Slides

Section : Diving deep into constructors and initialization

Slide intentionally left empty

Diving Deep into Constructors and Initialization : Intro

Constructors

A window to customize how our own class objects are put together

Slide intentionally left empty

Default parameters for constructors

```
class Cylinder
{
private :
    double base_radius{1};
    double height{2};

public:
    Cylinder() = default;
    Cylinder(double radius_param , double height_param );

    //Getters
    double get_base_radius() const;
    double get_height() const;

    //Setters
    void set_base_radius(double radius_param);
    void set_height(double height_param);

    double volume();

};
```

Default parameter

```
class Cylinder
{
private :
    double base_radius{1};
    double height{2};

public:
    Cylinder() = default;
    Cylinder(double radius_param , double height_param = 10 );

    //Getters
    double get_base_radius() const;
    double get_height() const;

    //Setters
    void set_base_radius(double radius_param);
    void set_height(double height_param);

    double volume();

};
```

Use default arguments

```
Cylinder cylinder1(5);
std::cout << "c1 radius : " << cylinder1.get_base_radius() << std::endl;
std::cout << "c1 height : " << cylinder1.get_height() << std::endl;
```

Specify all default parameters

```
class Cylinder
{
private :
    double base_radius{1};
    double height{2};

public:
    Cylinder() = default;
    Cylinder(double radius_param = 5 , double height_param = 10 );

    //Getters
    double get_base_radius() const;
    double get_height() const;

    //Setters
    void set_base_radius(double radius_param);
    void set_height(double height_param);

    double volume();

};
```

Confusion

```
Cylinder cylinder1;//Compiler will be confused as to which constructor to call
std::cout << "c1 radius : " << cylinder1.get_base_radius() << std::endl;
std::cout << "c1 height : " << cylinder1.get_height() << std::endl;
```

Slide intentionally left empty

Initializer lists for constructors

```
class Cylinder
{
private :
    double base_radius{1};
    double height{2};

public:
    Cylinder() = default;
    Cylinder(double radius_param , double height_param = 10 );

    //Getters
    double get_base_radius() const;
    double get_height() const;

    //Setters
    void set_base_radius(double radius_param);
    void set_height(double height_param);

    double volume() const;

};
```

Member wise assignment initialization

```
Cylinder::Cylinder(double radius_param , double height_param){
    base_radius = radius_param;
    height = height_param;
}
```

Initializer list initialization

```
Cylinder::Cylinder(double radius_param , double height_param )
    : base_radius(radius_param) , height(height_param)
{
    //Empty body
}
```

Initializer list benefits

- They avoid unnecessary copies. More on this in next lecture
- In some cases, they're the only way to initialize an object

Slide intentionally left empty

Initializer lists VS Member wise copy initialization

```
class Cylinder
{
private :
    double base_radius{1};
    double height{2};

public:
    Cylinder() = default;
    Cylinder(double radius_param , double height_param = 10 );

    //Getters
    double get_base_radius() const;
    double get_height() const;

    //Setters
    void set_base_radius(double radius_param);
    void set_height(double height_param);

    double volume() const;

};
```

Member wise assignment initialization

```
Cylinder::Cylinder(double radius_param , double height_param){
    base_radius = radius_param;
    height = height_param;
}
```

Initializer list initialization

```
Cylinder::Cylinder(double radius_param , double height_param )
    : base_radius(radius_param) , height(height_param)
{
    //Empty body
}
```

Member wise copy

- Two steps :
  - object creation
  - member variable assignment
- Potential unnecessary copies of data
- Order of member variables doesn't matter

Initializer lists

- Initialization happens at real object creation
- Unnecessary copies avoided
- Order of member variables matters

Recommendation

Always prefer initializer lists over member wise copy initialization.

Slide intentionally left empty

Explicit Constructors

One parameter constructor

```
class Square
{
public:
     Square(double side_param);
    ~Square();
    double surface() const;
private :
    double side;

};
```

One parameter constructor

```
Square::Square(double side_param) : side{side_param}
{
}

double Square::surface() const {
    return side*side;
}

Square::~Square()
{
}
```

Sneaky Implicit conversions

```
//Is square1 > to square2 ? true or false
bool compare( const Square& square1 ,const Square& square2){
    return (square1.surface() > square2.surface()) ? true : false;
}

int main(int argc, char **argv)
{
    Square s1(30.0);
    Square s2(20.0);
    std::cout << std::boolalpha;
    std::cout << "s1 > s2 : " << compare(s1,s2) << std::endl;

    std::cout << std::endl;
    std::cout << "Implicit converions" << std::endl;

    //44.5 Implicitly converted to Square(44.5)
    std::cout << "s1 > 44.5 : " << compare(s1,44.5) << std::endl;

    return 0;
}
```

Mark single parameter constructor as explicit

```
class Square
{
public:
     explicit Square(double side_param);
    ~Square();
    double surface() const;

private :
    double side;
};
```

Beware of constructors with all but one default parameters

```
class Square
{
public:
     explicit Square(double side_param, std::string color = "black");
    ~Square();
    double surface() const;

private :
    double side;
    std::string color;
};
```

Slide intentionally left empty

Constructor Delegation

Two separate constructors

```
class Square
{
public:
     explicit Square(double side_param);
     Square(double side_param , std::string color_param, int shading_param);

    ~Square();
    double surface() const;

private :
    double side;
    std::string color;
    int shading;
    double position;
};
```

Two separate constructors

```
Square::Square(double side_param) : side{side_param}
{
}

Square::Square(double side_param , std::string color_param, int shading_param)
    : side{side_param},color{color_param},shading(shading_param)
{

}
```

Delegate object construction

```
//Delegate object construction to other constructor
Square::Square(double side_param)  :  Square(side_param,"red",3)
{
    std::cout << "Body of Square constructor with single param" << std::endl;
}

Square::Square(double side_param , std::string color_param, int shading_param)
    : side{side_param},color{color_param},shading(shading_param)
{
    std::cout << "Body of Square constructor with multiple params" << std::endl;
}
```

Event sequence

. The one parameter constructor is called
. Before we get into the body of the one param constructor, the compiler realizes the delegation and calls the three param constructor to do actual object creation with the provided data

. The three param constructor constructs the object and initializes with the provided data. Notice that the actual object is constructed by the three param constructor

. Control reaches the body of the three param constructor

. Control reaches the body of the single param constructor

. COntrol goes back in main

. All these calls to constructors can be seen in the call stack with the debugger

What if you call the three param constructor yourself

```
//Delegate object construction to other constructor
Square::Square(double side_param)
{
    std::cout << "Body of Square constructor with single param" << std::endl;
    Square(side_param,"red",3);//DOESN'T DO WHAT YOU EXPECT. BAD!
}

Square::Square(double side_param , std::string color_param, int shading_param)
    : side{side_param},color{color_param},shading(shading_param)
{
    std::cout << "Body of Square constructor with multiple params" << std::endl;
}
```

No further initializations before/after delegation call

```
//Can't do further initialization after delegated call.
Square::Square(double side_param)  :  Square(side_param,"red",3),position(0.0)
//Square::Square(double side_param)  : position(0.0),Square(side_param,"red",3)
{
    std::cout << "Body of Square constructor with single param" << std::endl;
}
```

Can do anything we want in body though [after delegation call]

```
Square::Square(double side_param)  :Square(side_param,"red",3)
{
    std::cout << "Body of Square constructor with single param" << std::endl;
    position = 0.0;
}
```

Slide intentionally left empty

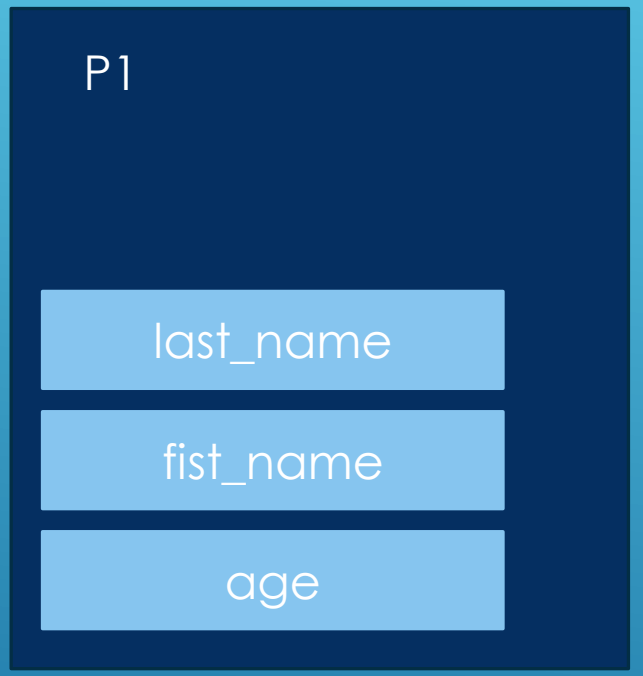
Copy Constructors

Person

```
class Person
{
private :
     std::string last_name{};
     std::string first_name{};
     int * age{};
public:
    //Constructors
    Person() = default;
    Person(std::string last_name_param, std::string first_name_param, int age_param);
    Person(std::string last_name_parm, std::string first_name_param);
    Person(std::string last_name);

    //Utilities
    void print_info(){
        std::cout << "Person object at : " << this
            <<" [ Last_name : " << last_name
            << ", First_name :  " << first_name
            << " ,age : " << *age
            << " , age address : " << age
            << " ]" << std::endl;
    }
};
```

Make a copy

```
Person p1("John","Snow",25);
p1.print_info();

//Creating copies
Person p2(p1);
p2.print_info();
```


P1
last_name
fist_name
age


P2
last_name
fist_name
age
P1
last_name
fist_name
age

Pointer address copied

```
Person p1("John","Snow",25);
p1.print_info();

Person p2(p1);
p2.print_info();

std::cout << std::endl;
std::cout << "Modifying age for p1 " << std::endl;
p1.set_age(30);
p1.print_info();
p2.print_info();
```

It is possible to set up your own copy constructor. This will disable the default one provided by the compiler. Yours will be the active one.

Our own copy constructor : Attempt 1

```
Person::Person(const Person source_person)
    :    last_name(source_person.get_last_name()),
         first_name(source_person.get_first_name()),
         age(source_person.get_age())
{

}
```

Our own copy constructor : Attempt 1

BAD

```
Person::Person(const Person source_person)
    :    last_name(source_person.get_last_name()),
         first_name(source_person.get_first_name()),
         age(source_person.get_age())
{

}
```

Our own copy constructor : Attempt 2

```
Person::Person(const Person & source_person)
    :    last_name(source_person.get_last_name()),
         first_name(source_person.get_first_name()),
         age(source_person.get_age())
{
    std::cout << "Copy constructor body" << std::endl;
}
```

Our own copy constructor : Attempt 2

BAD

```
Person::Person(const Person & source_person)
    :    last_name(source_person.get_last_name()),
         first_name(source_person.get_first_name()),
         age(source_person.get_age())
{
    std::cout << "Copy constructor body" << std::endl;
}
```

Our own copy constructor : Attempt 3

```
Person::Person(const Person & source_person)
    :   last_name(source_person.get_last_name()),
        first_name(source_person.get_first_name()),
        age(new int(*(source_person.get_age())))
{
    std::cout << "Copy constructor body" << std::endl;
}
```

Our own copy constructor : Attempt 3

GOOD

```
Person::Person(const Person & source_person)
    :   last_name(source_person.get_last_name()),
        first_name(source_person.get_first_name()),
        age(new int(*(source_person.get_age())))
{
    std::cout << "Copy constructor body" << std::endl;
}
```

Our own copy constructor : Delegating

```
Person::Person(const Person & source_person)
    :   Person(source_person.get_last_name(),
        source_person.get_first_name(),
        *(source_person.get_age()))
{
    std::cout << "Copy constructor body" << std::endl;
}
```

Slide intentionally left empty

Objects stored in arrays create copies

Array elements are copies

```
Person s1{"John","Out"};
Person s2{"Sean","Out"};
Person s3{"Bill","Out"};

//Copies created and stored in arrays
Person students[] {s1,s2,s3,Person(s1)};
```

Copies in range based for loop

```
Person s1{"John","Out"};
Person s2{"Sean","Out"};
Person s3{"Bill","Out"};

//Copies created and stored in arrays
Person students[] {s1,s2,s3,Person(s1)};

//Use references to avoid copies
for(Person s : students){
    s.set_first_name("Array");
}
```

References avoid copies

```
Person s1{"John","Out"};
Person s2{"Sean","Out"};
Person s3{"Bill","Out"};

//Copies created and stored in arrays . The last one is not a copy
Person students[] {s1,s2,s3,Person(s1),Person("Hills","Morion")};

//Use references to avoid copies
for(Person& s : students){
    s.set_first_name("Array");
}
```

Regular loops don't make copies

```
Person s1{"John","Out"};
Person s2{"Sean","Out"};
Person s3{"Bill","Out"};

//Copies created and stored in arrays . The last one is not a copy
Person students[] {s1,s2,s3,Person(s1),Person("Hills","Morion")};

for(size_t i{} ; i < std::size(students) ; ++i){
    students[i].set_first_name("Array");
    // (students + i)->set_first_name("Array");
    // (*(students + i)).set_first_name("Array");
}
```

Slide intentionally left empty

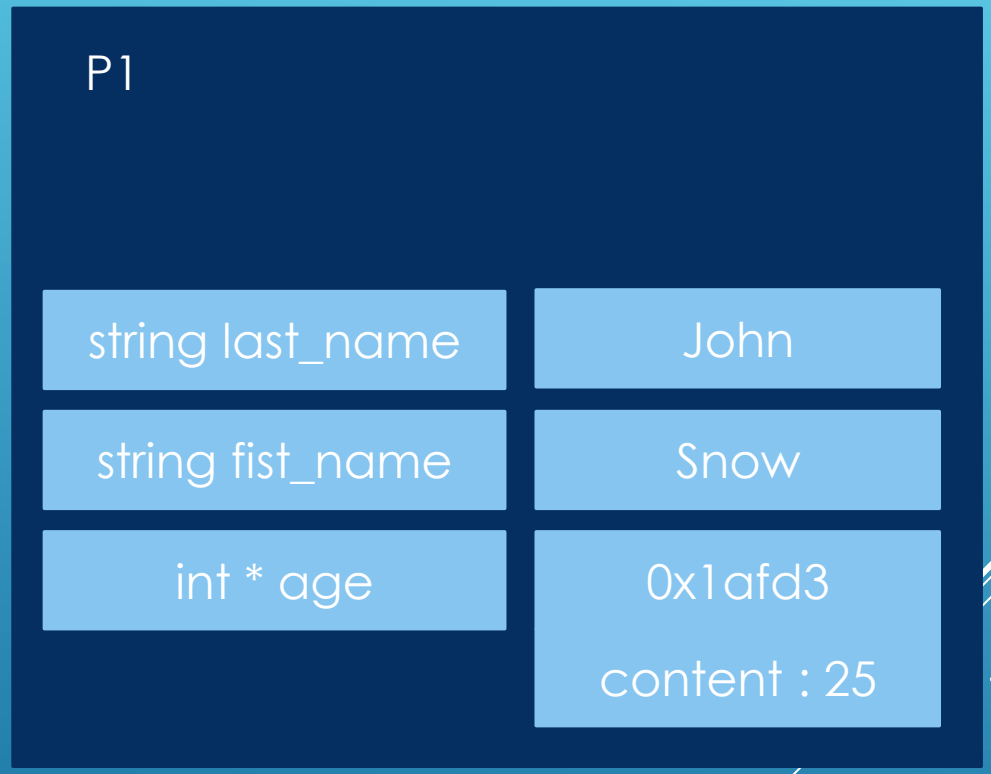
Deep copy VS Shallow copy

Shallow copy

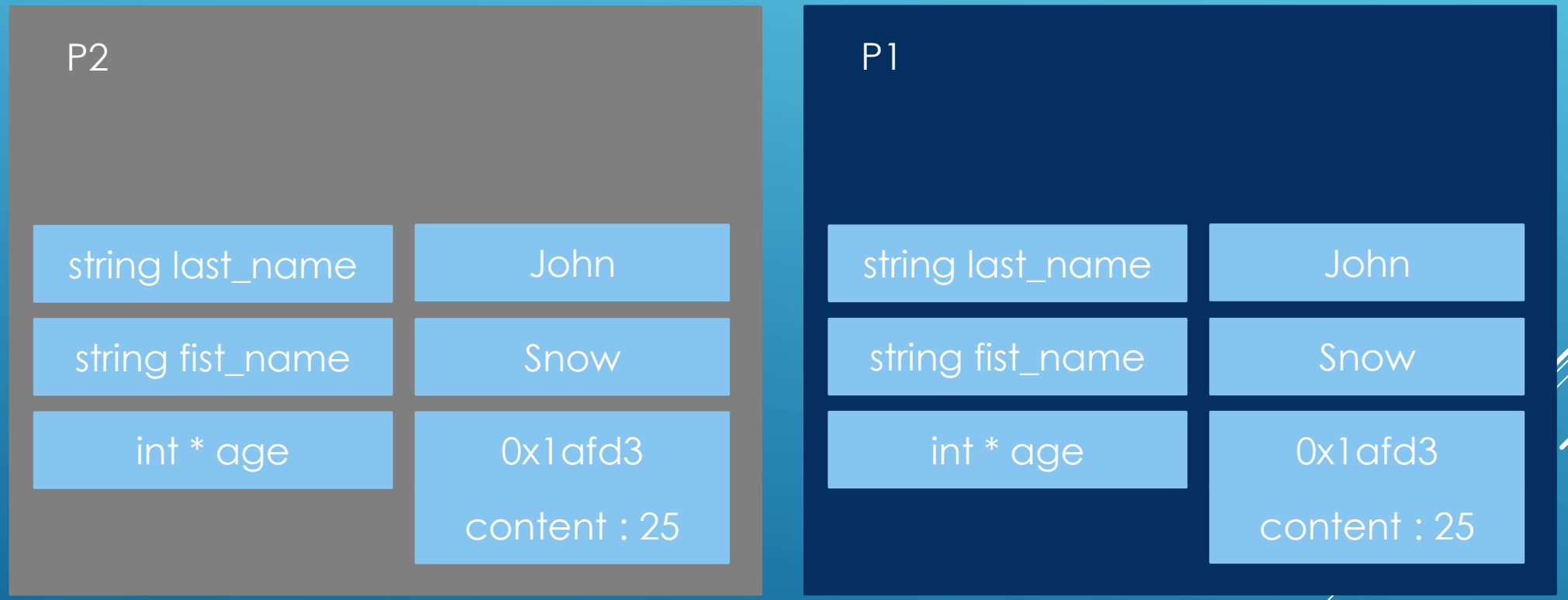
Member wise copy of member variables, even for pointers.

Deep copy

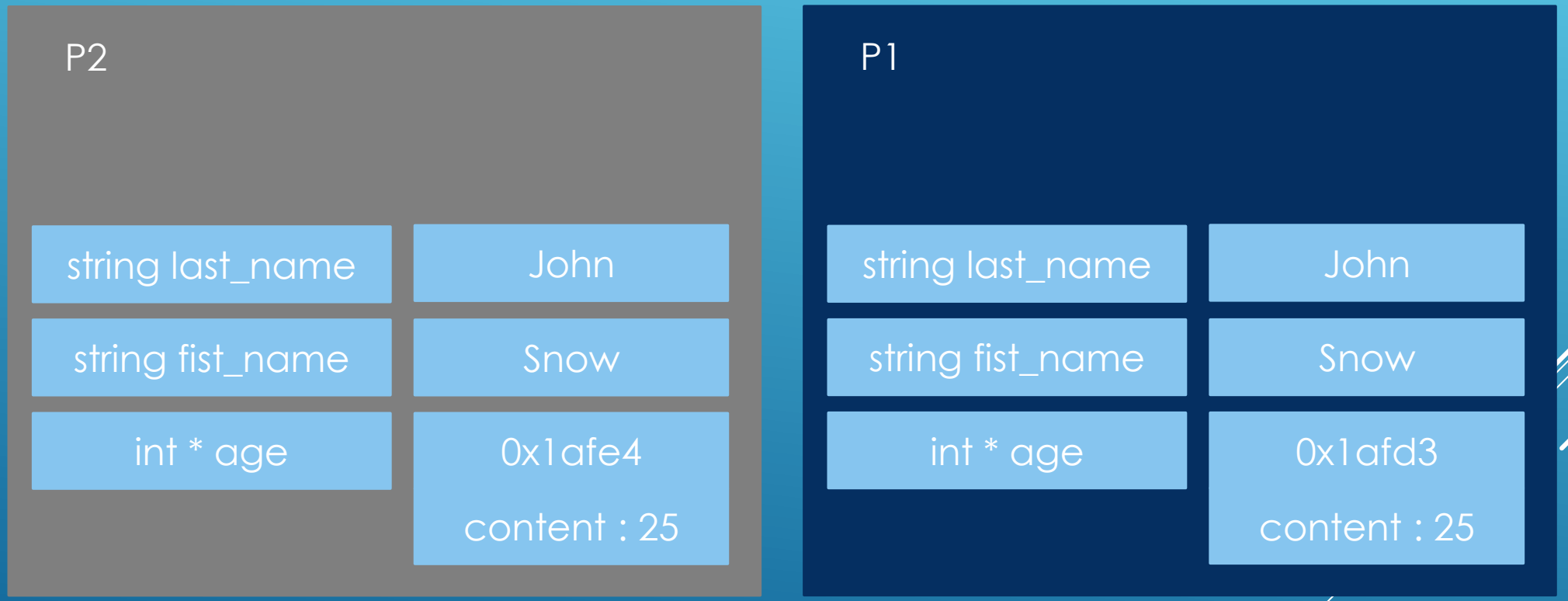
When pointer member variables are involved, allocating new memory and copying in data from the source pointer


P1
string last_name
John
string fist_name
Snow
int * age
0x1afd3
content : 25

Shallow copy



Deep copy


P2
string last_name
John
string fist_name
Snow
int * age
0x1afe4
content : 25
P1
string last_name
John
string fist_name
Snow
int * age
0x1afd3
content : 25

Slide intentionally left empty

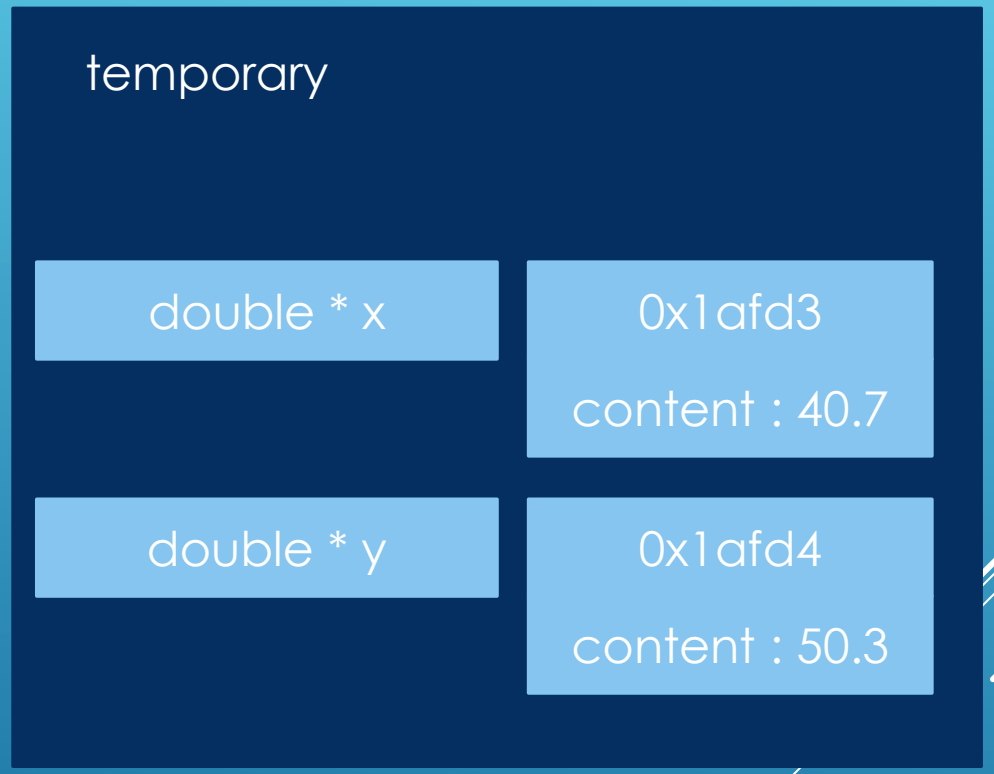
Move Constructors

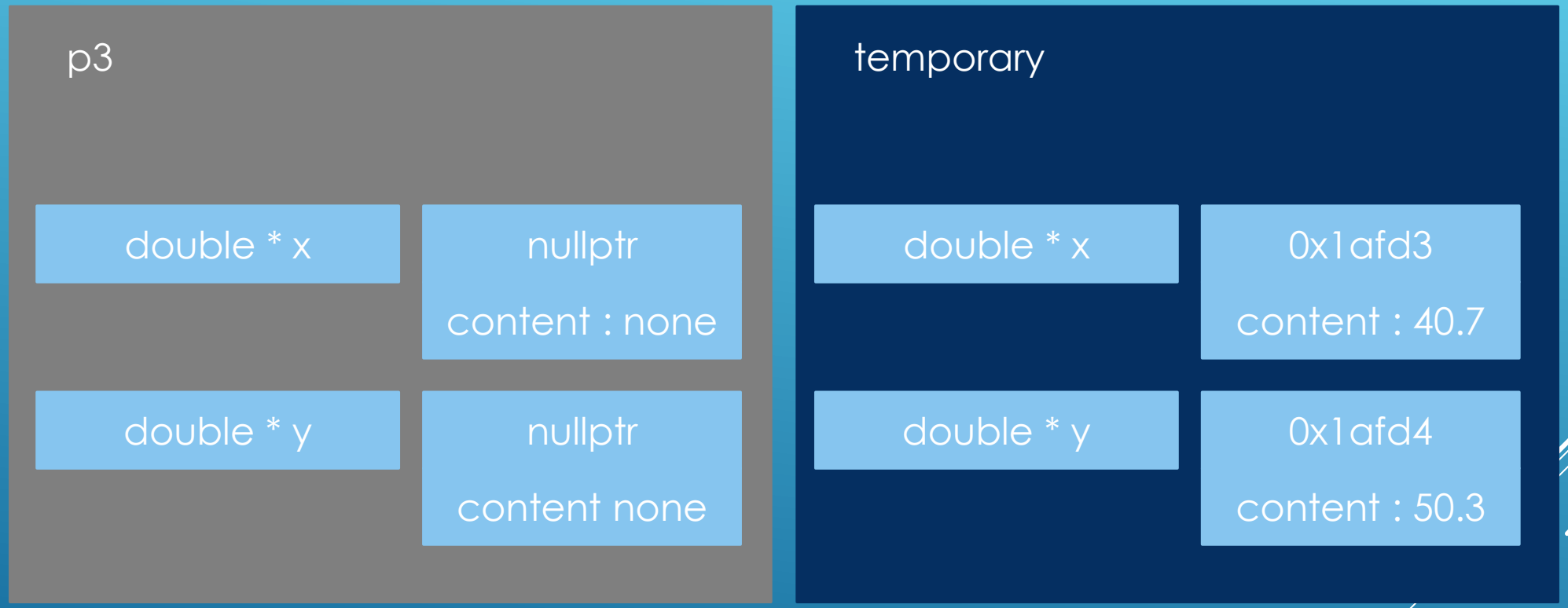
Stealing data from temporary objects

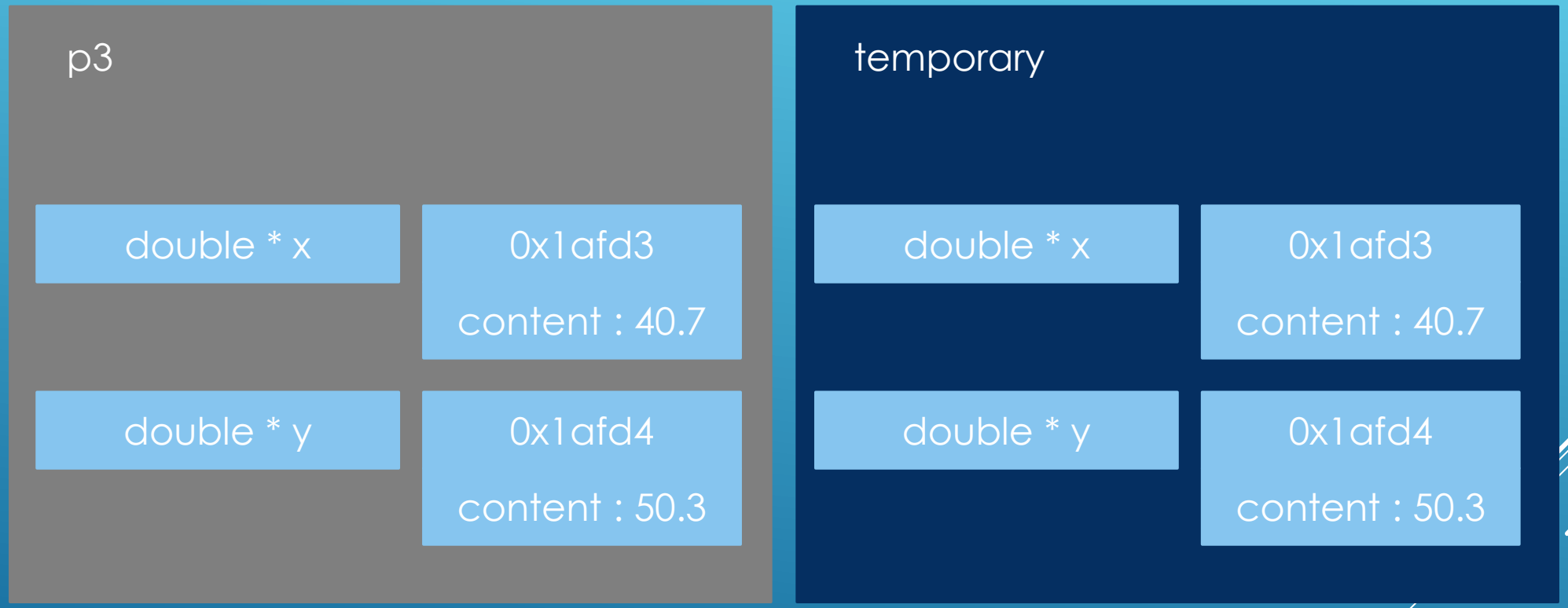
```
class Point
{
private :
    double* x{};
    double* y{};

public:
    Point(double x_param, double y_param);
    ~Point();

};
```

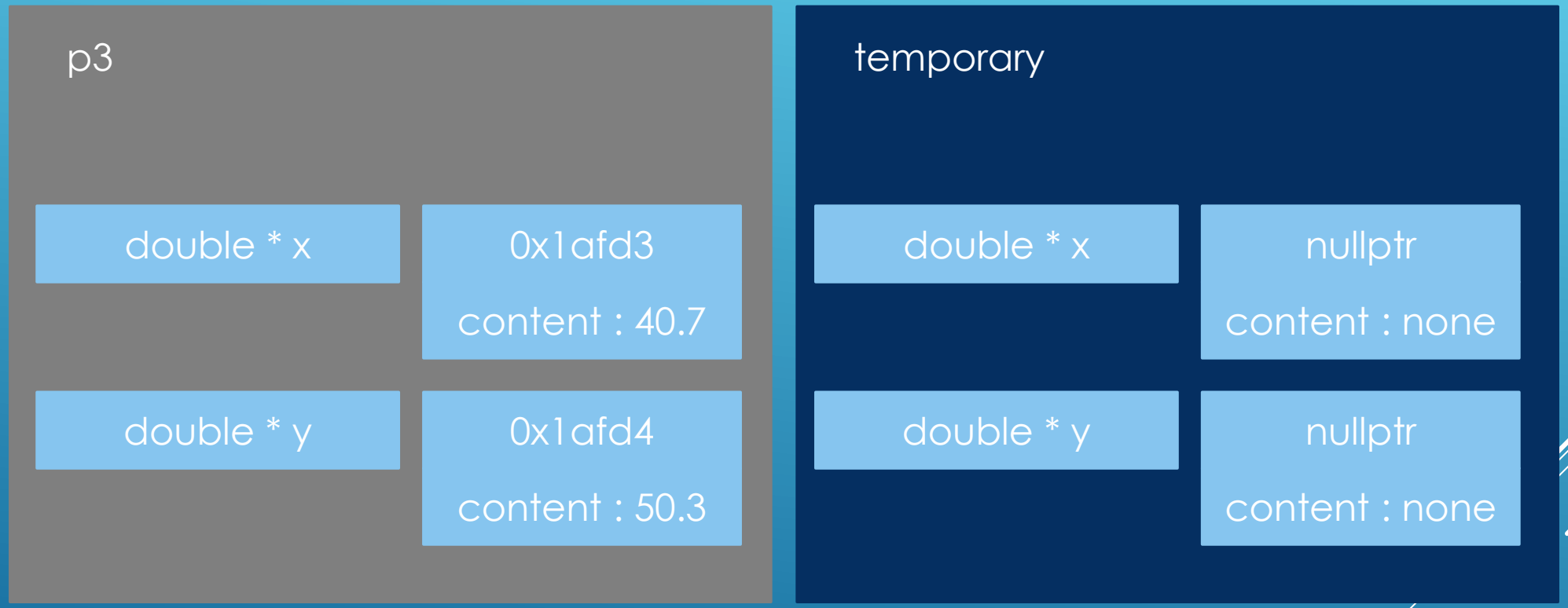
Building from temporaries : Syntax

```
Point p3(Point(40.7,50.3));
```


temporary
double * x
0x1afd3
content : 40.7
double * y
0x1afd4
content : 50.3


p3
double * x
nullptr
content : none
double * y
nullptr
content none
temporary
double * x
0x1afd3
content : 40.7
double * y
0x1afd4
content : 50.3


p3
double * x
0x1afd3
content : 40.7
double * y
0x1afd4
content : 50.3
temporary
double * x
0x1afd3
content : 40.7
double * y
0x1afd4
content : 50.3


p3
double * x
0x1afd3
content : 40.7
double * y
0x1afd4
content : 50.3
temporary
double * x
nullptr
content : none
double * y
nullptr
content : none

Your move constructor

```
//Move constructor
Point::Point( Point&& source_point)
    : x(source_point.get_x()),
        y(source_point.get_y())
{
    source_point.invalidate();
    std::cout << "Body of move constructor" << std::endl;
}
```

Making sure you have a temporary

```
//Point p3(Point(40.7,50.3));
Point p3(std::move(Point(40.7,50.3)));
```

Slide intentionally left empty

Deleted constructors

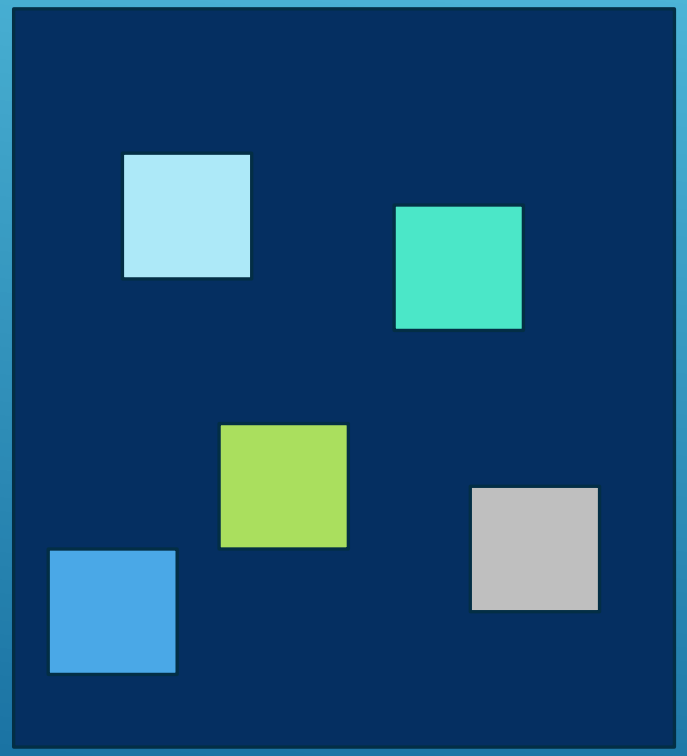
Disable a constructor from being used to create objects

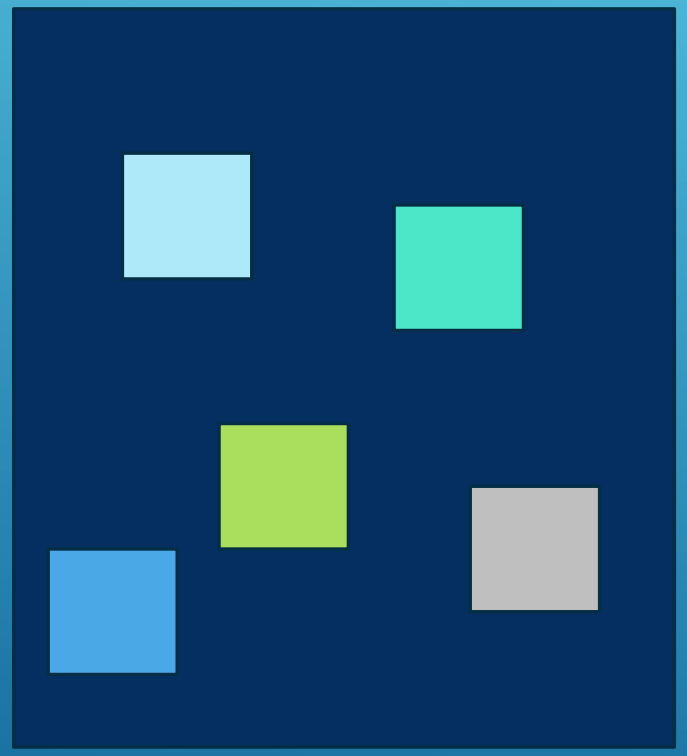
```
Point() = delete;
Point(const Point& source_point) = delete;
Point( Point&& source_point) = delete;
```

Slide intentionally left empty

Initializer list constructors

```
struct Point{
    double x;
    double y;
};

int main(int argc, char **argv)
{
    Point point1{12.5,45.3};
    std::cout << "Point [ x : " << point1.x
              << ", y : " << point1.y << "]" << std::endl;

    return 0;
}
```



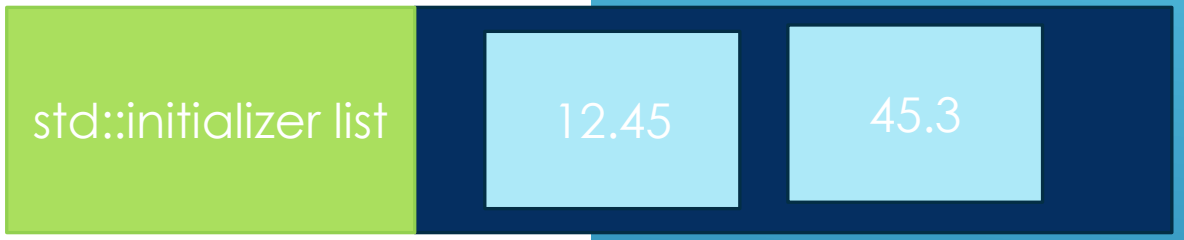


{1,2,3,4,...}

Initializer list constructor

```
class Point{
public :
    Point (std::initializer_list<double> list) {
        /* ...
    }

    void print_info() const{
        std::cout << "Point [ x : " << x << ", y : " << y << "]" << std::endl;
    }

private :
    double x;
    double y;
};
```

```cpp
struct Point{
    double x;
    double y;
};

int main(int argc, char **argv)
{
    Point point1{12.5,45.3};
    std::cout << "Point [ x : " << point1.x
              << ", y : " << point1.y << "]" << std::endl;

    return 0;
}
```



Traverse the std::initializer_list

```
for (auto i = list.begin(); i != list.end(); i++) {
        std::cout << *i << std::endl;
}
```

Pointer arithmetic

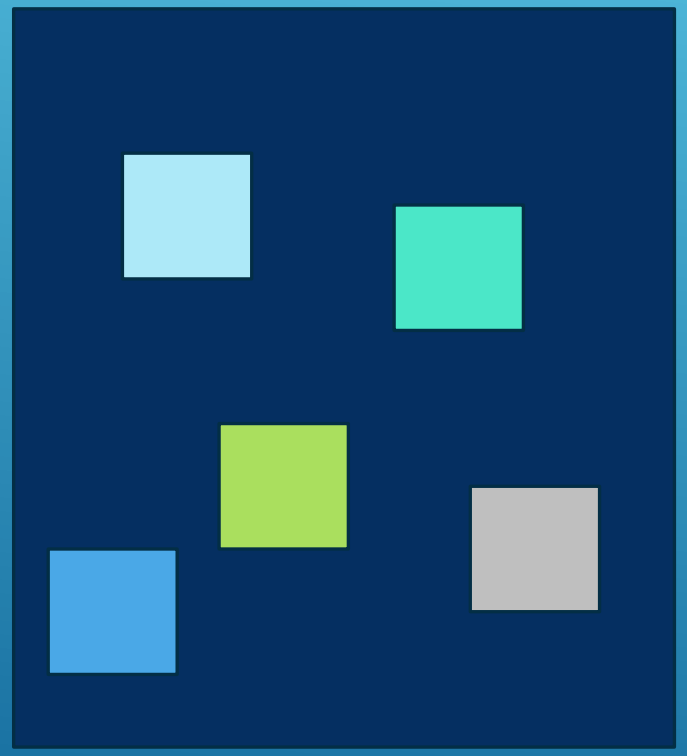
```
std::cout << "size : " << list.size() << std::endl;
std::cout << "first : " << *(list.begin()) << std::endl;
std::cout << "second : " << *(list.begin()+1) << std::endl;
```

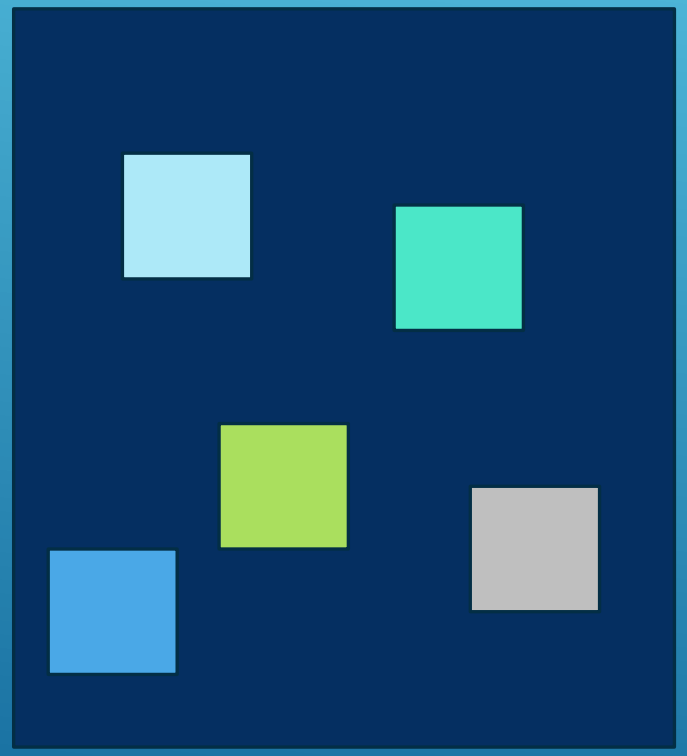
Store data into member variables

```
class Point{
public :
    Point (std::initializer_list<double> list) {
        x = *(list.begin());
        y = *(list.begin()+1);
    }

    void print_info() const{
        std::cout << "Point [ x : " << x << ", y : " << y << "]" << std::endl;
    }

private :
    double x;
    double y;
};
```

Slide intentionally left empty

Aggregate Initialization





{1,2,3,4,...}

All aggregates can be initialized with {}

```
struct Point{
    double x;
    double y;
};
int main(){

    Point p1{10.0,20.0}; // Struct
    int scores[] {1,2,3,4,5,6}; // Array

    std::cout << "Done!" << std::endl;
}
```

Slide intentionally left empty

Designated Initializers

```cpp
struct Component{
    double x;
    double y;
    double z;
};

void print_component(const Component& c){
    std::cout << "Component [ x : " << c.x << ", y : " << c.y << ", z : " << c.z << "]" << std::endl;
}

int main(){
    Component c1 {.x =10,.y = 20,.z =30};
    Component c2 {.x = 2.4, .z = 5.8};
    Component c3 {.y = 5.9, .z = 6.1};
    //Component c4 {.x = 4.3, .z = 5.3, .y = 9.4}; // Compiler error
    print_component(c1);
    print_component(c2);
    print_component(c3);

}
```

Slide intentionally left empty

Uniform Initialization for Aggregates

Uniform Initialization

Initializing any object either through () or {}

```cpp
struct Person{
    std::string name;
    int age;
};

void print_person(const Person& p){
    std::cout << "Person [ name : " << p.name << ", age : " << p.age << "]" << std::endl;
}

void print_array(int * arr, size_t size){
    for (size_t i{}; i < size;++i){
        std::cout << *(arr + i) << std::endl;
    }
}
```

```cpp
//Can initialize with {} : doesn't allow narrowing conversions
std::cout << "Initialization with {}: " << std::endl;
Person person1{"Steven",32}; // Aggregate initialization
print_person(person1);

int numbers1[5] {1,2,3,4,5}; // 5.55 in the place of 5 for example will throw a
                             // compiler error
print_array(numbers1,5);


//Can initialize with () : allows narrowing conversions
std::cout << std::endl;
std::cout << "Initialization with (): " << std::endl;
Person person2("Steven",32); // Aggregate initialization
print_person(person2);

int numbers2[5] (6,7,8.6,9,10);
print_array(numbers2,5);
```

Slide intentionally left empty

Diving Deep into Constructors and Initialization : Summary

Default parameters

```
class Cylinder
{
private :
    double base_radius{1};
    double height{2};

public:
    Cylinder() = default;
    Cylinder(double radius_param = 5 , double height_param = 10 );

    //Getters
    double get_base_radius() const;
    double get_height() const;

    //Setters
    void set_base_radius(double radius_param);
    void set_height(double height_param);

    double volume();

};
```

Initializer list initialization

```
Cylinder::Cylinder(double radius_param , double height_param )
    : base_radius(radius_param) , height(height_param)
{
    //Empty body
}
```

Member wise copy

- Two steps :
  - object creation
  - member variable assignment
- Potential unnecessary copies of data
- Order of member variables doesn't matter

Initializer lists

- Initialization happens at real object creation
- Unnecessary copies avoided
- Order of member variables matters

Explicit Constructors

```
class Square
{
public:
     explicit Square(double side_param);
    ~Square();
    double surface() const;

private :
    double side;
};
```

Constructor Delegation

```
//Delegate object construction to other constructor
Square::Square(double side_param)  :  Square(side_param,"red",3)
{
    std::cout << "Body of Square constructor with single param" << std::endl;
}

Square::Square(double side_param , std::string color_param, int shading_param)
    : side{side_param},color{color_param},shading(shading_param)
{
    std::cout << "Body of Square constructor with multiple params" << std::endl;
}
```

Copy constructors

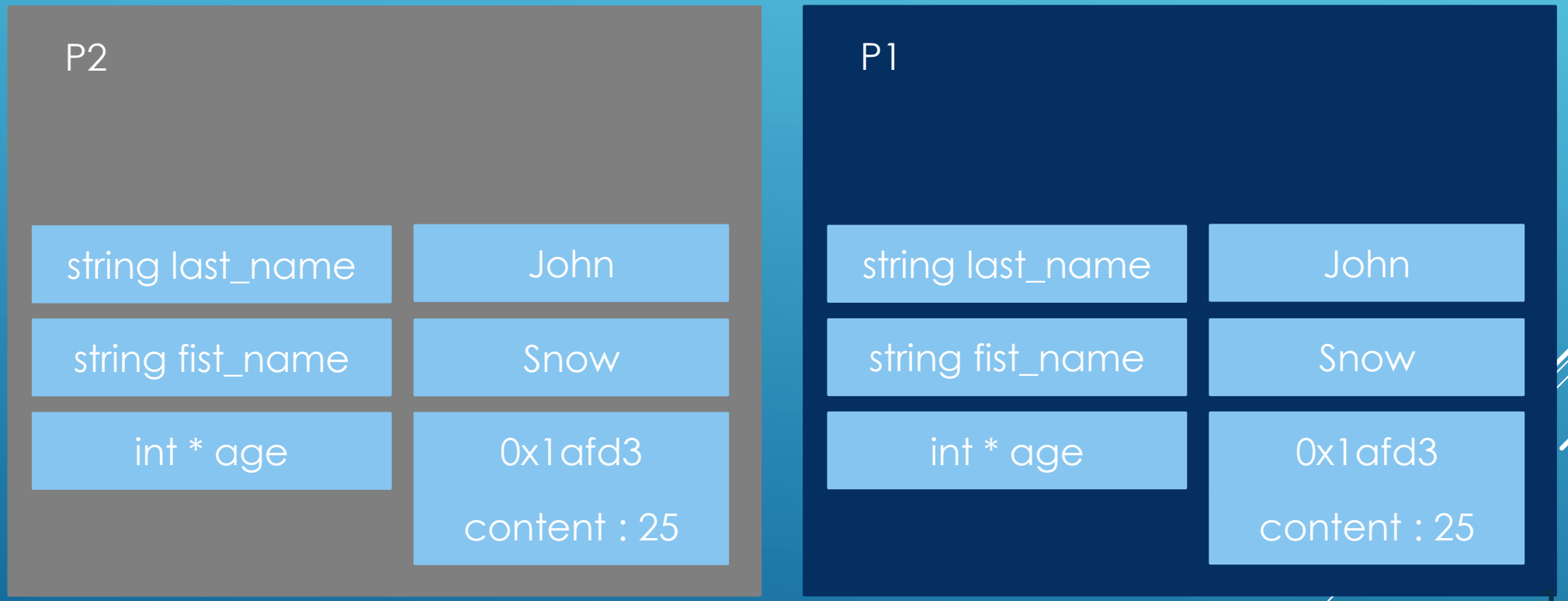
GOOD

```
Person::Person(const Person & source_person)
    :   last_name(source_person.get_last_name()),
        first_name(source_person.get_first_name()),
        age(new int(*(source_person.get_age())))
{
    std::cout << "Copy constructor body" << std::endl;
}
```

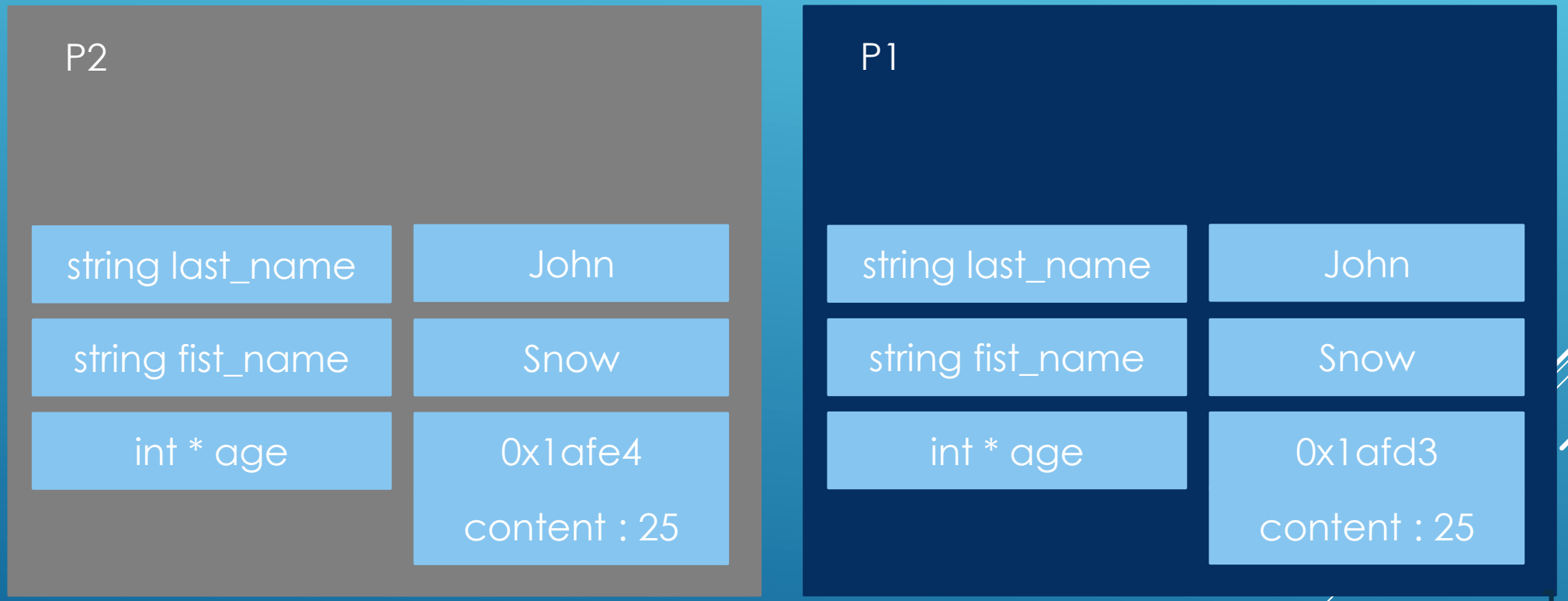
Copy constructor delegation

```
Person::Person(const Person & source_person)
    :   Person(source_person.get_last_name(),
        source_person.get_first_name(),
        *(source_person.get_age()))
{
    std::cout << "Copy constructor body" << std::endl;
}
```

Shallow copy



Deep copy


P2
string last_name
John
string fist_name
Snow
int * age
0x1afe4
content : 25
P1
string last_name
John
string fist_name
Snow
int * age
0x1afd3
content : 25

Move constructors

```
//Move constructor
Point::Point( Point&& source_point)
    : x(source_point.get_x()),
        y(source_point.get_y())
{
    source_point.invalidate();
    std::cout << "Body of move constructor" << std::endl;
}
```

Deleted constructors

```
Point() = delete;
Point(const Point& source_point) = delete;
Point( Point&& source_point) = delete;
```

Initializer list constructors

```
class Point{
public :
    Point (std::initializer_list<double> list) {
        x = *(list.begin());
        y = *(list.begin()+1);
    }

    void print_info() const{
        std::cout << "Point [ x : " << x << ", y : " << y << "]" << std::endl;
    }

private :
    double x;
    double y;
};
```

- Aggregate Initialization
- Designated Initializers
- Uniform Initialization for aggregates

Slide intentionally left empty