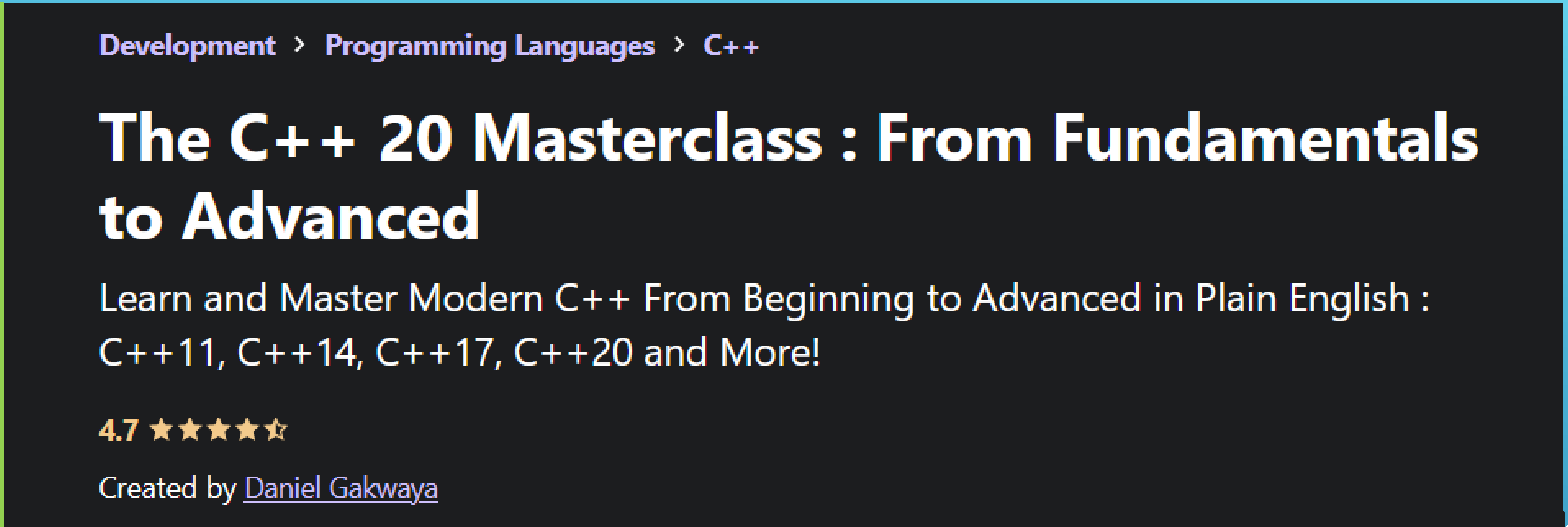
Slides


Development > Programming Languages > C++

The C++ 20 Masterclass : From Fundamentals to Advanced

Learn and Master Modern C++ From Beginning to Advanced in Plain English : C++11, C++14, C++17, C++20 and More!

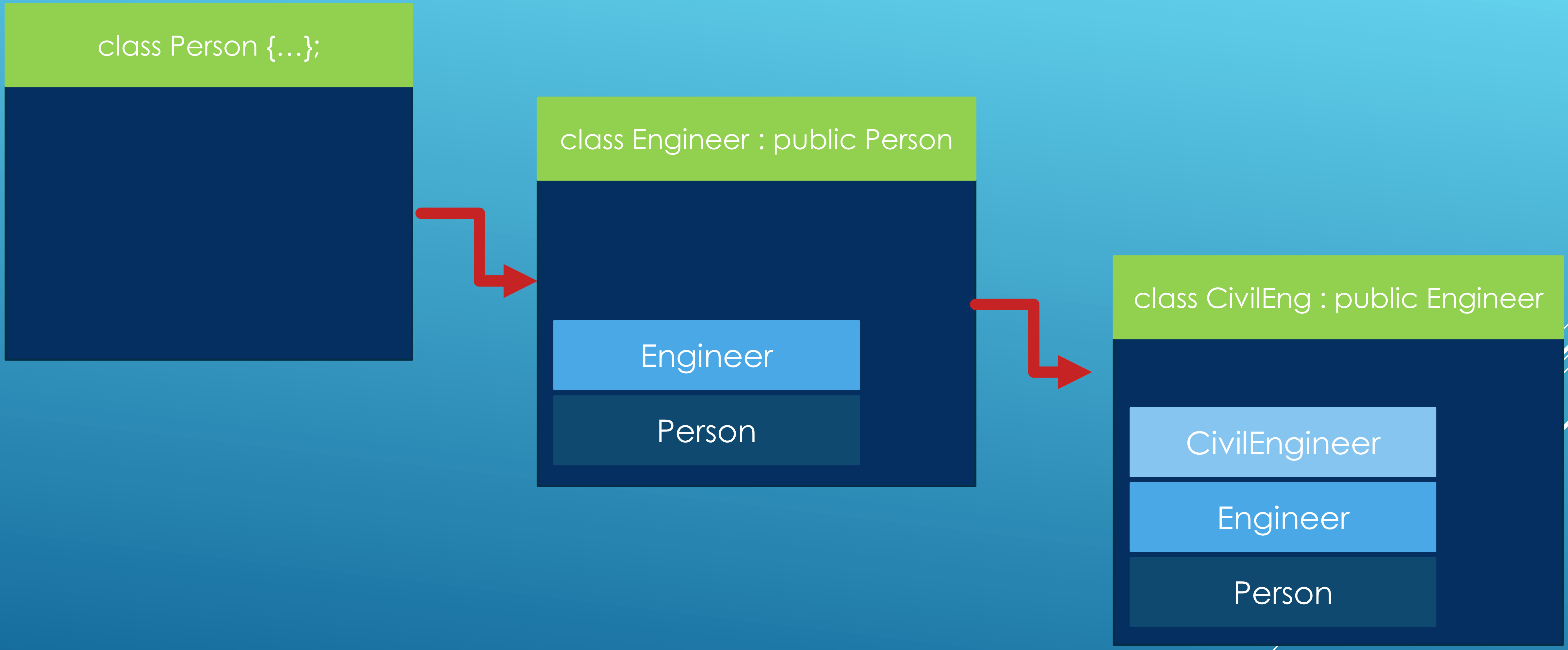
4.7 ★★★★☆

Created by Daniel Gakwaya

Section : Inheritance

Slide intentionally left empty

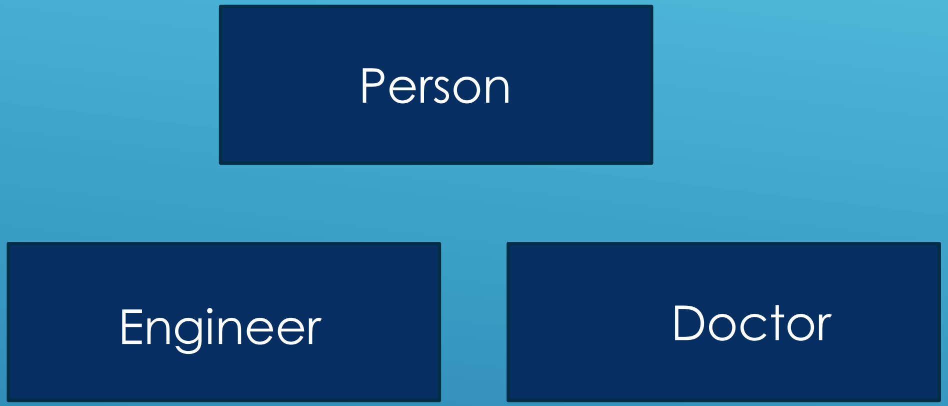
Inheritance

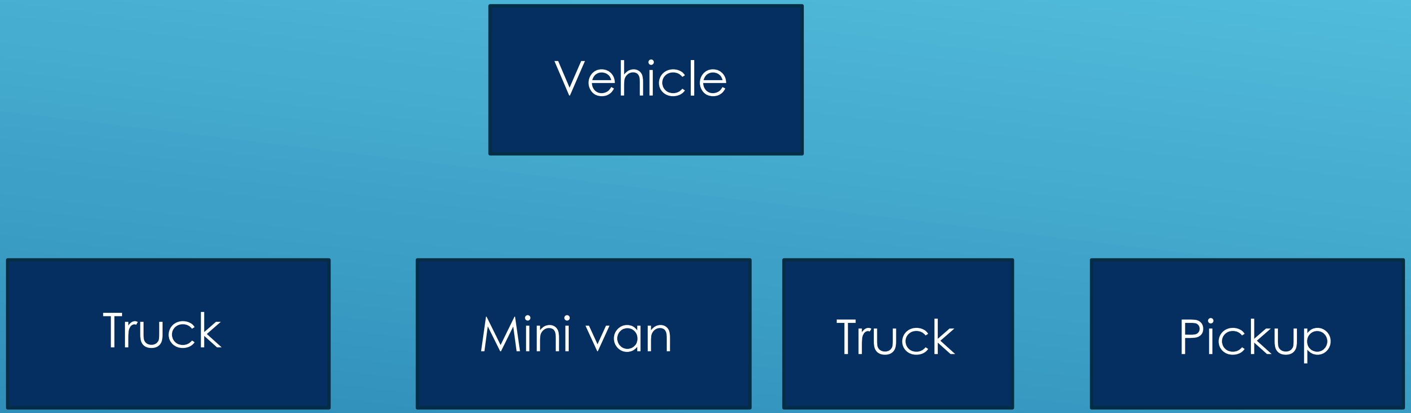

class Person {…};
class Engineer : public Person
Engineer
Person
class CivilEng : public Engineer
CivilEngineer
Engineer
Person

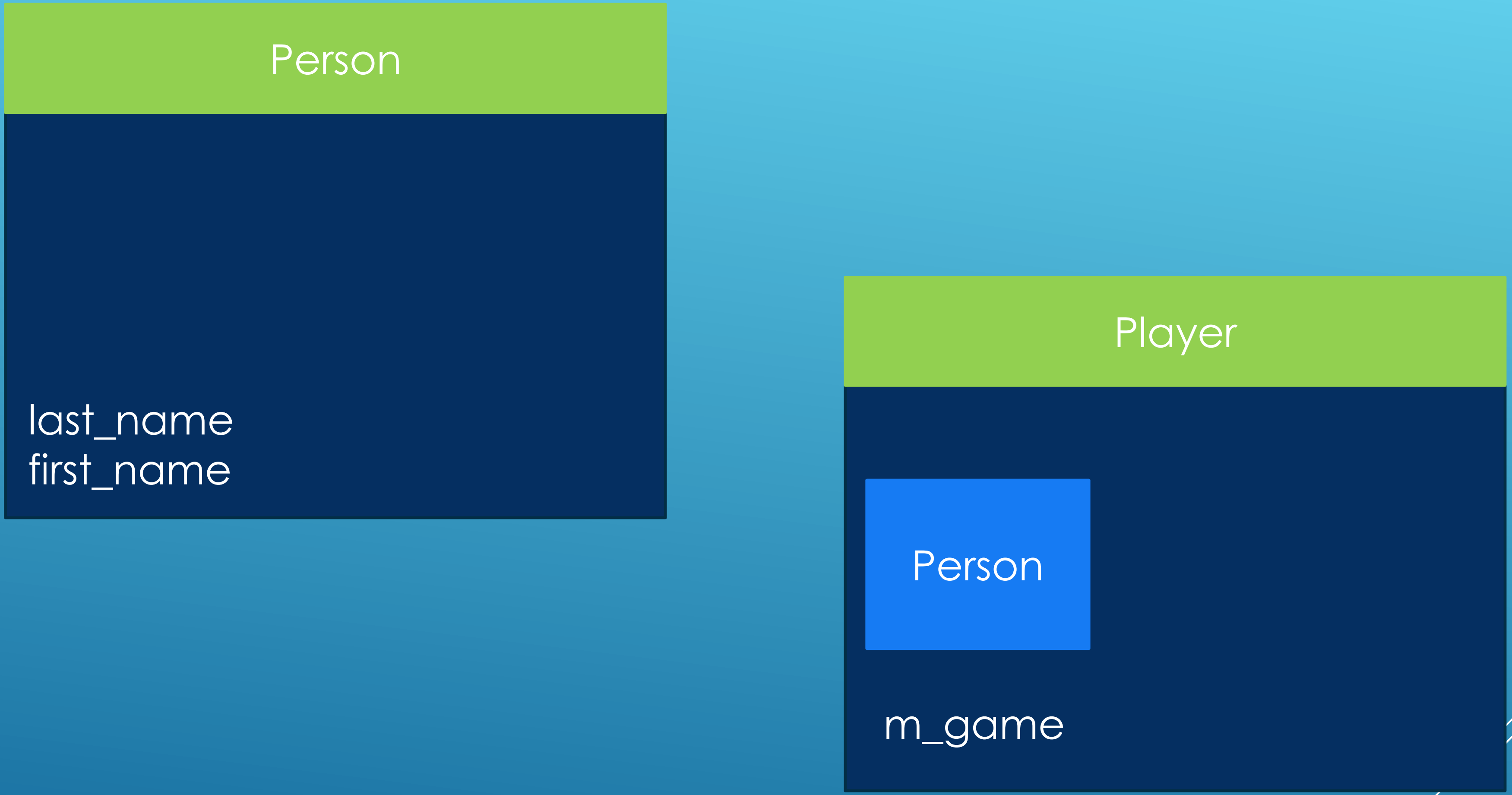
- A defining feature of Object Oriented Programming in C++
- Building types on top of other types
- Inheritance hierarchies can be set up to suit your needs
- Code reuse is improved

Slide intentionally left empty

Your First try on Inheritance


Person
Engineer
Doctor


Vehicle
Truck
Mini van
Truck
Pickup


Person
last_name
first_name
Player
Person
m_game

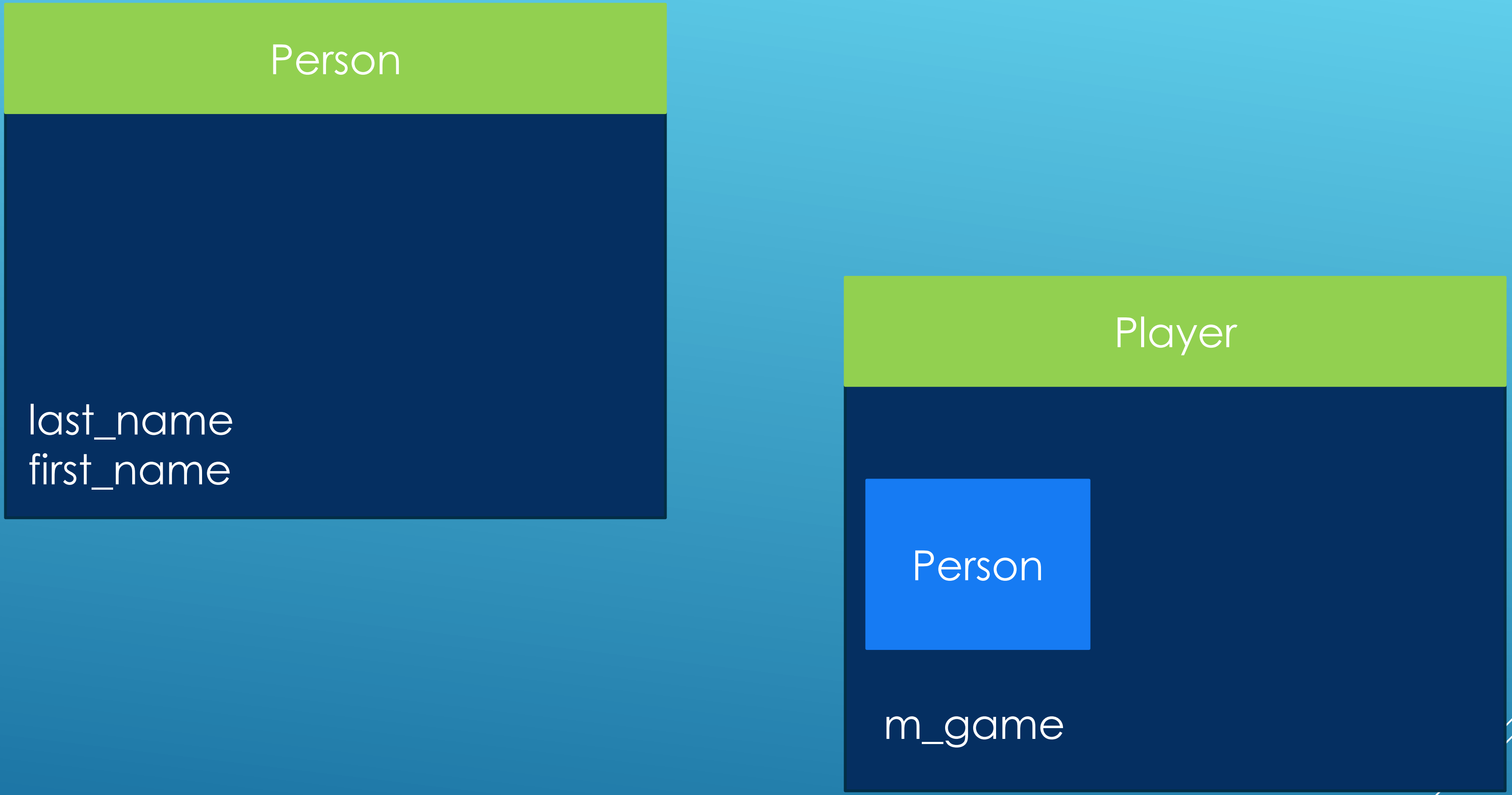
```
class Person
{
    friend std::ostream& operator<<(std::ostream& out, const Person& person);
public:
    Person();
    Person(std::string first_name_param, std::string last_name_param);
    ~Person();
private :
    std::string first_name{"Mysterious"};
    std::string last_name{"Person"};
};
```

```
//#include "person.h"
class Person; // Forward declaration

class Player : public Person
{
    friend std::ostream& operator<<(std::ostream& out, const Player& player);
public:
    Player() = default;
    Player(std::string game_param);
    ~Player();

private :
    std::string m_game{"None"};
};
```

- With public inheritance , derived classes can access and use public members of the base class, but the derived class can't directly access private members
- The same also applies to friends of the derived class. They have access to private members of derived, but don't have access to the base class

Slide intentionally left empty

Protected members


Person
last_name
first_name
Player
Person
m_game

```cpp
class Person
{
    friend std::ostream& operator<<(std::ostream& out, const Person& person);
public:
    Person();
    Person(std::string first_name_param, std::string last_name_param);
    ~Person();
private :
    std::string first_name{"Mysterious"};
    std::string last_name{"Person"};
};
```

```cpp
class Person
{
    friend std::ostream& operator<<(std::ostream& out, const Person& person);
public:
    Person();
    Person(std::string first_name_param, std::string last_name_param);
    ~Person();

protected :
    std::string first_name{"Mysterious"};
    std::string last_name{"Person"};
};
```

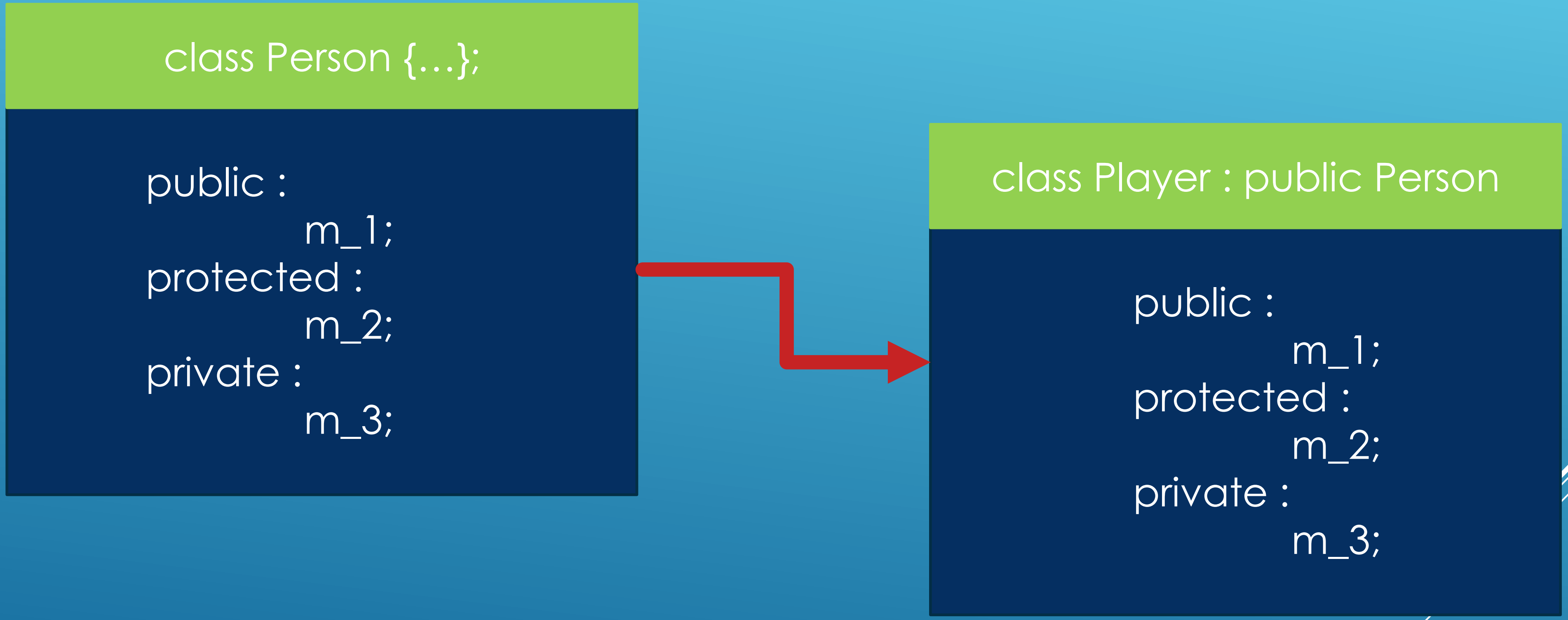
```cpp
//#include "person.h"
class Person; // Forward declaration

class Player : public Person
{
    friend std::ostream& operator<<(std::ostream& out, const Player& player);
public:
    Player() = default;
    Player(std::string game_param);
    ~Player();
private :
    std::string m_game{"None"};
};
```

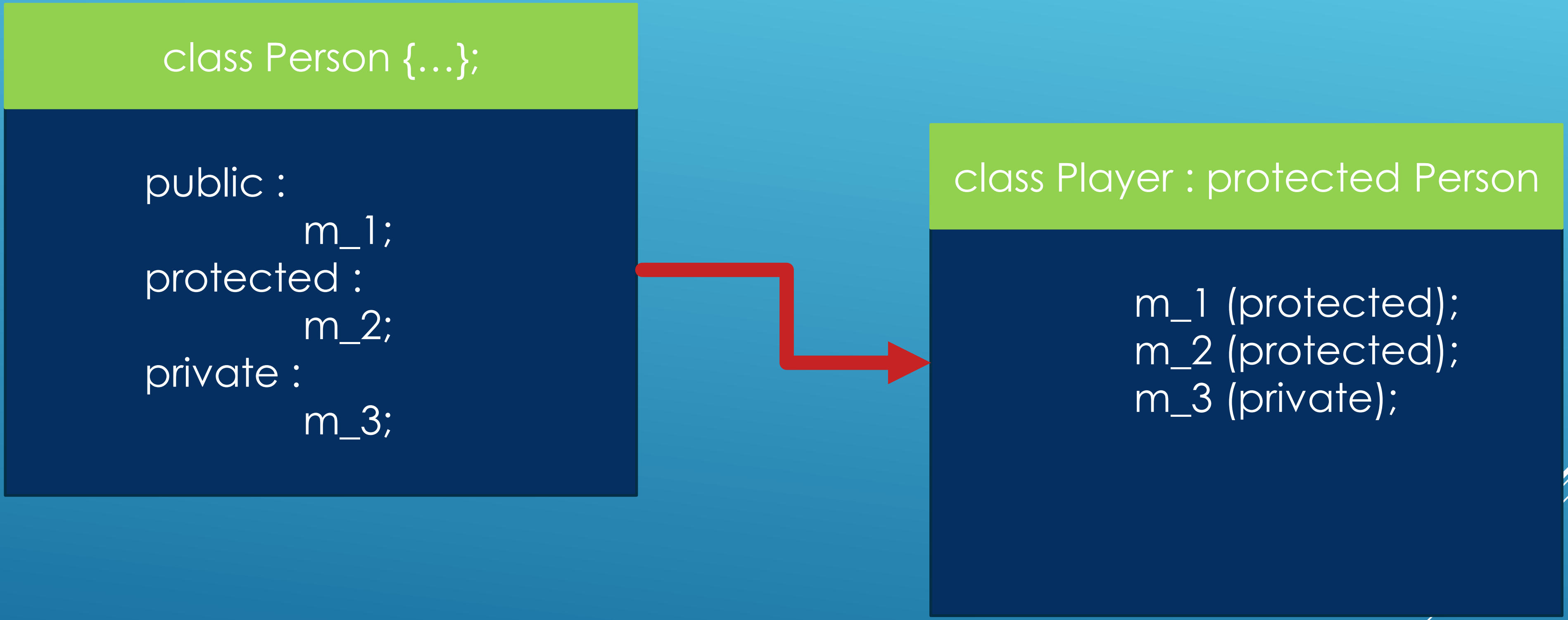
Slide intentionally left empty

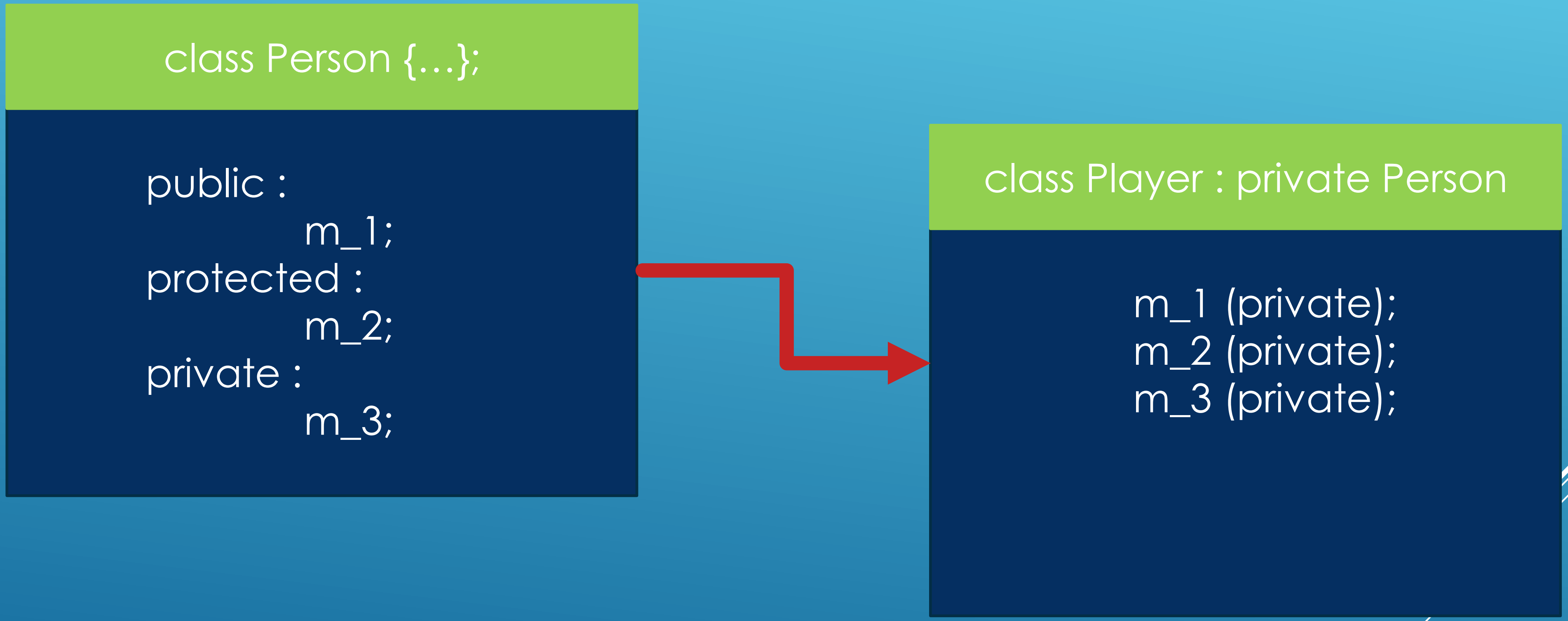
Base class access specifiers : Zooming in

Base access specifier

```cpp
//#include "person.h"
class Person; // Forward declaration

class Player : public Person
{
    friend std::ostream& operator<<(std::ostream& out, const Player& player);
public:
    Player() = default;
    Player(std::string game_param);
    ~Player();

private :
    std::string m_game{"None"};
};
```


class Person {…};
public :
m_1;
protected :
m_2;
private :
m_3;
class Player : public Person
public :
m_1;
protected :
m_2;
private :
m_3;

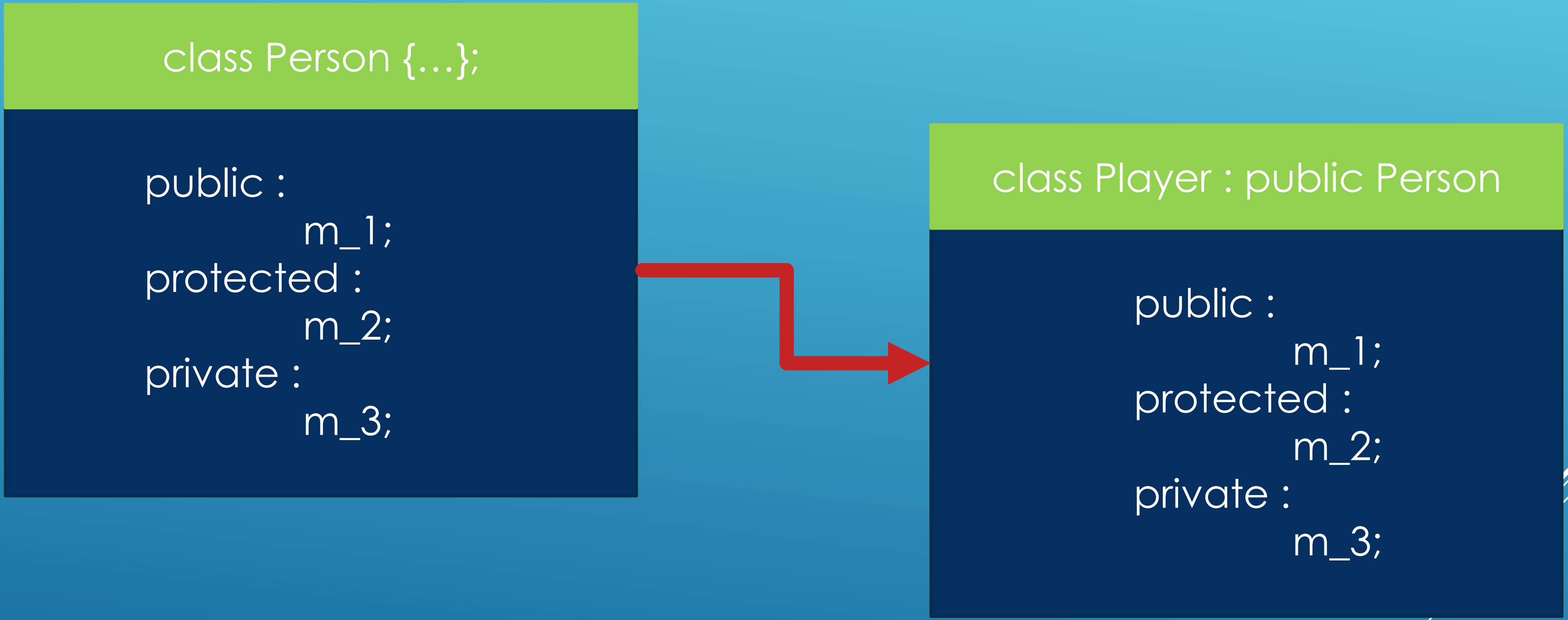

class Person {…};
public :
m_1;
protected :
m_2;
private :
m_3;
class Player : protected Person
m_1 (protected);
m_2 (protected);
m_3 (private);

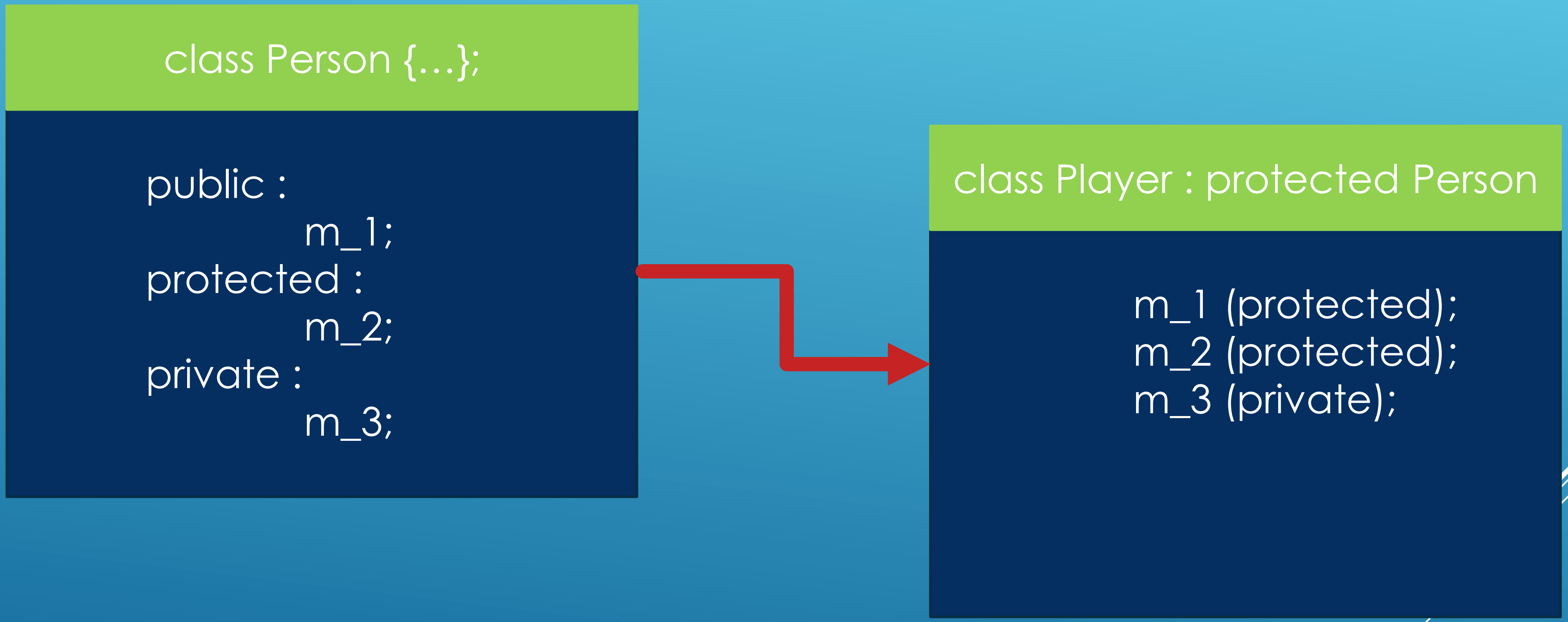

class Person {…};
public :
m_1;
protected :
m_2;
private :
m_3;
class Player : private Person
m_1 (private);
m_2 (private);
m_3 (private);

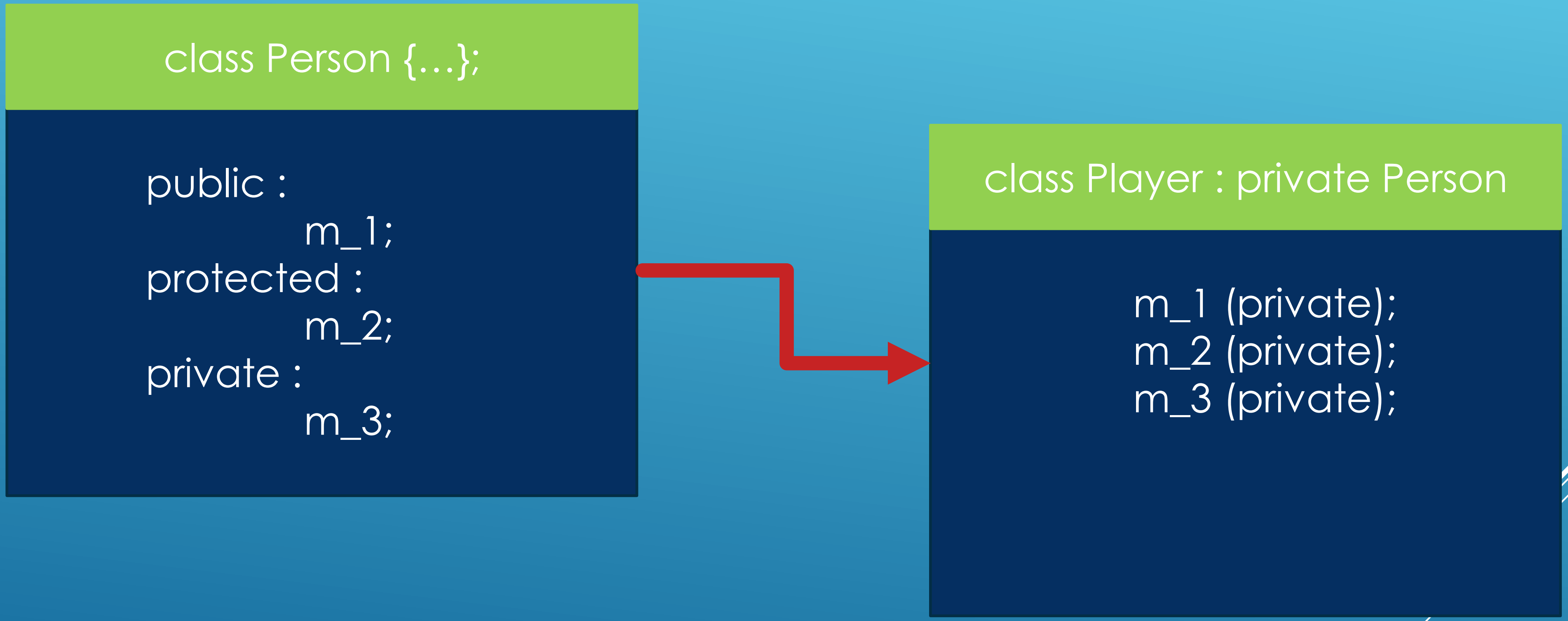
- Through the base class access specifier, we can control how relaxed or constrained is the access of base class members from the derived class.
- Regardless of the access specifier, private members of base class are never accessible from derived classes

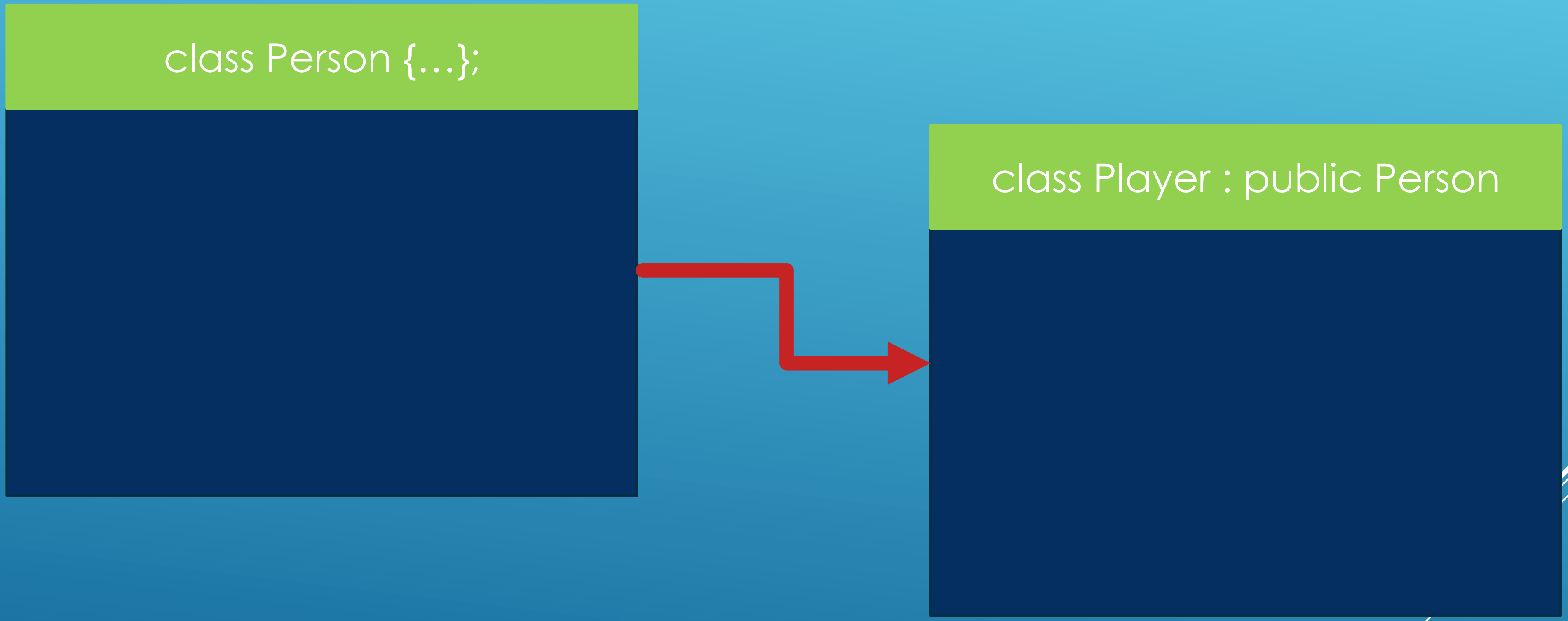
Slide intentionally left empty

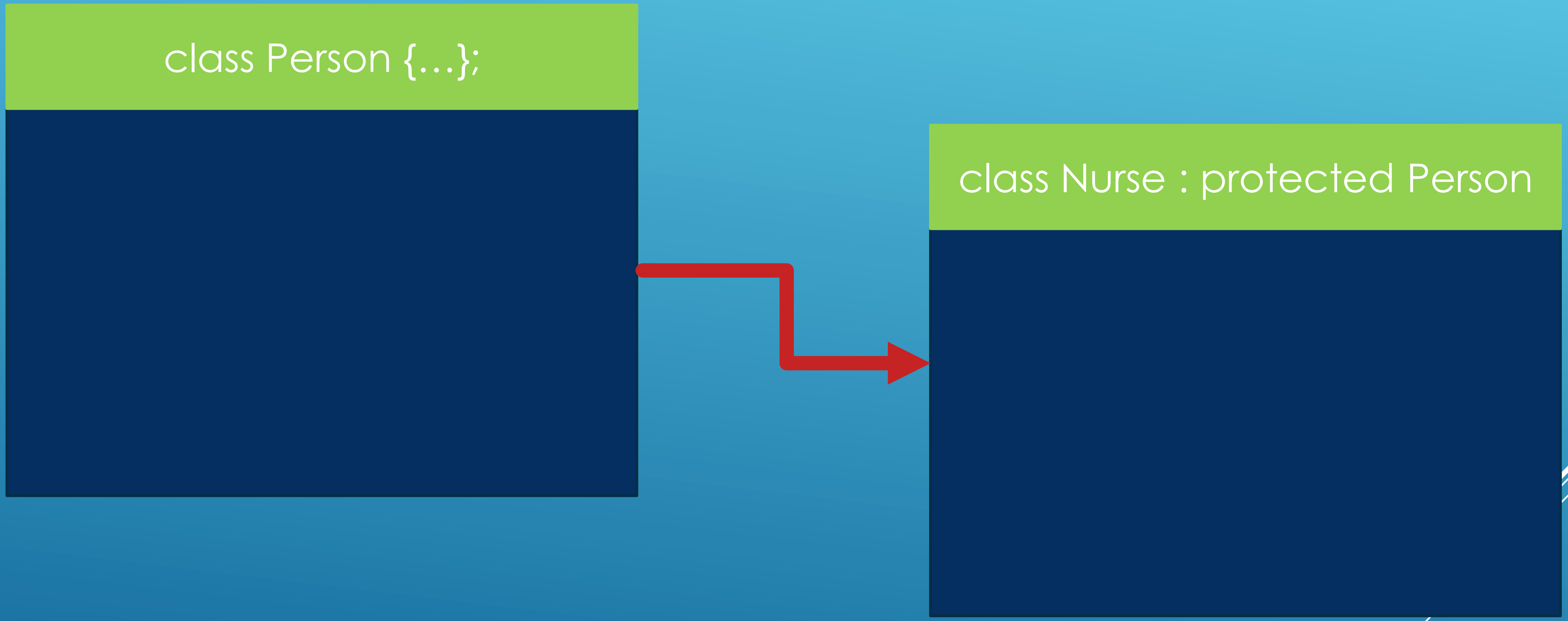
Base class access specifiers : A demo

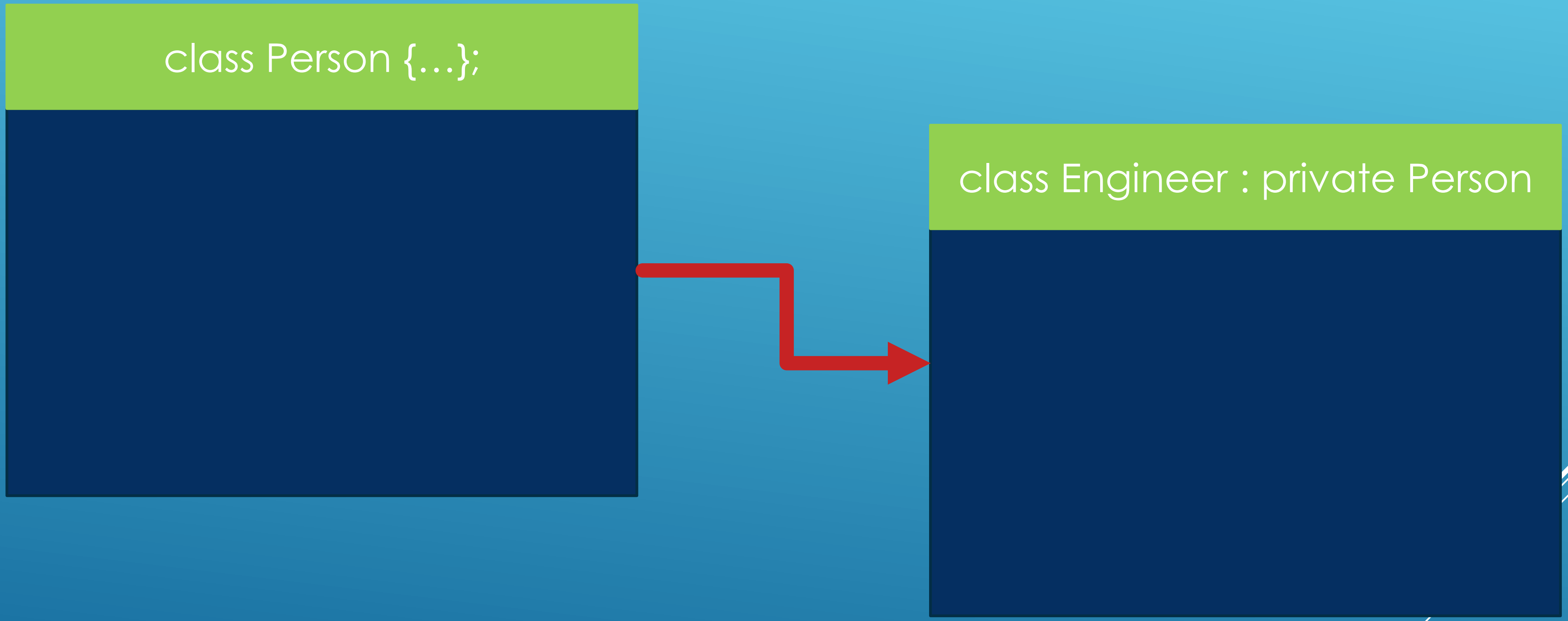

class Person {…};
public :
m_1;
protected :
m_2;
private :
m_3;
class Player : public Person
public :
m_1;
protected :
m_2;
private :
m_3;


class Person {…};
public :
m_1;
protected :
m_2;
private :
m_3;
class Player : protected Person
m_1 (protected);
m_2 (protected);
m_3 (private);


class Person {…};
public :
m_1;
protected :
m_2;
private :
m_3;
class Player : private Person
m_1 (private);
m_2 (private);
m_3 (private);

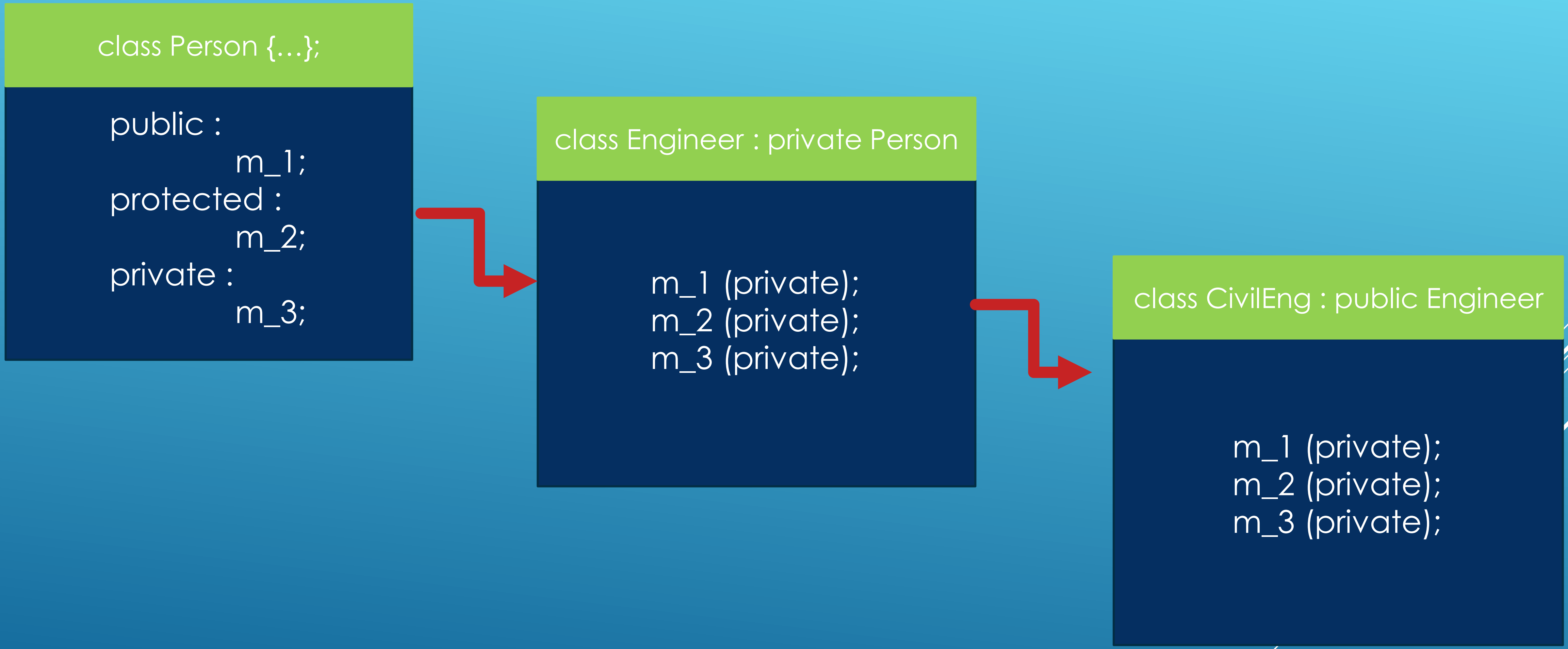

class Person {…};
class Player : public Person


class Person {…};
class Nurse : protected Person


class Person {…};
class Engineer : private Person

Slide intentionally left empty

Closing in on private inheritance


class Person {…};
public :
m_1;
protected :
m_2;
private :
m_3;
class Engineer : private Person
m_1 (private);
m_2 (private);
m_3 (private);
class CivilEng : public Engineer
m_1 (private);
m_2 (private);
m_3 (private);

Person

```
class Person
{
    friend std::ostream& operator<<(std::ostream& , const Person& person);
public:
    Person() = default;
    Person(const std::string& fullname,int age,
    const std::string address);
    ~Person();
public:
    std::string m_full_name{"None"};
protected:
    int m_age{0};
private :
    std::string m_address{"None"};
};
```

Engineer

```
class Person; // Forward declaration
class Engineer : private Person
{
friend std::ostream& operator<<(std::ostream& out , const Engineer& operand);
public:
    Engineer();
    ~Engineer();

    void build_something(){







    }
protected :
    int contract_count{0};
};
```

CivilEngineer

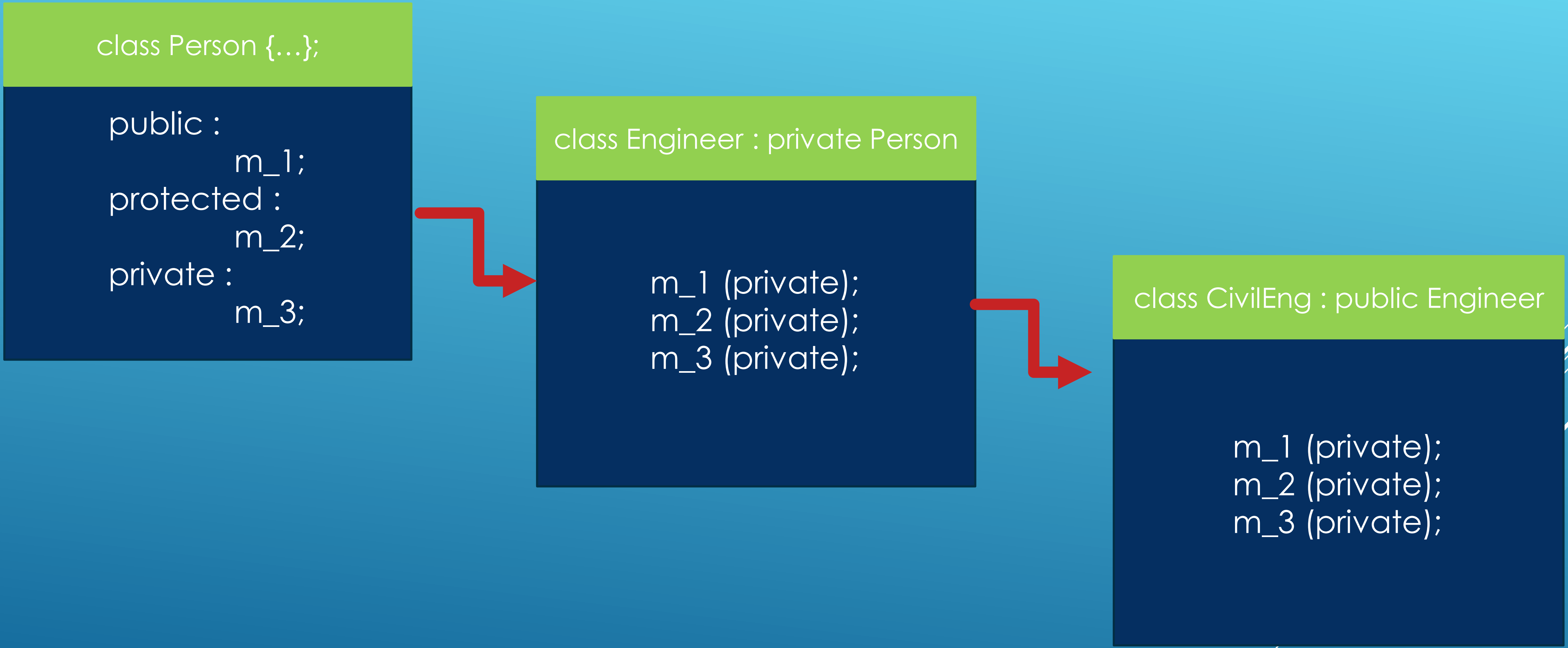
```cpp
class Engineer;
class CivilEngineer : public Engineer
{
    friend std::ostream& operator<<(std::ostream&, const CivilEngineer& operand);
public:
    CivilEngineer();
    ~CivilEngineer() ;

    void build_road(){







    }

private :
    std::string m_speciality{"None"};

};
```

Forward declarations in action

```
#include "person.h"
#include "engineer.h"
#include "civilengineer.h"

int main(int argc, char **argv)
{
    CivilEngineer ce;
    std::cout << "ce : " << ce << std::endl;

    return 0;
}
```

Slide intentionally left empty

Resurrecting members back in scope



```
class Person {…};

public :
        m_1;
protected :
        m_2;
private :
        m_3;
```

```
class Engineer : private Person

m_1 (private);
m_2 (private);
m_3 (private);
```

```
class CivilEng : public Engineer

m_1 (private);
m_2 (private);
m_3 (private);
```

Engineer

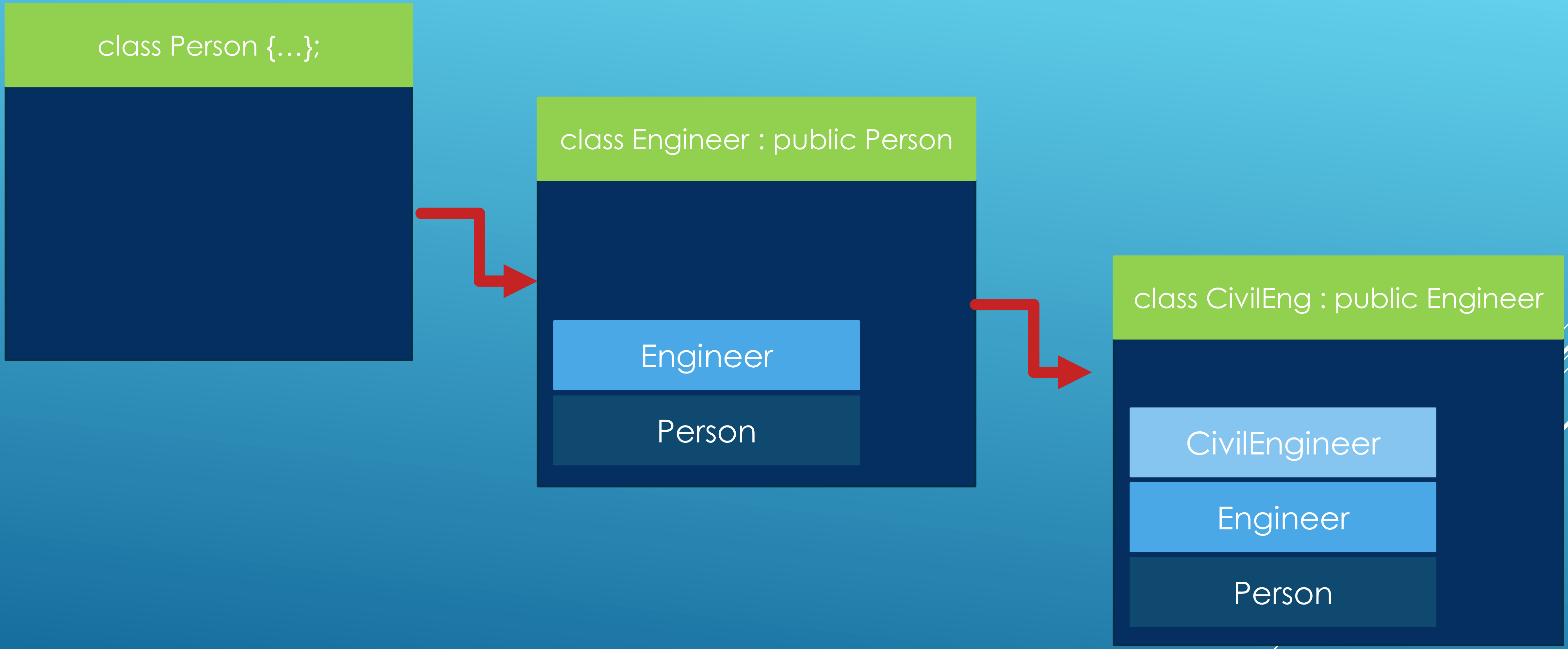
```
class Person
{
    friend std::ostream& operator<<(std::ostream& , const Person& person);
public:
    Person() = default;
    Person(const std::string& fullname,int age,
    const std::string address);
    ~Person();
public:
    std::string m_full_name{"None"};
protected:
    int m_age{0};
private :
    std::string m_address{"None"};
};
```

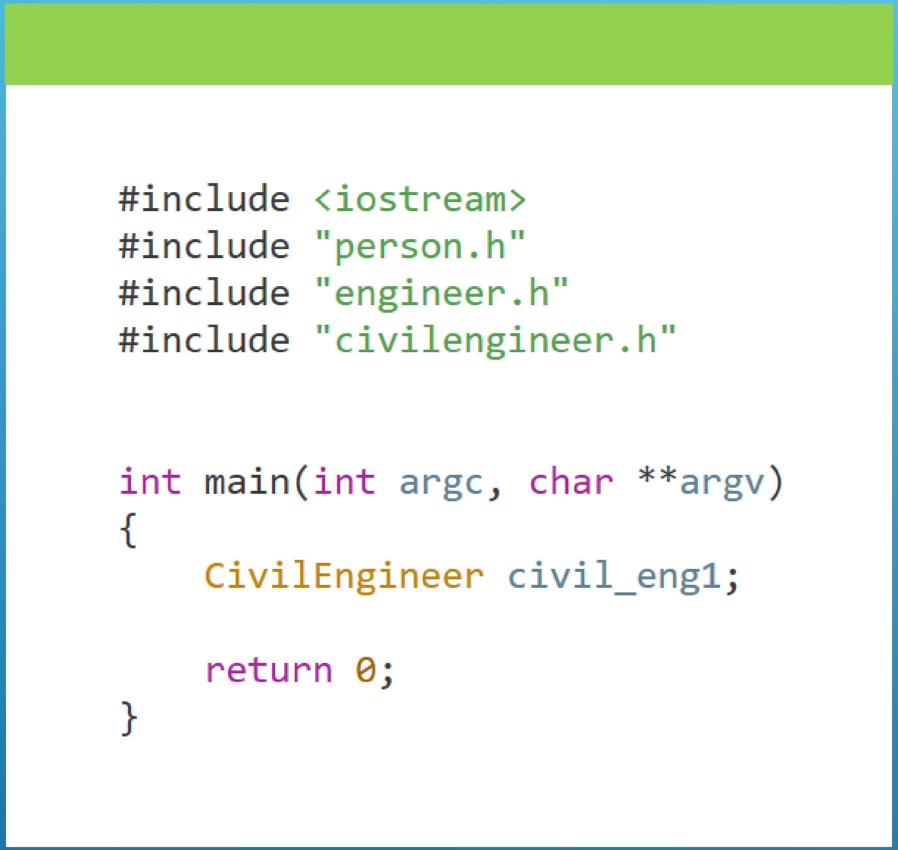
Engineer

```
class Person; // Forward declaration
class Engineer : private Person
{
friend std::ostream& operator<<(std::ostream& out , const Engineer& operand);
public:
    Engineer();
    ~Engineer();

protected :
     using Person::get_full_name;
     using Person::get_age;
     using Person::get_address;

public :
    using Person::m_full_name;
    //using Person::m_address; // Compiler error.
    using Person::add_numbers; // Resurect back to public access
protected :
    int contract_count{0};
};
```

Slide intentionally left empty

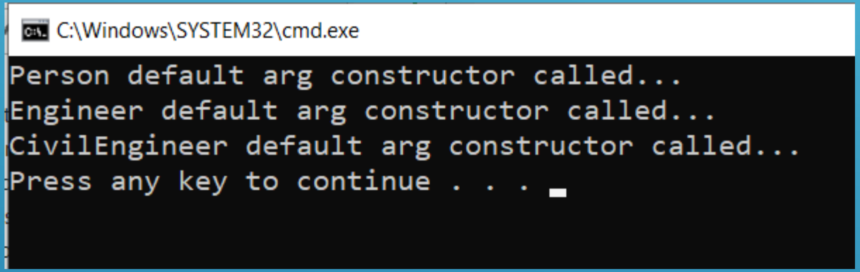
Default arg constructors with inheritance


class Person {…};
class Engineer : public Person
Engineer
Person
class CivilEng : public Engineer
CivilEngineer
Engineer
Person



```cpp
#include <iostream>
#include "person.h"
#include "engineer.h"
#include "civilengineer.h"


int main(int argc, char **argv)
{
    CivilEngineer civil_eng1;

    return 0;
}
```

```cpp
#include <iostream>
#include "person.h"
#include "engineer.h"
#include "civilengineer.h"


int main(int argc, char **argv)
{
    CivilEngineer civil_eng1;

    return 0;
}
```



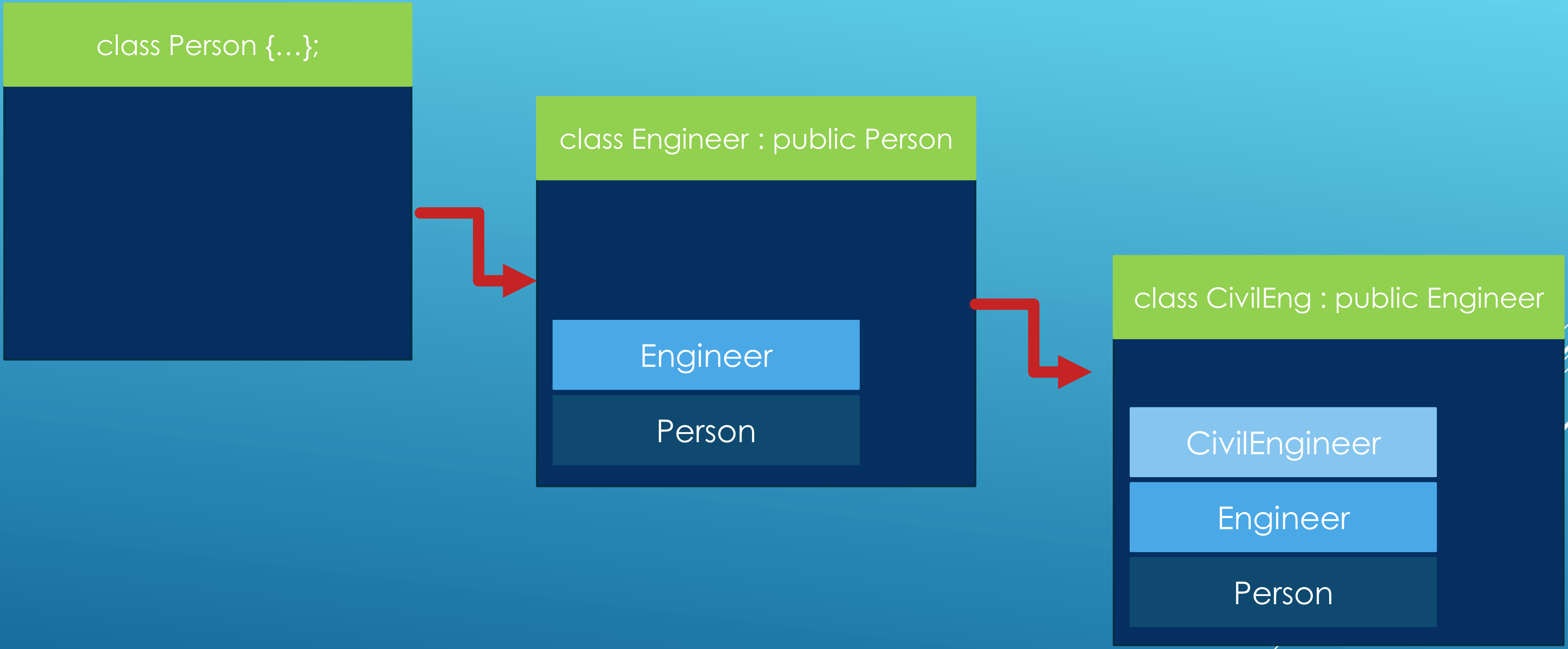
C:\Windows\SYSTEM32\cmd.exe

```
Person default arg constructor called...
Engineer default arg constructor called...
CivilEngineer default arg constructor called...
Press any key to continue . . .
```

Always provide a default constructor for your classes, especially if they will be part of an inheritance hierarchy

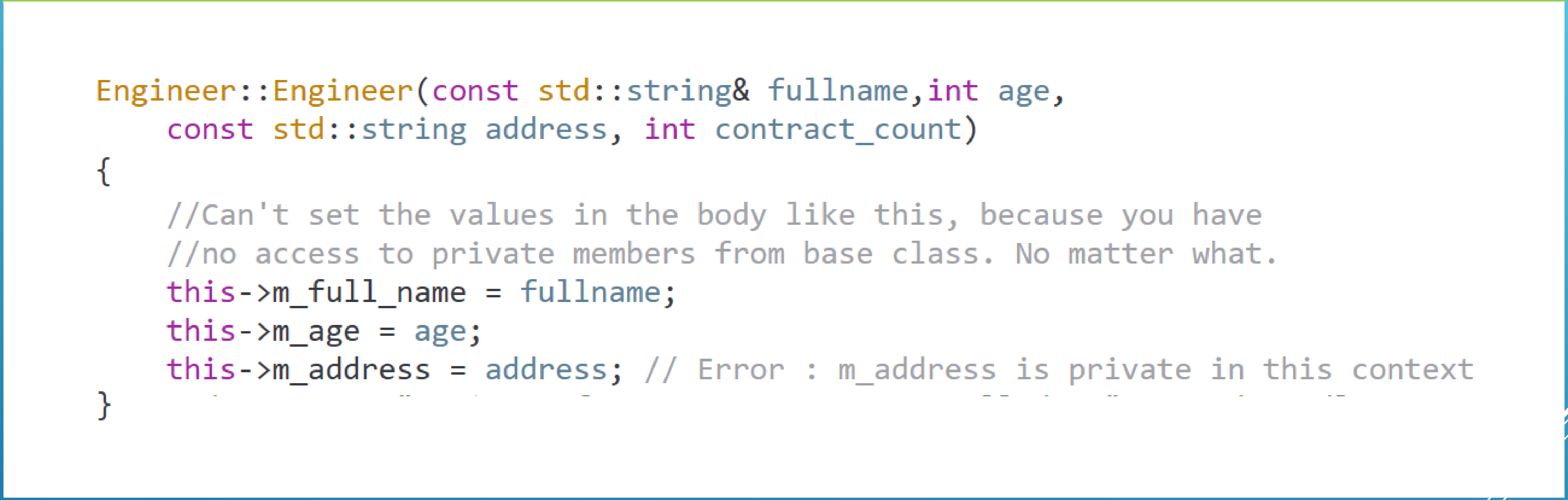
Slide intentionally left empty

Custom Constructors with Inheritance


class Person {…};
class Engineer : public Person
Engineer
Person
class CivilEng : public Engineer
CivilEngineer
Engineer
Person

```cpp
#include <iostream>
#include "person.h"
#include "engineer.h"
#include "civilengineer.h"


int main(int argc, char **argv)
{
    Person person1("John Snow",27,"Winterfell Cold 33St#75");
    std::cout << "person1 : " << person1 << std::endl;

    std::cout << "---------------------"<< std::endl;
    Engineer eng1("Daniel Gray",41,"Green Sky Oh Blue 33St#75",12);
    std::cout << "eng1 : " << eng1 << std::endl;

    std::cout << "---------------------" << std::endl;
    CivilEngineer civil_eng1("John Travolta",51,"Tiny Dog 42St#89",31,"Road Strength");
    std::cout << "civil_eng1 : " << civil_eng1 << std::endl;

    return 0;
}
```

Not ideal



```cpp
Engineer::Engineer(const std::string& fullname,int age,
    const std::string address, int contract_count)
{
    //Can't set the values in the body like this, because you have
    //no access to private members from base class. No matter what.
    this->m_full_name = fullname;
    this->m_age = age;
    this->m_address = address; // Error : m_address is private in this context
}
```

Initializer lists : Doing it wrong

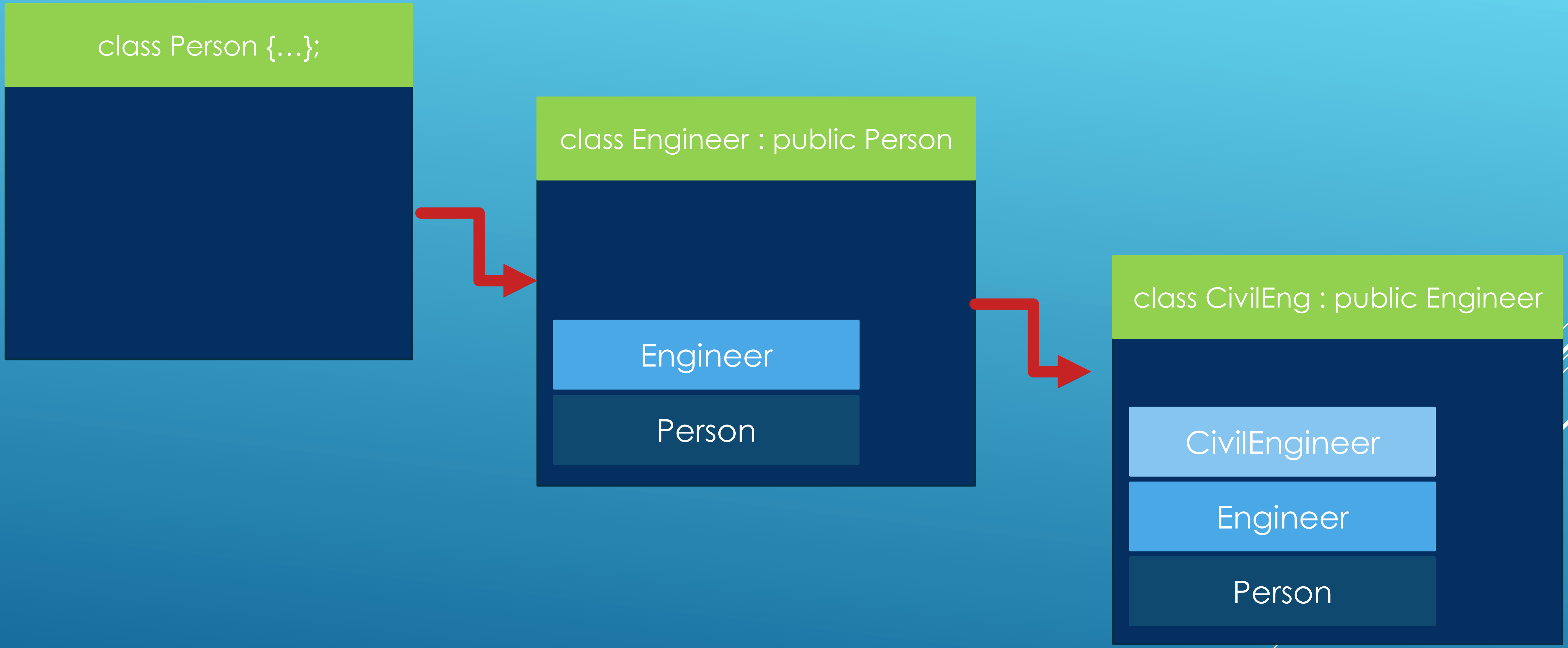
```
Engineer::Engineer(const std::string& fullname,int age,
    const std::string address, int contract_count)
     : m_full_name(fullname), m_age(age) , m_address(address),contract_count(contract_count)
{

}
```

Initializer lists

```
Engineer::Engineer(const std::string& fullname,int age,
    const std::string address, int contract_count)
    : Person(fullname,age,address) ,contract_count(contract_count)
{
}
```

Slide intentionally left empty

Copy constructors with Inheritance


class Person {…};
class Engineer : public Person
Engineer
Person
class CivilEng : public Engineer
CivilEngineer
Engineer
Person

```cpp
#include <iostream>
#include "person.h"
#include "engineer.h"
#include "civilengineer.h"

int main(int argc, char **argv)
{
    Engineer eng1("Daniel Gray",41,"Green Sky Oh Blue 33St#75",12);
    std::cout << "eng1 : " << eng1 << std::endl;

    std::cout << "---------------------" << std::endl;
    Engineer eng2(eng1);
    std::cout << "eng2 : " << eng2 << std::endl;
    return 0;
}
```

Person copy constructor

```
Person::Person(const Person& source)
     : m_full_name{source.m_full_name},
    m_age{source.m_age},
    m_address{source.m_address}
{
    std::cout << "Person Copy Constructor Called..." << std::endl;
}
```

Default arg constructor for base called

```
Engineer::Engineer(const Engineer& source)
    :   contract_count{source.contract_count}
{
    std::cout << "Engineer copy constructor called..." << std::endl;
}
```

```cpp
Engineer::Engineer(const Engineer& source)
 :Person(source.m_full_name, source.m_age,source.m_address),
        contract_count{source.contract_count}
{
    std::cout << "Engineer copy constructor called..." << std::endl;
}
```

```
Engineer::Engineer(const Engineer& source)
 :Person(source.m_full_name, source.m_age,source.m_address),
        contract_count{source.contract_count}
{
    std::cout << "Engineer copy constructor called..." << std::endl;
}
```

- Not reusing the copy constructor we have in Person
- m_address is private to Person, can't be directly accessed from Engineer object
- We could set up a public method to return the address but that could go against your design guidelines

Proper copy constructor

```
Engineer::Engineer(const Engineer& source)
    : Person(source),
        contract_count{source.contract_count}
{
    std::cout << "Engineer copy constructor called..." << std::endl;
}
```

Slide intentionally left empty

Inheriting base constructors

```cpp
class Person
{
    friend std::ostream& operator<<(std::ostream& , const Person& person);
public:
    Person() = default;
    Person(const std::string& fullname,int age,
    const std::string address);
    Person(const Person& source); // Copy constructor
    ~Person();

//Member variables
public:
    std::string m_full_name{"None"};
protected:
    int m_age{0};
private :
    std::string m_address{"None"};
};
```

Using statement : Inherit the constructors

```
class Engineer : public Person
{
    using Person::Person; // Inheriting the constructor
friend std::ostream& operator<<(std::ostream& out , const Engineer& operand);
public:
    Engineer(const Engineer& source);
    ~Engineer();

protected :
    int contract_count{999999};// Default value
};
```

Compiler generated constructor as result of inheritance

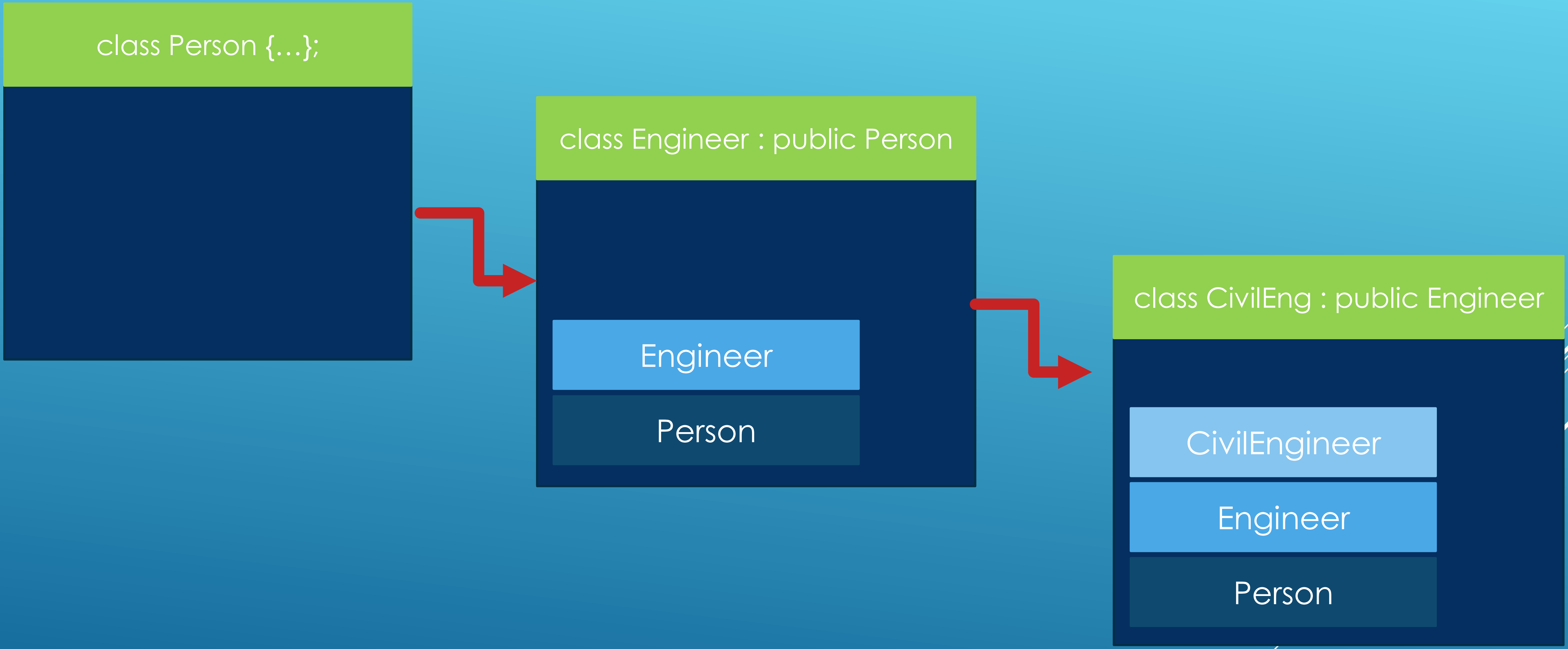
```
Engineer(const std::string& fullname,int age,
        const std::string address) : Person(fullname,age,address)
{
}
```

Some facts

- Copy constructors are not inherited. But you won't usually notice this as the compiler will insert an automatic copy constructor
- Inherited constructors are base constructors. They have no knowledge of the derived class. Any member from the derived class will just contain junk or whatever default value it's initialized with
- Constructors are inherited with whatever access specifier they had in base class
- On top of derived constructors, you can add your own that possibly properly initialize derived member variables
- Inheriting constructors adds a level of confusion to your code, it's not clear which constructor is building your object. It is recommended to avoid them and only use this feature if no other option is available.

Slide intentionally left empty

Inheritance with destructors

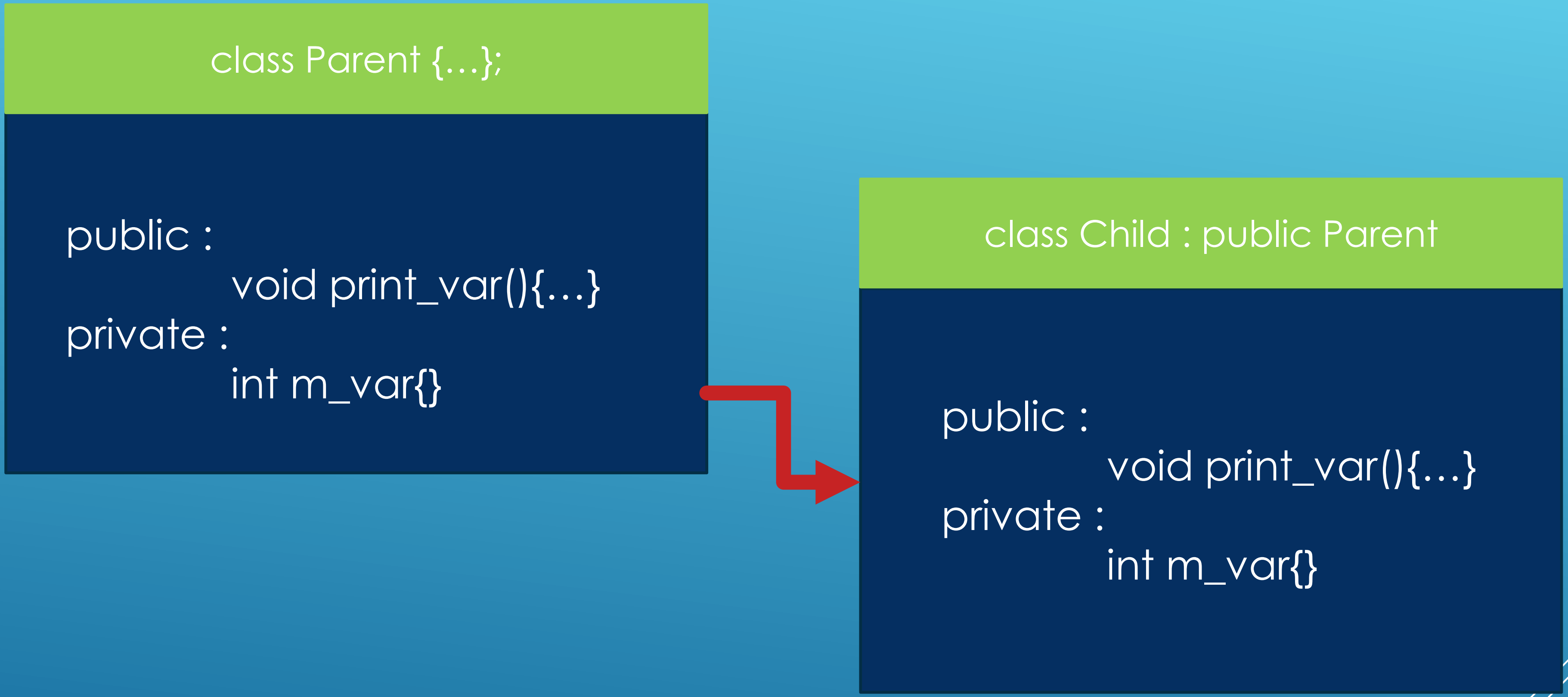

class Person {…};
class Engineer : public Person
Engineer
Person
class CivilEng : public Engineer
CivilEngineer
Engineer
Person

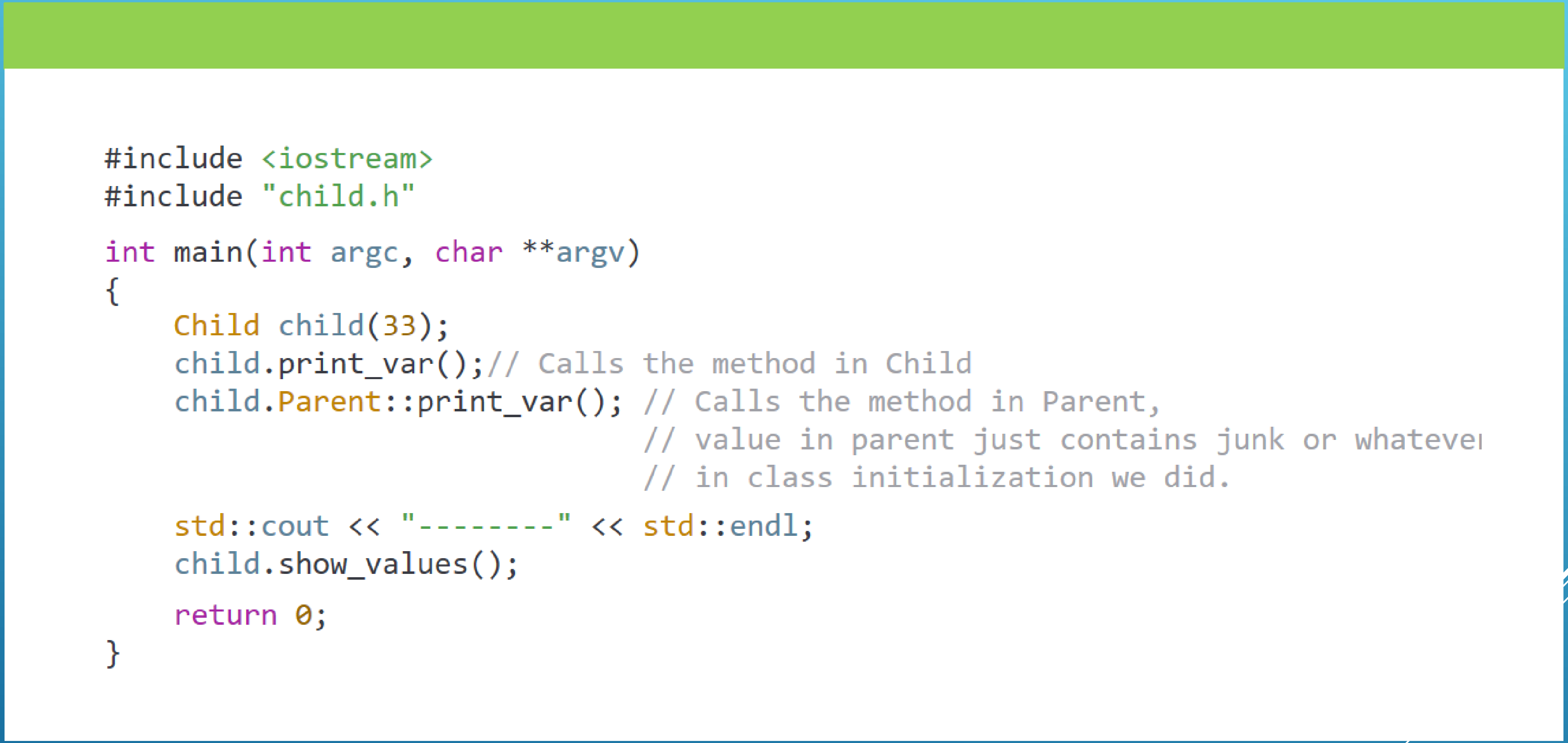
Base class part of derived object constructed first and destructed last

```cpp
#include <iostream>
#include "person.h"
#include "engineer.h"
#include "civilengineer.h"

int main(int argc, char **argv)
{
    CivilEngineer civil_eng1;

    std::cout << std::endl; // Destructors called in reverse order
                            // than the constructors

    return 0;
}
```

Slide intentionally left empty

Reused Symbols in Inheritance


class Parent {…};
public :
void print_var(){…}
private :
int m_var{}
class Child : public Parent
public :
void print_var(){…}
private :
int m_var{}

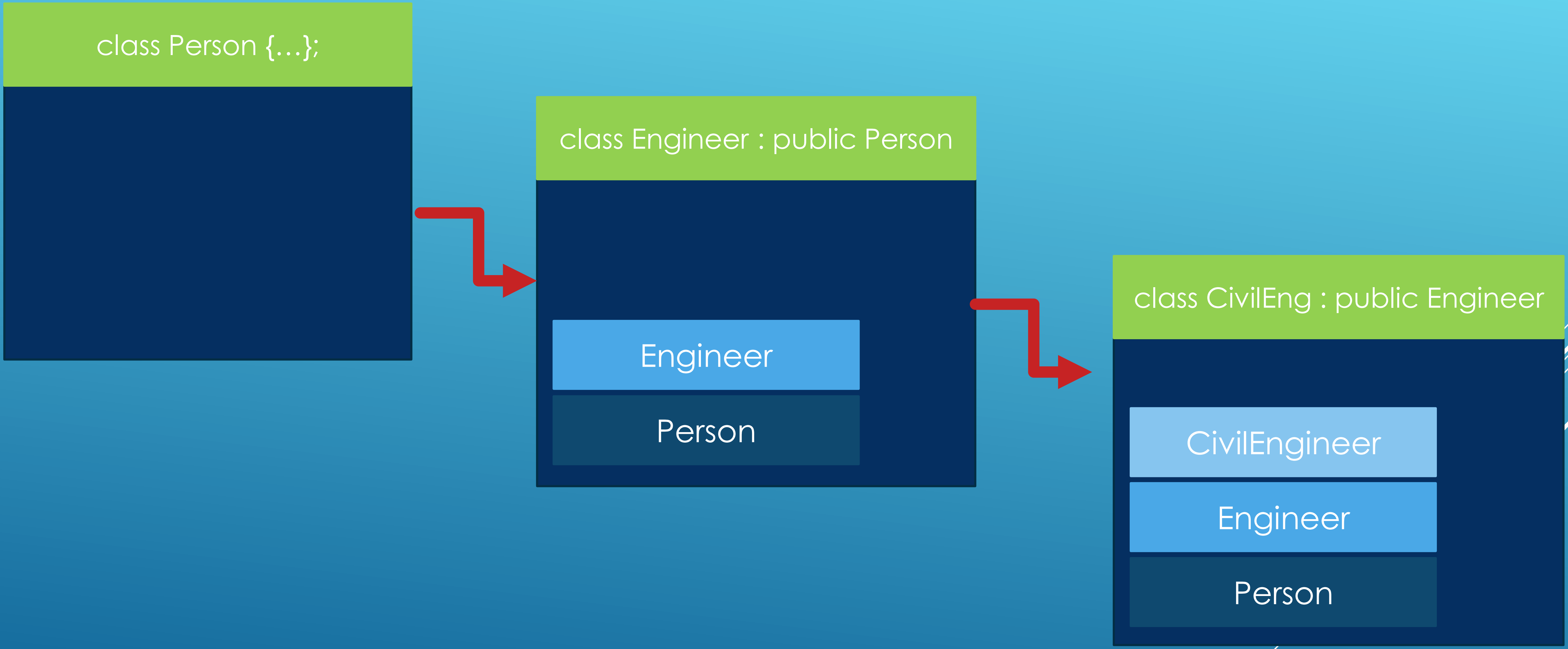


```cpp
#include <iostream>
#include "child.h"

int main(int argc, char **argv)
{
    Child child(33);
    child.print_var();// Calls the method in Child
    child.Parent::print_var(); // Calls the method in Parent,
                               // value in parent just contains junk or whateve
                               // in class initialization we did.

    std::cout << "--------" << std::endl;
    child.show_values();

    return 0;
}
```

```cpp
class Child : public Parent
{
public:
    Child();
    Child( int member_var) : m_member_var(member_var){
    }
    ~Child();

    void print_var()const{
        std::cout << "The value in child is : " << m_member_var << std::endl;
    }

    void show_values()const{
        std::cout << "The value in child is :" << m_member_var << std::endl;
        std::cout << "The value in parent is : " << Parent::m_member_var << std::endl;
                // The value in parent must be in accessible scope from the derived class.
    }
private:
    int m_member_var{1000};
};
```

Slide intentionally left empty

Inheritance : Summary


class Person {…};
class Engineer : public Person
Engineer
Person
class CivilEng : public Engineer
CivilEngineer
Engineer
Person

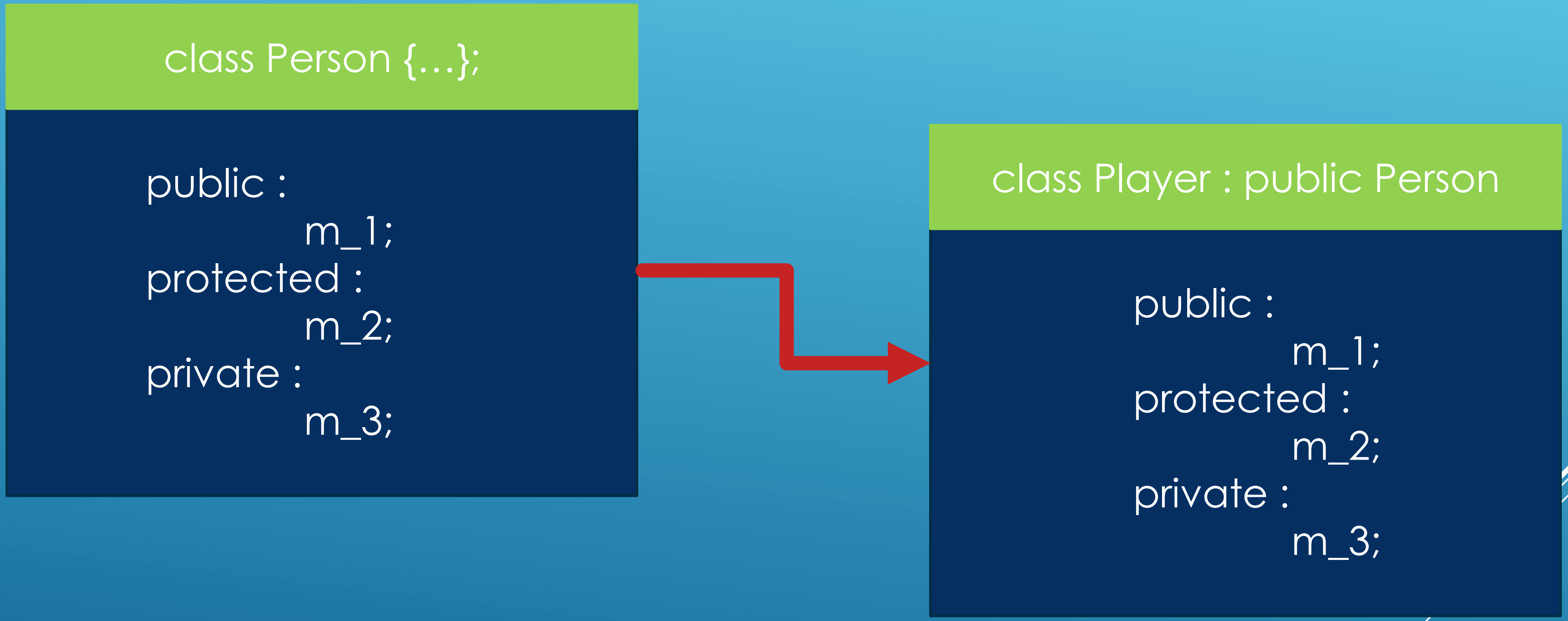
- A defining feature of Object Oriented Programming in C++
- Building types on top of other types
- Inheritance hierarchies can be set up to suit your needs
- Code reuse is improved

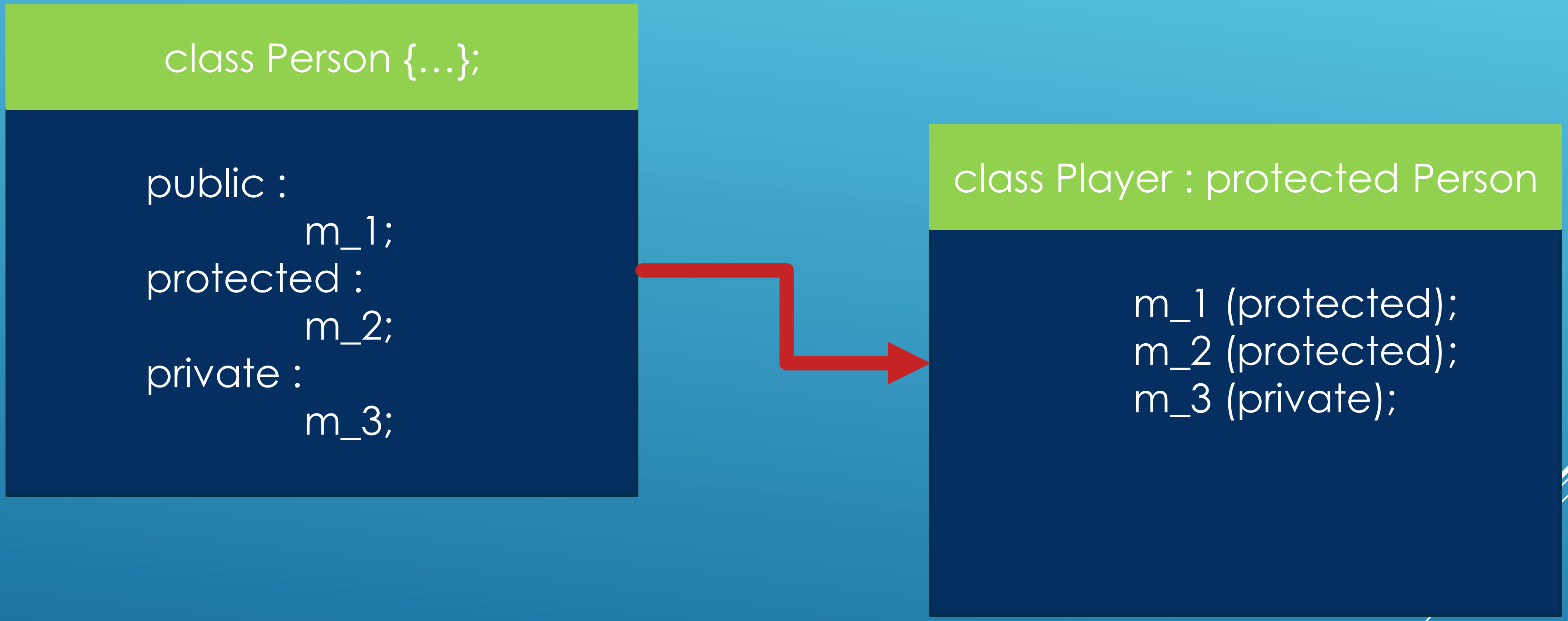
First try on Inheritance

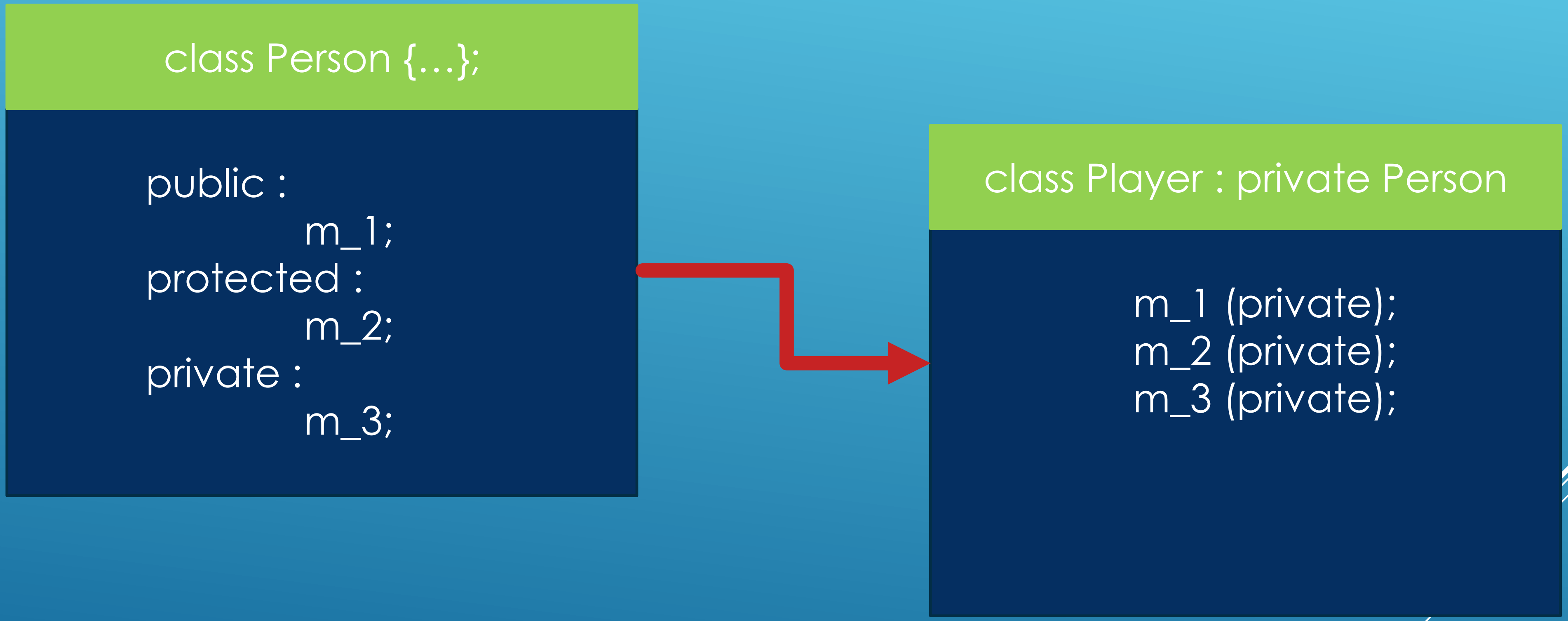
```cpp
//#include "person.h"
class Person; // Forward declaration

class Player : public Person
{
    friend std::ostream& operator<<(std::ostream& out, const Player& player);
public:
    Player() = default;
    Player(std::string game_param);
    ~Player();

private :
    std::string m_game{"None"};
};
```

Protected members

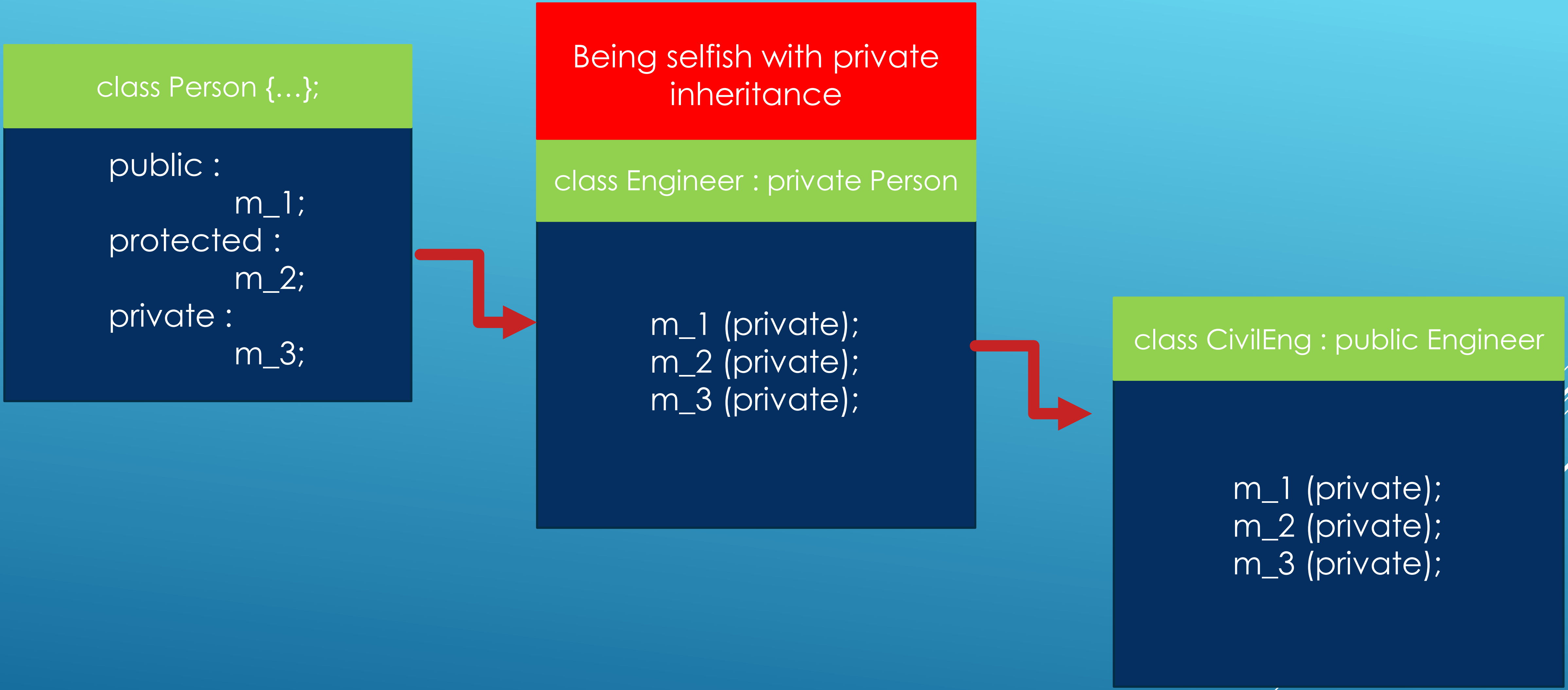
```
class Person
{
    friend std::ostream& operator<<(std::ostream& out, const Person& person);
public:
    Person();
    Person(std::string first_name_param, std::string last_name_param);
    ~Person();

protected :
    std::string first_name{"Mysterious"};
    std::string last_name{"Person"};
};
```


class Person {…};
public :
m_1;
protected :
m_2;
private :
m_3;
class Player : public Person
public :
m_1;
protected :
m_2;
private :
m_3;


class Person {…};
public :
m_1;
protected :
m_2;
private :
m_3;
class Player : protected Person
m_1 (protected);
m_2 (protected);
m_3 (private);


class Person {…};
public :
m_1;
protected :
m_2;
private :
m_3;
class Player : private Person
m_1 (private);
m_2 (private);
m_3 (private);

Being selfish with private inheritance



class Person {…};

```
public :
        m_1;
protected :
        m_2;
private :
        m_3;
```

class Engineer : private Person

```
m_1 (private);
m_2 (private);
m_3 (private);
```

class CivilEng : public Engineer

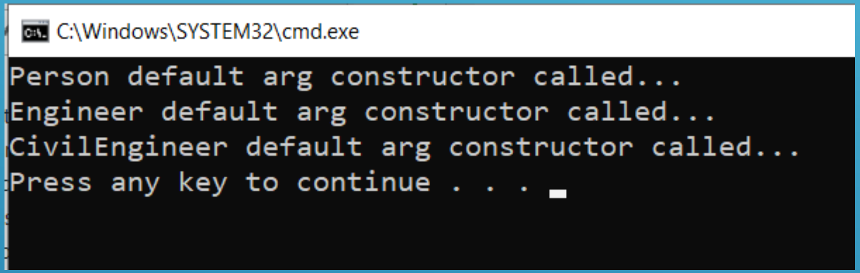
```
m_1 (private);
m_2 (private);
m_3 (private);
```

Resurrecting members

```
class Person; // Forward declaration
class Engineer : private Person
{
friend std::ostream& operator<<(std::ostream& out , const Engineer& operand);
public:
    Engineer();
    ~Engineer();

protected :
     using Person::get_full_name;
     using Person::get_age;
     using Person::get_address;

public :
    using Person::m_full_name;
    //using Person::m_address; // Compiler error.
    using Person::add_numbers; // Resurect back to public access
protected :
    int contract_count{0};
};
```

Base constructor call order

```cpp
#include <iostream>
#include "person.h"
#include "engineer.h"
#include "civilengineer.h"


int main(int argc, char **argv)
{
    CivilEngineer civil_eng1;

    return 0;
}
```


C:\Windows\SYSTEM32\cmd.exe

```
Person default arg constructor called...
Engineer default arg constructor called...
CivilEngineer default arg constructor called...
Press any key to continue . . .
```

Always provide a default constructor for your classes, especially if they will be part of an inheritance hierarchy

Calling custom base constructors from derived constructors

```
Engineer::Engineer(const std::string& fullname,int age,
    const std::string address, int contract_count)
    : Person(fullname,age,address) ,contract_count(contract_count)
{
}
```

Proper copy constructor

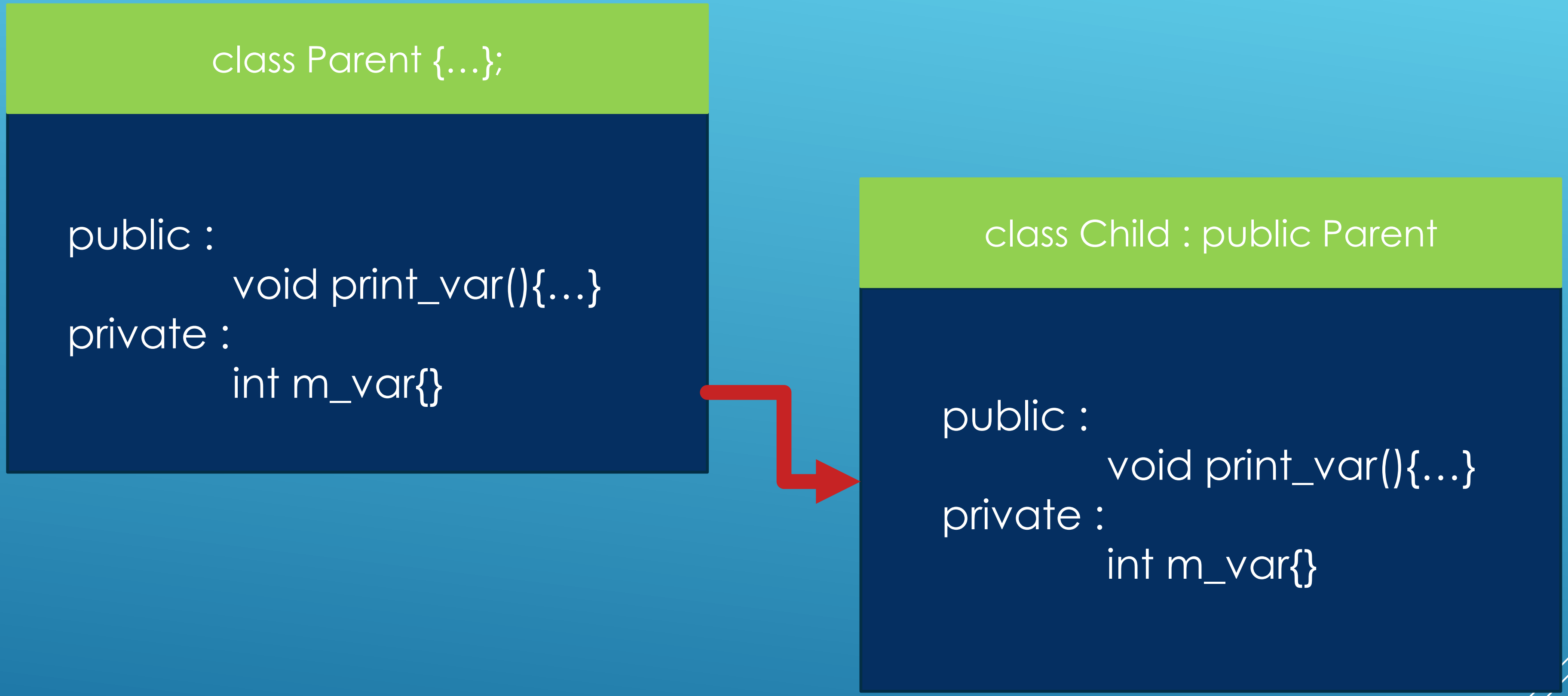
```cpp
Engineer::Engineer(const Engineer& source)
    : Person(source),
        contract_count{source.contract_count}
{
    std::cout << "Engineer copy constructor called..." << std::endl;
}
```

Using statement : Inherit the constructors

```
class Engineer : public Person
{
    using Person::Person; // Inheriting the constructor
friend std::ostream& operator<<(std::ostream& out , const Engineer& operand);
public:
    Engineer(const Engineer& source);
    ~Engineer();

protected :
    int contract_count{999999};// Default value
};
```

Inheritance and destructors

- Base class part of derived object constructed first and destructed last
- Destructors are called in a reverse order compared to constructors



```
class Parent {…};

public :
          void print_var(){…}
private :
          int m_var{}
```

```
class Child : public Parent

public :
          void print_var(){…}
private :
          int m_var{}
```