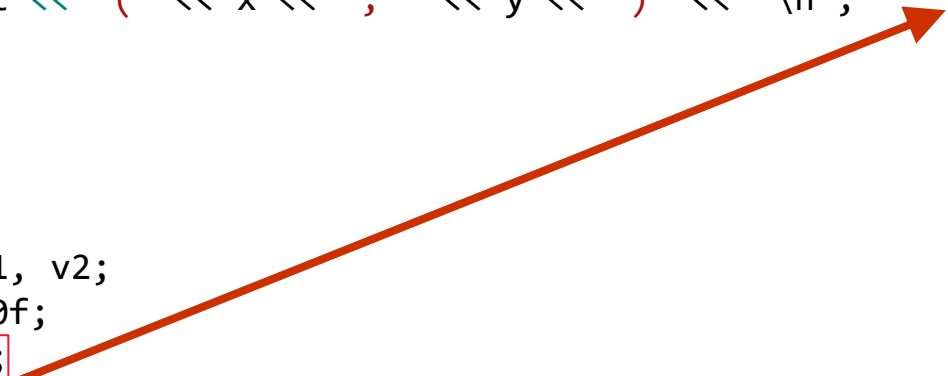
A diagram with three orange arrows pointing from the right side of the code to a single point on the right. The first arrow starts at 'float x, y;' in the struct definition. The second arrow starts at 'std::cout << '(' << x << ', ' << y << ')' << '\n';' in the Print() function. The third arrow starts at 'v1.x = 21.0f;' in the main() function. All three arrows converge towards the right edge of the slide, illustrating that functions have direct access to the data without needing a parameter.

- Functions are added to the struct.
- Functions have direct access to the data, not needing a parameter.

Using object orientation:

```
struct Vector2f
{
    float x, y;
    void Print()
    {
        std::cout << '(' << x << ", " << y << ")" << '\n';
    }
};

int main()
{
    Vector2f v1, v2;
    v1.x = 21.0f;
    v1.Print();
    v2.Print();
}
```




- Functions are added to the struct.
- Functions have direct access to the data, not needing a parameter.
- Functions can be called the same way the data can be accessed: member selection operator '.'.

Using object orientation:

```
struct Vector2f
{
    float x, y;
    void Print()
    {
        std::cout << '(' << x << ", " << y << ")" << '\n';
    }
};

int main()
{
    Vector2f v1, v2;
    v1.x = 21.0f;
    v1.Print();
    v2.Print();
}
```

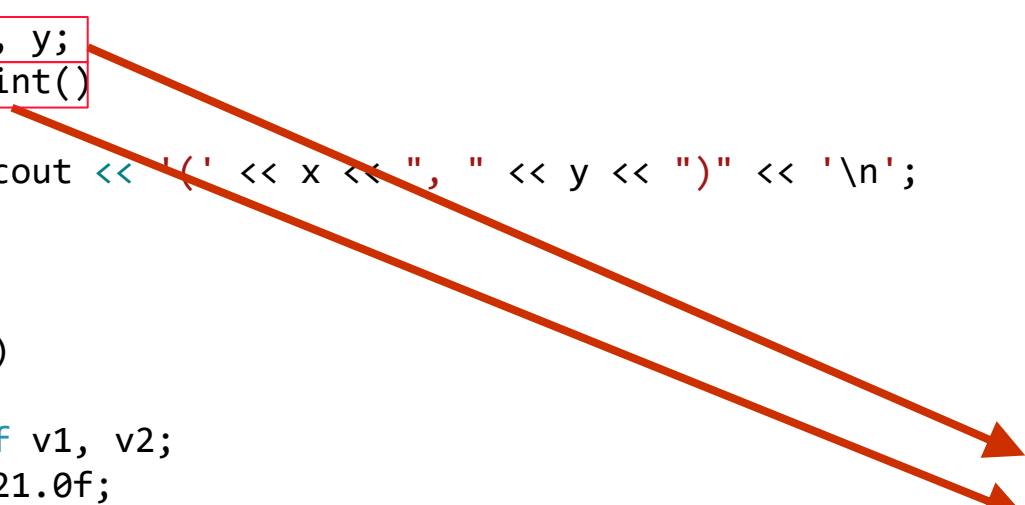


- Functions are added to the struct.
- Functions have direct access to the data, not needing a parameter.
- Functions can be called the same way the data can be accessed: member selection operator '.'.
- v1 and v2 are **instances** or **objects** of Vector2f

Using object orientation:

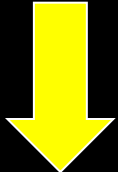
```
struct Vector2f
{
    float x, y;
    void Print()
    {
        std::cout << "(" << x << ", " << y << ")" << '\n';
    }
};

int main()
{
    Vector2f v1, v2;
    v1.x = 21.0f;
    v1.Print();
    v2.Print();
}
```



- Functions are added to the struct.
- Functions have direct access to the data, not needing a parameter.
- Functions can be called the same way the data can be accessed: member selection operator '.'.
- v1 and v2 are **instances** or **objects** of Vector2f
- x, y are **"datamembers"**
- Print is a **"member function"**

from struct to class

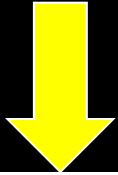


```
class Vector2f
{
    float x, y;
    void Print()
    {
        std::cout << "(" << x << ", " << y << ")" << '\n';
    }
};

int main()
{
    Vector2f v1, v2;
    v1.x = 21.0f;
    v1.Print();
    v2.Print();
}
```

- struct: free external access to data members: **public**
- class: by default no external access to datamembers and member functions: **private**

class > access restricted



```
class Vector2f
{
    float x, y;
    void Print()
    {
        std::cout << "(" << x << ", " << y << ")" << '\n';
    }
};

int main()
{
    Vector2f v1, v2;
    v1.x = 21.0f;
    v1.Print();
    v2.Print();
}
```

ERROR

- struct: free external access to data members: **public**
- class: by default no external access to datamembers and member functions: **private**
- **Why?**
 - protection of member variables
 - abstraction

struct vs class: access

STRUCT

- Default: datamembers and member functions can be directly accessed by externals.
- default **public** access.

CLASS

- Default: datamembers and member functions can **NOT** be directly accessed by externals. Only by other members of the class.
- default **private** access.
- "Encapsulation".

Object orientation: access specifiers

```
class Vector2f
{
public:
    void Print()
    {
        cout << "(" << m_X << ", " << m_Y << ")" << endl;
    }
private:
    float m_X, m_Y; // members are private by default
};

int main()
{
    Vector2f v1, v2;
    //v1.x = 21.0f;
    v1.Print();
    v2.Print();

    cin.get();
}
```

→ ok

- **Access specifiers** allow changes to this default behavior:
 - public, private (and protected see later)
- **Datamembers** should always have **private** access specifier unless you have a very good reason to grant public access. In that case, change the class into a struct.
- **Member functions** can be **public** only if **external access** is required.

Object orientation: access specifiers

```
struct Vector2f
{
    float x, y;
    void Print()
    {
        std::cout << '(' << x << ", " << y << ")" << '\n';
    }
};

int main()
{
    Vector2f v1, v2;
    v1.x = 21.0f;
    v1.Print();
    v2.Print();
}
```

- In this example, a Vector2f is typical a struct and not a class. There is no need to protect the data members x and y from anything.

Object orientation: creating objects

➤ Objects of a class (or struct) can be created in different ways:

➤ Using automatic memory allocation

- We already used this for Point2f and Vector2f structs
- What you can do when creating objects of structs
- Uses stack/global memory
- Are destroyed automatically when going out of scope
- More on this with classes in Programming 2

```
struct Vector2f
{
    float x, y;
};

int main()
{
    Vector2f v1, v2;
    v1.x = 21.0f;
    v1.Print();
    v2.Print();
}
```

➤ Using dynamic memory allocation

- What we will do here, when creating objects of **classes**.

Creating objects using dynamic memory allocation

```
class Cube
{
    private:
        float m_Size;
    public:
        void Print();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

int main()
{
    Cube *pCube{nullptr}; // 1
    pCube = new Cube{}; // 2
    pCube->Print(); // 3
    delete pCube; // 4
    pCube = nullptr; // 5
}
```

- 1: create and initialize a pointer.
 - A pointer can store a memory address of an object.
 - It knows the type of the object it points at.
 - This pointer is initialized with nullptr, meaning it points at nothing.

Creating objects using dynamic memory allocation

```
class Cube
{
    private:
        float m_Size;
    public:
        void Print();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

int main()
{
    Cube *pCube{nullptr}; // 1
    pCube = new Cube{}; // 2
    pCube->Print(); // 3
    delete pCube; // 4
    pCube = nullptr; // 5
}
```

- 2: create an object on the heap and assign its memory address to the pointer pCube.
 - The new operator dynamically allocates heap memory that fits the size of the Cube object.
 - The cube object is default initialized using uniform initialization. Another option is to not use any brackets at all. Don't use parenthesis, it confused the compiler assuming Cube() is a function.
 - The new operator returns the memory address of the object, the assignment stores it in the pointer.

Creating objects using dynamic memory allocation

```
class Cube
{
    private:
        float m_Size;
    public:
        void Print();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

int main()
{
    Cube *pCube{nullptr}; // 1
    pCube = new Cube{}; // 2
    pCube->Print(); // 3
    delete pCube; // 4
    pCube = nullptr; // 5
}
```

- 3: Members of the object can be accessed using the member access operator “>” the member of pointer operator (cppreference)
 - Only public members can be accessed.
 - If they are member functions, you can use them the same way global functions are used.

Creating objects using dynamic memory allocation

```
class Cube
{
    private:
        float m_Size;
    public:
        void Print();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

int main()
{
    Cube *pCube{nullptr}; // 1
    pCube = new Cube{}; // 2
    pCube->Print(); // 3
    delete pCube; // 4
    pCube = nullptr; // 5
}
```

- 4: When the object is no longer needed, it needs to be removed from the memory.
 - Not doing this, creates unreleased memory or a memory leak.
 - Never, ever forget to delete objects

Creating objects using dynamic memory allocation

```
class Cube
{
private:
    float m_Size;
public:
    void Print();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

int main()
{
    Cube *pCube{nullptr}; // 1
    pCube = new Cube{}; // 2
    pCube->Print(); // 3
    delete pCube; // 4
    pCube = nullptr; // 5
}
```

- 5: Setting the pointer back to a neutral nullptr
 - Once the object is **deleted**, the pointer still points at the same memory address the object used to be. This memory address is now **invalid**, turning the pointer into a **dangling pointer**.
 - If one would accidentally perform the **delete operation** with the same **invalid pointer**, it would result in an error.
 - To avoid these issues, always **set** a dangling pointer to a **neutral nullptr**. This allows checking the pointer value for being nullptr or not.

Creating objects using dynamic memory allocation

```
class Cube
{
private:
    float m_Size;
public:
    void Print();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

int main()
{
    Cube *pCube{new Cube{}}; // 1 + 2
    pCube->Print(); // 3
    delete pCube; // 4
    pCube = nullptr; // 5
}
```

- Combining 1 and 2:
 - Perfect shorter alternative

Creating objects using dynamic memory allocation

```
class Cube
{
private:
    float m_Size;
public:
    void Print();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

int main()
{
    Cube *pCube{new Cube{}}; // 1 + 2
    pCube->Print(); // 3
    delete pCube; // 4
    pCube = nullptr; // 5
}
```

- The resulting output: 0 is printed.
- Surprised? You should be. Why?
- m_Size is never initialized.
- It seems that in the current version of visual studio, it initializes variables for you if the object is created using uniform initialisation{} (see new Cube{}). This is not in the C++ standard.

Encapsulation

```
class Cube
{
    private:
        float m_Size;
    public:
        void Print();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

int main()
{
    Cube *pCube{new Cube{}}; // 1 + 2
    pCube->Print(); // 3
    delete pCube; // 4
    pCube = nullptr; // 5
}
```

- Hold on! → How can we **change** the value of **m_Size** if it is private? We have **no access**!
- Indeed there is no way of modifying that value unless we provide a public member function to do this for us.

Encapsulation: mutator or setter

```
class Cube
{
    private:
        float m_Size;
    public:
        void Print();
        void SetSize(float size);
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

void Cube::SetSize(float size)
{
    if (size > 0) m_Size = size;
}

int main()
{
    Cube *pCube{new Cube{}};
    pCube->SetSize(10);
    delete pCube;
    pCube = nullptr;
}
```

- Hold on! → How can we **change** the value of **m_Size** if it is private? We have **no access**!
- Indeed there is no way of modifying that value unless we provide a public member function to do this for us.
- These kind of member functions are called **mutators** or **setter** public member functions.
- Their name generally starts with Set. In our case here: SetSize
- There typically is **one parameter**, and **no return value**. The parameter type is the same type as the value we want to change:
 - void SetSize(float size);
- The **purpose** is to **protect** the variable from the **outside world**, to prevent it from getting illegal values, such as negative values in this example. (a cube with negative size makes no sense)

Encapsulation: mutator or setter

```
class Cube
{
    private:
        float m_Size;
    public:
        void Print();
        void SetSize(float size);
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

void Cube::SetSize(float size)
{
    if (size > 0) m_Size = size;
}

int main()
{
    Cube *pCube{new Cube{}};
    pCube->SetSize(10);
    delete pCube;
    pCube = nullptr;
}
```

- Do I always need a mutator? NO
- It's up to you as designer of the class to decide if that variable should have a mutator function.
- Don't provide a mutator if there is no need to.

Encapsulation

```
class Cube
{
    private:
        float m_Size;
    public:
        void Print();
        void SetSize(float size);
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

void Cube::SetSize(float size)
{
    if (size > 0) m_Size = size;
}

int main()
{
    Cube *pCube{new Cube{}};
    pCube->SetSize(10);
    delete pCube;
    pCube = nullptr;
}
```

- Hold on! → How can we **retrieve** the value of **m_Size** if it is private? We have **no access**!
- Indeed there is no way of knowing that value unless we provide a public member function to do this for us.

Encapsulation: accessor or getter

```
class Cube
{
private:
    float m_Size;
public:
    void Print();
    void SetSize(float size);
    float GetSize();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

void Cube::SetSize(float size)
{
    if (size > 0) m_Size = size;
}

float Cube::GetSize()
{
    return m_Size;
}

int main()
{
    Cube *pCube{new Cube{}};
    pCube->SetSize(10);
    float size{pCube->GetSize()};
    delete pCube;
    pCube = nullptr;
}
```

- Hold on! → How can we **retrieve** the value of **m_Size** if it is private? We have **no access**!
- Indeed there is no way of knowing that value unless we provide a public member function to do this for us.
- These kind of member functions are called **accessors** or **getter** public member functions.
- Their name generally starts with Get. In our case here: GetSize
- There typically is no parameter, and a return value. The return type is the same type as the value we want to retrieve:
 - float GetSize();
- The **purpose** is to **protect** the variable from the outside world, so in most cases a **copy** is returned.

Encapsulation: accessor or getter

```
class Cube
{
private:
    float m_Size;
public:
    void Print();
    void SetSize(float size);
    float GetSize();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

void Cube::SetSize(float size)
{
    if (size > 0) m_Size = size;
}

float Cube::GetSize()
{
    return m_Size;
}

int main()
{
    Cube *pCube{new Cube{}};
    pCube->SetSize(10);
    float size{pCube->GetSize()};
    delete pCube;
    pCube = nullptr;
}
```

- Do I always need an accessor? NO
- It's up to you as designer of the class to decide if the value of that variable should be available.
- Only provide an accessor when there is a need to do so.

Object orientation: Encapsulation

➤ Public interface

- The collection of public member functions and public members is called the "Public interface". It is what the user of the class needs to know.

➤ Encapsulaton

- Details about how an object is implemented are hidden away from users of the object.
- Access through the public interface.

➤ Access specifiers

- Data members are always private unless there is a very good reason to make them public
- member functions can be private (preferred) or public if outside access is required.

Encapsulation, why?

- Benefit: **encapsulated classes** are easier to use and **reduce the complexity** of your programs
 - **Your class becomes easier to use because elements that shouldn't be used are inaccessible**
- Benefit: encapsulated classes help protect your data and prevent misuse
 - **Other programmers who use your class can't accidentally change the value of your data members to something that makes no sense in the program**
- Benefit: Separation
 - **The behavior of the class is separated from the specific implementation of the class**
- Benefit: encapsulated classes are **easier to change**
- Benefit: encapsulated classes are **easier to debug**

Encapsulation: recap

```
class Cube
{
    private:
        float m_Size;
    public:
        void Print();
        void SetSize(float size);
        float GetSize();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

void Cube::SetSize(float size)
{
    if (size > 0) m_Size = size;
}

float Cube::GetSize()
{
    return m_Size;
}
```

- Data members are private
- Create member functions to grant access to data members if needed:
 - **accessor** (getter)
 - **mutator** (setter)
- Protect data members from getting illegal values.

Constructor member function

```
class Cube
{
private:
    float m_Size;
public:
    void Print();
    void SetSize(float size);
    float GetSize();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

void Cube::SetSize(float size)
{
    if (size > 0) m_Size = size;
}

float Cube::GetSize()
{
    return m_Size;
}

int main()
{
    Cube *pCube{new Cube{}};
    pCube->SetSize(10);
    float size{pCube->GetSize()};
    delete pCube;
    pCube = nullptr;
}
```

- Look at the main function:
- I need a mutator to set the size to a value.
 - That exposes the m_Size variable. It is possible to **modify** it once the cube is **created**.
- **What if I do not want that?**

Constructor member function

```
class Cube
{
private:
    float m_Size{10.f};
public:
    void Print();
    float GetSize();
};

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

float Cube::GetSize()
{
    return m_Size;
}

int main()
{
    Cube *pCube{new Cube{}};
    float size{pCube->GetSize()};
    delete pCube;
    pCube = nullptr;
}
```

- Look at the main function:
- I need a mutator to set the size to a value.
 - That exposes the m_Size variable. It is possible to **modify** it once the cube is **created**.
- **What if I do not want that?**
- One option is to **initialize the variable** while **declaring** it.

Constructor member function

```
class Cube
{
private:
    float m_Size;
public:
    Cube(float size);
    void Print();
    float GetSize();
};

Cube::Cube(float size):m_Size{size}
{
}

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

float Cube::GetSize()
{
    return m_Size;
}

int main()
{
    Cube *pCube{new Cube{10,f}};
    float size{pCube->GetSize()};
    delete pCube;
    pCube = nullptr;
}
```

- Look at the main function:
- I need a mutator to set the size to a value.
 - That exposes the m_Size variable. It is possible to **modify** it once the cube is **created**.
- **What if I do not want that?**
- One option is to **initialize the variable** while **declaring** it.
- The other option is to define a constructor member function with **a member initializer list**.
 - The list separates variables using a comma
- ERROR? A default constructor is not generated if a constructor is provided. → default constructor?

Constructor member function

- Constructor is a special member function
 - Has **same name** as the class
 - Has **no return type**
 - Can be **overloaded**.
 - Can **not be called directly**
 - Is **called automatically** when an object **is instantiated (created)**.
- Default constructor
 - is a constructor **without parameters** (or default parameters)
 - is **generated automatically by the compiler if no other constructor function is provided**
 - Yes, the compiler has generated a default constructor for the Point2f struct!
 - It does **NOT auto-initialize** the variables.

Constructor member function

```
class Cube
{
private:
    float m_Size;
public:
    Cube(float size);
    Cube(const Color4f& color);
    void Print();
    float GetSize();
};
```

➤ Constructors can be overloaded.

Constructor member function

```
class Cube
{
private:
    float m_Size;
public:
    Cube(float size = 10);
    Cube(const Color4f& color);
    void Print();
    float GetSize();
};
```

- Constructors can be overloaded.
- Or use default parameters

Constructor overloading

➤ Default constructor

- is a constructor without parameters

➤ Constructor overloading

- are allowed -> with parameters
- be careful -> can be ambiguous

```
class MyClass
```

```
{
```

```
    MyClass();
```

```
    MyClass(float x, float y);
```

```
    MyClass(const Point2f& pos);
```

```
};
```

Constructors: auto generated, no initialization

- The main task of a constructor is to initialize the data members of the class: get the object ready to be used.
- a **constructor** is always **called** when an object is **instantiated**.
- If **no constructor** is defined, the **compiler** generates a default constructor.
- This generated constructor does **not initialize the datamembers (!)**
 - This results in **undefined values** of all the data members. Unless they are initialized when defined.
- It is recommended to always add an explicit defined constructor.

Constructor member initializer list

```
class Cube
{
private:
    const float m_Size;
public:
    Cube(float size);
    void Print();
    float GetSize();
};

Cube::Cube(float size):m_Size{size}
{
}

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

float Cube::GetSize()
{
    return m_Size;
}

int main()
{
    Cube *pCube{new Cube{10,f}};
    float size{pCube->GetSize()};
    delete pCube;
    pCube = nullptr;
}
```

- Member initializer lists also allow **consts** or **references** to be initialized.

Constructor member initializer list

```
class Cube
{
private:
    const float m_Size;
public:
    Cube(float size);
    void Print();
    float GetSize();
};

Cube::Cube(float size)
{
    m_Size = size; // not efficient
}

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

float Cube::GetSize()
{
    return m_Size;
}

int main()
{
    Cube *pCube{new Cube{10,f}};
    float size{pCube->GetSize()};
    delete pCube;
    pCube = nullptr;
}
```

- Do not initialize members in the **body of the constructor!** This takes more time and is bad practice.

Destructor

```
class Cube
{
private:
    const float m_Size;
public:
    Cube(float size);
    ~Cube();
    void Print();
};

Cube::Cube(float size)
{
    m_Size = size;
}

Cube::~~Cube()
{
    std::cout << "I am being deleted.\n" ;
}

void Cube::Print()
{
    std::cout << m_Size << '\n';
}

int main()
{
    Cube *pCube{new Cube{10,f}};
    float size{pCube->GetSize()};
    delete pCube;
    pCube = nullptr;
}
```

- Is a special member function with the **same name** as the **class** preceded by a **~** sign and with **no return type**.
- Is **generated automatically** when none is provided.
- **Executes automatically** when the object is about to be **destroyed**.
- Its purpose is to **clean up resources and free memory**. See later for a more practical example.
- **Implement only if the class has heap memory or other resources to release. (e.g. dyn array)**
 - Don't add an empty destructor.

Methods or Member functions

- Are functions that belong to a class.
- Terminology:
 - “Global Function”: isn’t part of a class: global scope.
 - “Member Function” or “method”: function that is part of a class.
- Determine what an object of a class can do.
- Will typically change data members or do something with their values.

Methods or Member functions

- Have access to the following variables:
 - Their own local variables.
 - All data members of the object.
 - All data members of **other** objects of the same class (!) even if they are private

```
void Player::GetInfo(const Player& other)
{
    m_Info = other.m_Info;
}
```

Methods or Member functions

- Can be public or private:
 - Only public if outside access is needed. (outside being: not from the same class)
 - **Private** is the **default** you should choose.
 - Choose only **public** if it is really **needed**, such as mutator and accessors.
- The collection of public member functions is also called the “**public interface**” of a class.

Procedure when designing a class

You'll typically follow these steps when making a new class:

- Decide on the data members
- Add the constructor(s) initializing the data members
- Decide on the need of having to write the destructor.
- Add the member functions

Practical: Coding conventions

- For classes only: member variables start with the prefix "m_" followed by a capital letter.

```
struct Vector2f
{
    float x, y;
    void Print()
    {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};
```

```
class Cube
{
    private:
        float m_Size;
    public:
        Cube(float size = 10) : m_Size{ size }{}
        float GetSize()
        {
            return m_Size;
        }
};
```

Practical: Separate the declaration and the implementation

- declaration: a **header** file with the extension **h**

```
#include <string>
class Player
{
public:
    Player(const std::string& name);
    ~Player(); // destructor
    void Print();
private:
    std::string m_Name;
};
```

- implementation: a **source** file with the extension **cpp**
 - prefix the class name to the function using the
 - **scope resolution operator (::)**

```
#include "stdafx.h"
#include "Player.h"

Player::Player(const std::string& name)
: m_Name{ name }
{
    std::cout << "Player " << m_Name << " constructor fired.\n";
}

Player::~~Player()
{
    std::cout << m_Name << ": Destructor fired.\n";
}

void Player::Print()
{
    std::cout << m_Name << "\n";
}
```


Using one class as datamember of another class

- class **forward declaration** in the header(.h) file if possible: let the compiler know there is a class with that name, do that it can process the pointer declaration.
- use the **#include statement** in the source (.cpp) file: the compiler needs to know what the constructor parameters are.

```
class Cube; // class forward declaration

class Shapes
{
public:
    Shapes();

private:
    Cube* m_pCube;
};
```

```
#include "Cube.h"

#include "Shapes.h"

Shapes::Shapes() : m_pCube{ new Cube(10) }
{
}
```

Class Diagrams (UML)

- What: a clear and concise way of specifying the contents of a class.
- Also: a clear and concise way to give a programmer instructions for what class he is supposed to write, and what is supposed to be in it.

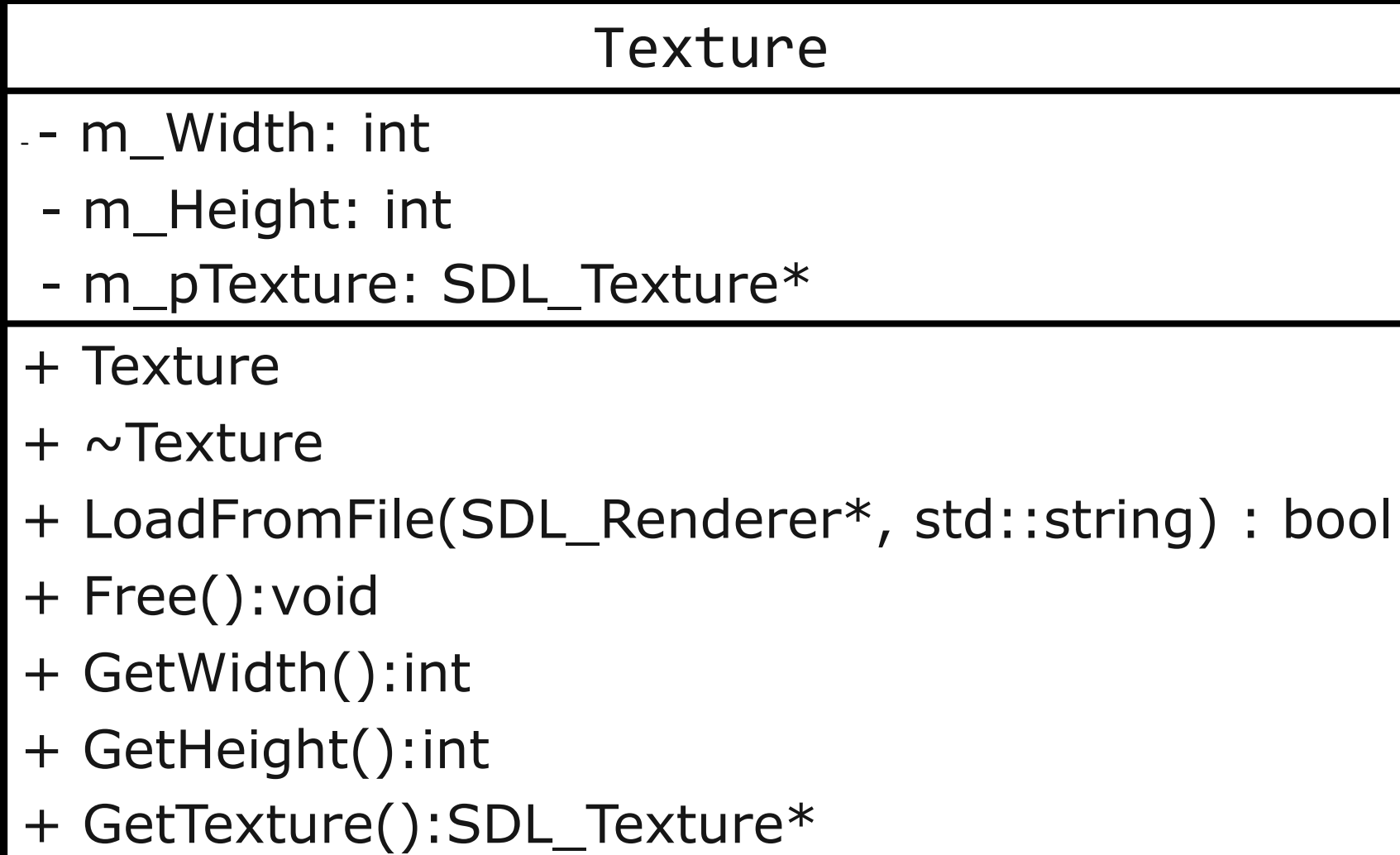
Class Diagrams (UML)

Texture

```
- m_Width: int
- m_Height: int
- m_pTexture: SDL_Texture*

+ Texture
+ ~Texture
+ LoadFromFile(SDL_Renderer*, std::string) : bool
+ Free():void
+ GetWidth():int
+ GetHeight():int
+ GetTexture():SDL_Texture*
```

Class Diagrams (UML)

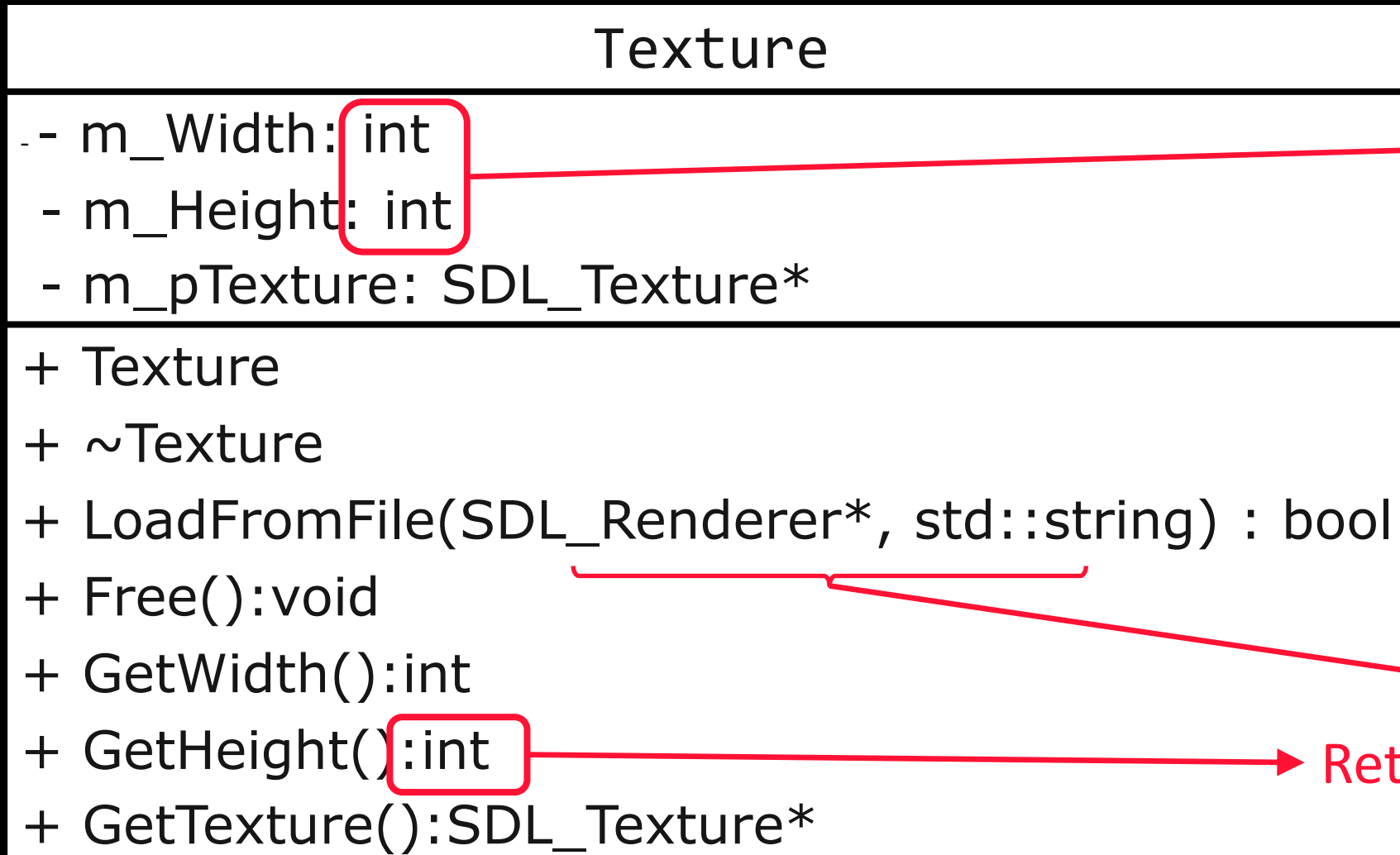


← Name of the class

← Datamembers

← Methods

Class Diagrams (UML)



type of the variable

Parameter types

Return type of the function

References:

<http://www.learncpp.com/cpp-tutorial/81-welcome-to-object-oriented-programming/>

chapter 8 sections 1,2,3,4,5,6,7,8,9

http://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm