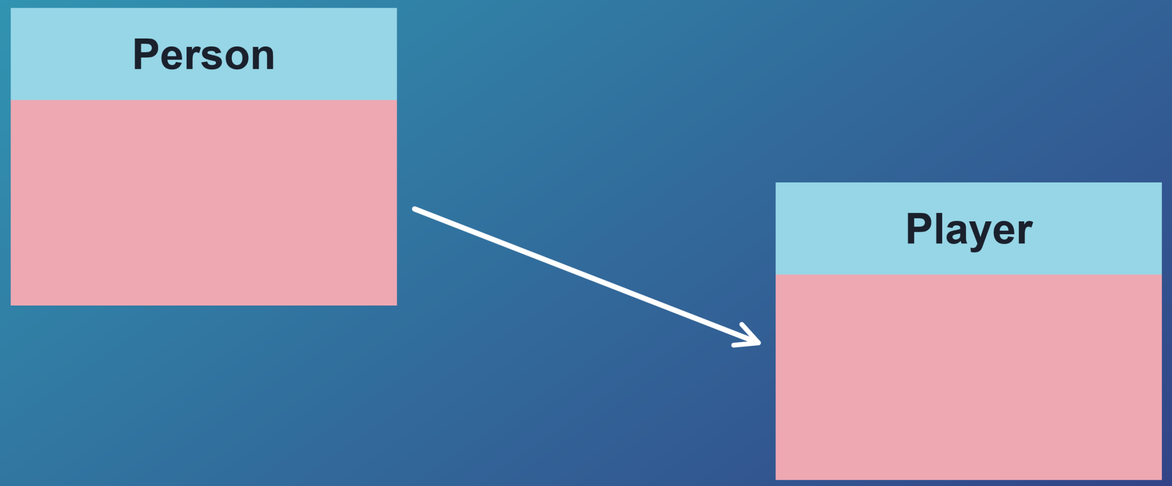
Inheritance: Constructors and Destructors

```
/*

    . Topics:
        . How objects are constructed and destructed in inheritance
        . The constructors are called from the base class to the derived class
            . The base parts are constructed first, and then the derived parts
                are constructed on top of them.
        . The destructors are called from the derived class to the base class

*/
```



Inheritance: Constructors and Destructors

```
export class Person {
public:
    // Constructors
    Person() {
        fmt::println("Person: Default constructor called.");
    }

    Person(const std::string& fullname, int age, const std::string& address)
        : m_full_name(fullname), m_age(age), m_address(address) {
        fmt::println("Person: Parameterized constructor called.");
    }

    // Destructor
    ~Person() {
        fmt::println("Person: Destructor called.");
    }

    //A function that uses fmt::format to store info about the object in the returned sting
    std::string get_info() const {
        return fmt::format("Person [Full name: {}, Age: {}, Address: {}]", m_full_name, m_age, m_address);
    }

protected:
    std::string m_full_name{"None"};
    int m_age{0};
    std::string m_address{"None"};
};
```

Inheritance: Constructors and Destructors

```
// Derived Class: Engineer
 export class Engineer : public Person {
 public:
     // Constructors
     Engineer() : Person() { // Explicitly calling base class constructor (optional here)
         fmt::println("Engineer: Default constructor called.");
     }

     Engineer(const std::string& fullname, int age, const std::string& address, int contracts)
         : Person(fullname, age, address), contract_count(contracts) {
         fmt::println("Engineer: Parameterized constructor called.");
     }

     // Destructor
     ~Engineer() {
         fmt::println("Engineer: Destructor called.");
     }

     //Get info
     std::string get_info() const {
         return fmt::format("Engineer [Full name: {}, Age: {}, Address: {}, Contracts: {}]",
                                                 m_full_name, m_age, m_address, contract_count);
     }

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

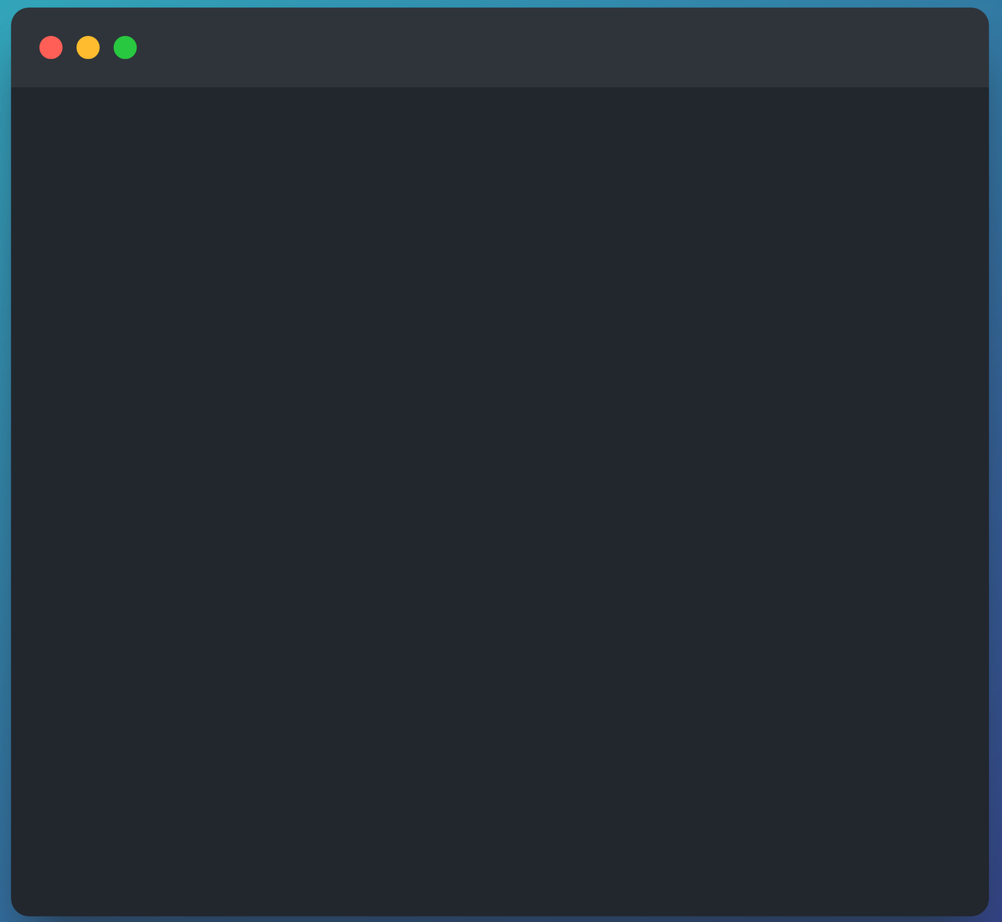
Inheritance: Constructors and Destructors

```cpp
// Derived Class: CivilEngineer
export class CivilEngineer : public Engineer {
public:
    // Constructors
    CivilEngineer() : Engineer() { // Explicitly calling base class constructor (optional here)
        fmt::println("CivilEngineer: Default constructor called.");
    }

    CivilEngineer(const std::string& fullname, int age, const std::string& address, int contracts, const std::string& speciality)
        : Engineer(fullname, age, address, contracts), m_speciality(speciality) {
        fmt::println("CivilEngineer: Parameterized constructor called.");
    }

    // Destructor
    ~CivilEngineer() {
        fmt::println("CivilEngineer: Destructor called.");
    }

    //Get info
    std::string get_info() const {
        return fmt::format("CivilEngineer [Full name: {}, Age: {}, Address: {}, Contracts: {}, Speciality: {}]",
                                                                m_full_name, m_age, m_address, contract_count, m_speciality);
    }

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

Inheritance: Constructors and Destructors

```
    //Try it out!
    //Default construct
    CivilEngineer eng1;
    fmt::println("eng1: {}", eng1.get_info());
```

Demo time!



Inheritance: Copy constructors and copy assignment operators

```
/*

    . Topics:

      . Copy Constructor Behavior with Inheritance:
          . Base Class Initialization: When a derived class object is copied,
              the base class's copy constructor is called first. This ensures
               that the base part of the derived object is initialized correctly.
          . Slicing: If a derived class object is assigned to a base class object,
              only the base part of the derived object is copied, leading to slicing.
              This can result in loss of derived class data.

      . Copy Assignment Operator Behavior with Inheritance:
          . Base Class Assignment: Similar to the copy constructor, when a derived
             class's copy assignment operator is invoked, the base class's
              assignment operator should be called explicitly to ensure proper
              assignment of the base part.
          . Self-Assignment Check: Always check for self-assignment in the copy
              assignment operator to prevent unintended behavior.

*/
```

Inheritance: Copy constructors and copy assignment operators

```
Person
│
├── Members: m_full_name, m_age, m_address
├── Constructors: Default, Custom, Copy
├── Operators: Copy assignment
├── Methods: get_info(), get_full_name(), get_age(), get_address()
│
└── Inherited by:
    └── Engineer
        │
        ├── Members: m_contract_count
        ├── Constructors: Default, Custom, Copy
        ├── Operators: Copy assignment
        ├── Methods: get_info(), get_contract_count()
        │
        └── Inherited by:
            └── CivilEngineer
                ├── Members: m_speciality
                ├── Constructors: Default, Custom, Copy
                ├── Operators: Copy assignment
                └── Methods: get_info()
```

Inheritance: Copy constructors and copy assignment operators

```
// Person class
export class Person {
public:

    // Copy constructor
    Person(const Person &source)
        : m_full_name{source.m_full_name}, m_age{source.m_age}, m_address{source.m_address} {
        fmt::println("Copy constructor for Person called...");
    }

    // Copy assignment operator
    Person &operator=(const Person &source) {
        fmt::println("Copy assignment operator for Person called...");
        if (this == &source) return *this;  // Check for self-assignment
        m_full_name = source.m_full_name;
        m_age = source.m_age;
        m_address = source.m_address;
        return *this;
    }
    ~Person() { fmt::println("Destructor for Person called..."); }

protected:
    std::string m_full_name{ "None" };
    int m_age{ 0 };

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

Inheritance: Copy constructors and copy assignment operators

```
// Engineer class
 export class Engineer : public Person {
 public:
     // Copy constructor
     Engineer(const Engineer &source)
         : Person(source), m_contract_count{source.m_contract_count} {
         fmt::println("Copy constructor for Engineer called...");
     }

     // Copy assignment operator
     Engineer &operator=(const Engineer &source) {
         fmt::println("Copy assignment operator for Engineer called...");
         if (this == &source) return *this;  // Check for self-assignment
         Person::operator=(source);          // Delegate to Person's copy assignment
         m_contract_count = source.m_contract_count;
         return *this;
     }

     ~Engineer() { fmt::println("Destructor for Engineer called..."); }

     //Get info
     std::string get_info() const {
         return fmt::format("Engineer [Full name: {}, Age: {}, Address: {}, Contracts: {}]",
                                         get_full_name(), get_age(), get_address(), m_contract_count);
     }
     int get_contract_count() const { return m_contract_count; }
 private:
     int m_contract_count{ 0 };
 };
```

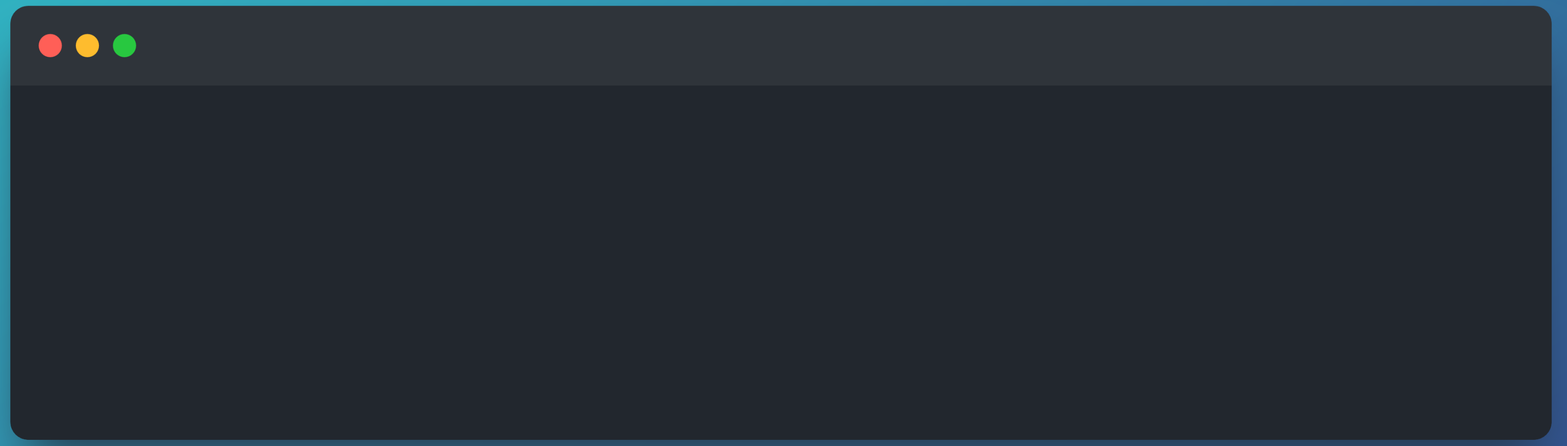
Inheritance: Copy constructors and copy assignment operators

```
// CivilEngineer class
    export class CivilEngineer : public Engineer {
    public:
        // Copy constructor
        CivilEngineer(const CivilEngineer &source)
            : Engineer(source), m_speciality{source.m_speciality} {
            fmt::println("Copy constructor for CivilEngineer called...");
        }

        // Copy assignment operator
        CivilEngineer &operator=(const CivilEngineer &source) {
            fmt::println("Copy assignment operator for CivilEngineer called...");
            if (this == &source) return *this;  // Check for self-assignment
            Engineer::operator=(source);        // Delegate to Engineer's copy assignment
            m_speciality = source.m_speciality;
            return *this;
        }

        //Get info
        std::string get_info() const {
            return fmt::format("CivilEngineer [Full name: {}, Age: {}, Address: {}, Contracts: {}, Speciality: {}]",
                                        get_full_name(), get_age(), get_address(), get_contract_count(), m_speciality);
        }

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

Inheritance: Copy constructors and copy assignment operators

```
//Trying it out

    inh_poly_2::CivilEngineer ce1{"Alex Johnson", 50, "789 Oak St", 10, "Structural"};
    inh_poly_2::CivilEngineer ce2{ce1};    //Copy construction
    inh_poly_2::CivilEngineer ce3;
    ce3 = ce1;    //Copy assignment operator
```

Demo time!



Constructor inheritance

```
/*
        Topics:
            . Constructor Inheritance:

                . By default, constructors of a base class are not inherited by
                  derived classes. This means if you want to create an instance
                  of the derived class (Engineer) using the same parameters as
                  the base class (Person), you would need to define those
                  constructors explicitly in the derived class.
                . Using using Person::Person; tells the compiler to inherit all
                  constructors from the Person class, which allows instances
                  of Engineer to be created using the same constructor signatures
                  as Person. Pay attention to the fact that ALL constructors are
                  inherited, including the default constructor.

                . You can add your own constructors to the derived class, which
                  will be called in addition to the inherited constructors.
                  This is useful when you need to initialize additional members
                  in the derived class.
                . ATTENTION: Copy constructors are not inherited in C++.


*/
```

Constructor inheritance

```
Person
│
├── Members: m_full_name, m_age, m_address
├── Constructors: Default, Custom, Copy
├── Methods: get_info(), get_full_name(), get_age(), get_address()
│
└── Inherited by:
    └── Engineer
        │
        ├── Members: m_contract_count
        ├── Constructors: Default, Custom, Copy
        ├── Methods: get_info(), get_contract_count()
```

Constructor inheritance

```
// Person class
export class Person {
public:
    // Default constructor
    Person() {
        fmt::print("Default constructor for Person called...\n");
    }

    // Custom constructor
    Person(std::string_view fullname, int age, std::string_view address)
        : m_full_name{fullname}, m_age{age}, m_address{address} {
        fmt::print("Custom constructor for Person called...\n");
    }

    // Copy constructor (not inheritable)
    Person(const Person &source)
        : m_full_name(source.m_full_name), m_age(source.m_age), m_address(source.m_address) {
        fmt::print("Custom copy constructor for Person called...\n");
    }

private:
    std::string m_full_name{"None"};
    int m_age{0};
    std::string m_address{"None"};
};
```

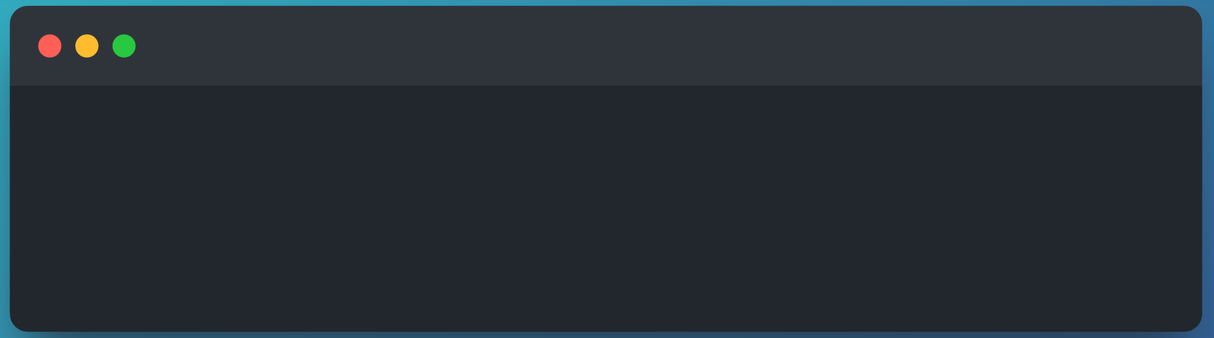
Constructor inheritance

```cpp
// Engineer class
export class Engineer : public Person {
    //The location of this using declaration doesn't matter.
    //We can build objects using the three param inherited constructor,
    //even if it shows up in a private section of the Engineer class.
    using Person::Person; // Inheriting constructors

public:
    // Custom constructor
    Engineer(std::string_view fullname, int age, std::string_view address, int contract_count_param)
        : Person(fullname, age, address), m_contract_count(contract_count_param) {
        fmt::print("Custom constructor for Engineer called...\n");
    }

    // Destructor
    ~Engineer() {}

    // Returns formatted information about the engineer
    std::string get_info() const {
        return fmt::format("Engineer [Full name: {}, Age: {}, Address: {}, Contract Count: {}]",
                           get_full_name(), get_age(), get_address(), m_contract_count);
    }

    // Getter for contract count
    int get_contract_count() const { return m_contract_count; }

private:
    int m_contract_count{0};
};
```

Constructor inheritance

```
//Trying it out
  Engineer eng1("Daniel Gray", 41, "Green Sky Oh Blue 33St#75");
  fmt::println("eng1 : {}", eng1.get_info());
```

Demo time!



Inheritance: Member hiding.

```
/*

      Topics:
        . Member Hiding in Inheritance

            . In C++, when a derived class defines a member (either a variable or a
              function) with the same name as one in its base class, the base class
              member is said to be "hidden."
            . This means that within the scope of the derived class, any attempt to access
              the member with that name will refer to the derived class's member, rather
              than the base class's member.

*/
```

Inheritance: Member hiding.

```
// Parent class
export class Parent {
public:
    Parent() = default;
    Parent(int member_var) : m_member_var(member_var) {}

    void print_var() const {
        fmt::println("The value in parent is: {}", m_member_var);
    }
protected:
    int m_member_var{ 100 }; // Default value
};

// Child class
export class Child : public Parent {
public:
    Child() = default; // Default constructor
    Child(int member_var) : Parent(member_var), m_member_var(member_var) {} // Initialize base member

    void print_var() const {
        fmt::println("The value in child is: {}", m_member_var);
    }

    void show_values() const { }
private:
    int m_member_var{ 1000 }; // Child-specific member
};
```

Inheritance: Member hiding.

```
//Trying it out
inh_poly_4::Parent parent1;
parent1.print_var();  // Calls Parent's print_var

fmt::println("");

inh_poly_4::Child child1;
child1.print_var();    // Calls Child's print_var
child1.Parent::print_var();  // Calls Parent's print_var
child1.show_values();  // Calls Child's show_values, which in turn accesses Parent's m_member_var
```