

SMART POINTERS & DESIGN PATTERNS



PROGRAMMA

1. RAll; herhaling
2. Smart Pointers; wat?
3. Smart Pointers; hoe?
4. Smart Pointers; de perfecte oplossing?
5.

6. Creational Design Patterns; wat?
7. CDP; Singleton
8. CDP; Factory Pattern

RAII; HERHALING

- Resource Acquisition Is Initialization
- Vorm van resource management
- Gebonden aan lifetime van een object

```
1  class FileHandler{
2  private:
3      std::ofstream file; // gemanagede resource
4  public:
5      // Resource Acquisition
6      FileHandler(const std::string& filename){ ... } // Open file
7
8      // Resource Release
9      ~FileHandler(){ ... } // Close file
10 }
```

RAII; STAPJE VERDER!

```
1  template<typename T>
2  class PointerHandler{
3  private:
4      T* ptr;
5  public:
6      explicit PointerHandler(T* resource = nullptr) : ptr(resource){
7          std::cout << "Resource managed!" << std::endl;
8      }
9
10     ~PointerHandler(){
11         delete ptr;
12         std::cout << "Resource deleted!" << std::endl;
13     }
14
15     ... //Operator overloads
16 }
17
18 int main(){
19     {
20         PointerHandler<int> intptr(new int(42));
21     }
22 }
```

SMART POINTERS; WAT?

- ... Dat is (deels) wat smartpointers zijn
- Taal feature voor RAI bij het managen van dynamische objecten
- (Met wat extra slimmigheden)

3 typen Smart Pointers, uit de `<memory>` header:

- `std::unique_ptr`
 - **Één eigenaar** van de resource; geen kopieën mogelijk
 - Eigenaarschap **doorgeven** d.m.v. `std::move`
- `std::shared_ptr`
 - **Meerdere eigenaren** van de resource; kopieën mogelijk
 - Maakt gebruik van **Reference Counting**
- `std::weak_ptr`
 - Specifiek voor gebruik met `std::shared_ptr`
 - Draagt **niet** bij aan Reference Counting

SMART POINTERS; HOE?

```

1 class Res {
2 private:
3     std::string name;
4 public:
5     Res(const std::string& name) : name(name){ ... }
6     ~Res(){ ... }
7     void sayHello() const { ... }
8 };
9
10 int main(){
11     // 3. std::weak_ptr: Geen eigenaarschap
12     {
13         std::shared_ptr<Res> sharedRes = std::make_shared<Res>("Weak");
14         std::weak_ptr<Res> weakRes = sharedRes; // Geen eigenaarschap
15         if (auto lockedRes = weakRes.lock()) { // Controleer of resource nog bestaat
16             lockedRes->sayHello();
17         }
18         sharedRes.reset(); // Resource vrijgeven
19         if (weakRes.expired()) {
20             std::cout << "Resource is niet meer beschikbaar.\n";
21         }
22     }
23 }

```

SMART POINTERS; DE PERFECTE OPLOSSING?

- Helaas :(
- Unique Ptrs zijn ★ perfect ★
- Weak Ptrs 'look'
- Shared Ptrs gebruiken **éxtra geheugen** (reference counting)
- Shared Ptrs vragen **íets meer performance** voor reference counting
- Weak Ptrs geen extra overhead, maar hebben Shared Ptrs nodig
- Shared Ptr reference counting is thread safe, maar resource access niet
- Shared Ptrs kunnen Cyclic Dependency veroorzaken -> Memory Leak

SMART POINTERS; DE PERFECTE OPLOSSING?

- Shared Ptrs kunnen Cyclic Dependency veroorzaken

```
1 class B; // Forward declaration
2
3 class A {
4 public:
5     std::shared_ptr<B> b_ptr; // Shared pointer naar B
6     ~A() { std::cout << "A verwijderd\n"; }
7 };
8
9 class B {
10 public:
11     std::weak_ptr<A> a_ptr; // Weak pointer naar A
12     ~B() { std::cout << "B verwijderd\n"; }
13 };
14
15 int main() {
16     auto a = std::make_shared<A>();
17     auto b = std::make_shared<B>();
18
19     a->b_ptr = b; // A wijst naar B
20     b->a_ptr = a; // B wijst naar A, maar via weak_ptr
21     // Nu worden beide objecten correct vrijgegeven, geen memory leak :)
22 }
```


**
**

CREATIONAL DESIGN PATTERNS; WAT?

- Voor het creëren van objecten
- Promoot flexibiliteit en DRY principe

Grofweg zijn er 5:

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

CREATIONAL DESIGN PATTERNS; WAT?

- Voor het creëren van objecten
- Promoot flexibiliteit en DRY principe

Grofweg zijn er 5:

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

CDP; SINGLETON

- Verzekert dat er maar één instantie van dat object is
- Biedt global access naar een resource

```
1 class Singleton{
2 private:
3     ... // Een resource waar je toegang toe wilt controleren
4     Singleton(){ ... }
5     static std::unique_ptr<Singleton> instance;
6 public:
7     static Singleton& getInstance( ... ){
8         if (!instance){
9             instance = std::make_unique<Singleton>();
10        }
11        return *instance;
12    }
13    void doSomething(){ ... }
14 };
15
16 int main(){
17     Singleton& A = Singleton::getInstance();
18     A.doSomething();
19     Singleton& B = Singleton::getInstance(); // Toch niet magisch.. :(
20     B.doSomething();
21 }
```

CDP; SINGLETON

- Zonder dure memory management;

```
1  class Singleton {
2  private:
3      Singleton() { ... }
4  public:
5      static Singleton& getInstance() {
6          static Singleton instance; // Gegarandeerd maar één keer
7          return instance;
8      }
9      void doSomething() { ... }
10 };
11
12 int main() {
13     Singleton& singleton = Singleton::getInstance();
14     singleton.doSomething();
15 }
```

CDP; SINGLETON

Krachtig, maar...

Nadelen, dus niet zonder controverse:

- Single Responsibility Principle
- Gevoelig voor Undefined Behaviour
- Globale State / Side Effects
- Hidden Dependencies
- Mogelijk niet Thread Safe

CDP; FACTORY PATTERN

- Stel, we willen een aantal objecten creëren

```
1  class Colour; // Forward Declaration
2
3  class Drawable{
4  private:
5      std::string Name; // Default onderdelen van een drawable object
6      std::vector Pos;
7      Colour colour;
8  public:
9      Drawable(std::vector Pos, Colour colour){ ... }
10     virtual void draw const = 0;
11 }
12
13 class Rectangle : public Drawable{ ... }; // Implementeert specifieke onderdelen voor dit object
14 class Circle : public Drawable{ ... };
15 class Line : public Drawable{ ... };
16
17 int main(){
18     Rectangle rect1 = Rectangle( ... );
19     Rectangle rect2 = Rectangle( ... );
20     Circle circ1 = Circle( ... );
```

KLEINE UITSTAP; POLYMORFISME

- Het concept dat je een derived class pointer in een base class pointer kwijt kunt

```
1 class Drawable{ ... };
2 class Rectangle : public Drawable{ ... };
3 class Circle : public Drawable{ ... };
4
5 int main(){
6     std::vector<std::unique_ptr<Drawable>> drawables;
7     drawables.push_back(std::make_unique<Rectangle>( ... ));
8     drawables.push_back(std::make_unique<Circle>( ... ));
9
10    for(auto *obj : drawables){
11        obj->draw();
12    }
13 }
```


CDP; FACTORY PATTERN

```
1  Drawable* makeDrawable(std::ifstream & input){
2      std::string name;
3      std::vector pos;
4      Colour colour;
5
6      input >> name >> pos >> colour;
7
8      if (name == "Circle"){ return new Circle( ... ); }
9      if (name == "Rectangle"){ return new Rectangle( ... ); }
10     ...
11 }
12
13 int main(){
14     std::vector<Drawable*> drawables;
15
16     while(true){
17         drawables.push_back(makeDrawable(input));
18     }
19
20     for(auto drawable : drawables){
21         drawable->draw();
22     }
23 }
```

– E inde –