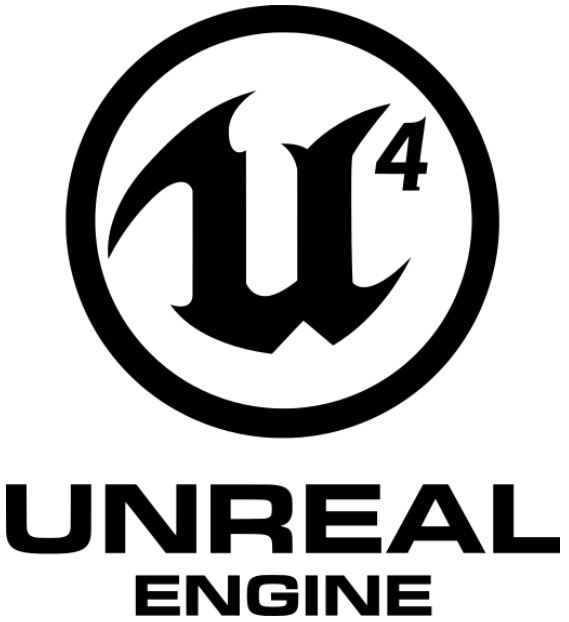
Learning modern C++?
Start with Cpp Core Guidelines

https://isocpp.github.io/CppCoreGuidelines/

- Kamil Grodecki, ONE MORE LEVEL S.A.
- God’s Trigger → Ghostrunner
- Unity 5.x/2017.x → Unreal Engine 4.2x
- C# → C++



C++98 → C++11

C++ to coś więcej niż „C z klasami”

- RAII (Resource Acquisition Is Initialization)
- unique_ptr / shared_ptr
- std::array
- {} – uniform initialization
- auto
- static_assert
- Wyrażenia lambda
- … i wiele innych

CppCon 2015: Kate Gregory “Stop Teaching C"

```cpp
void oldStyle(int src[42])
{
    // Error prone
    int dst[42];
    for (int i = 0; i < 42; ++i)
    {
        dst[i] = src[i];
    }
}
```

```cpp
void modernStyle(std::array<int, 42> src)
{
    // Deep copy, static assert
    std::array<int, 42> dst = src;
}
```

- Krótszy kod
- Porównanie długości tablicy w czasie kompilacji
- Zerowy koszt
- Dodatkowe funkcje kontenera

```cpp
void foo()
{
    Widget* widget = new Widget();
    // ...
    if (widget->x < 0)
    {
        return; // Resource leak!
    }
    // ...
}
```

```cpp
void fooRAII()
{
    std::unique_ptr<Widget> widget = std::make_unique<Widget>();
    // ...
    if (widget->x < 0)
    {
        return; // Widget end life here
    }           // Call destructor
    // ...
}
```

- Wszystkie ścieżki są automatycznie obsłużone
- Zerowy koszt (w porównaniu do new/delete)

Czym jest Cpp Core Guidelines?

Turn ON syntax

P: Philosophy

The rules in this section are very general.

Philosophy rules summary:

Przykładowa zasada

P.8: Don’t leak any resources

Reason Even a slow growth in resources will, over time, exhaust the availability of those resources. This is particularly important for long-running programs, but is an essential piece of responsible programming behavior.

Example, bad

```
void f(char* name)
{
    FILE* input = fopen(name, "r");
    // ...
    if (something) return;   // bad: if something == true, a file handle is leaked
    // ...
    fclose(input);
}
```

Prefer RAII:

```
void f(char* name)
{
    ifstream input {name};
    // ...
    if (something) return;   // OK: no leak
    // ...
}
```

Note A leak is colloquially "anything that isn't cleaned up." The more important classification is "anything that can no longer be cleaned up." For example, allocating an

Enforcement

- Look at pointers: Classify them into non-owners (the default) and owners. Where feasible, replace owners with standard-library resource handles (as in the example above). Alternatively, mark an owner as such using `owner` from the GSL.
- Look for naked `new` and `delete`
- Look for known resource allocating functions returning raw pointers (such as `fopen`, `malloc`, and `strdup`)

Dlaczego akurat Cpp Core Guidelines?

- Rozwiązywanie konfliktów
- Nauka mechanizmów nowoczesnego C++
- Poprawienie jakości kodu

Rozwiązywanie konfliktów

- Cpp Core Guidelines – odpowiednik STL z zakresu pisania „dobrego” kodu
- Przemyślany zestaw zasad
- Potwierdzony przykładami
- Edytorzy: Bjarne Stroustrup, Herb Sutter

C.131: Avoid trivial getters and setters

Reason A trivial getter or setter adds no semantic value; the data item could just as well be `public`.

Example

```
class Point {   // Bad: verbose
    int x;
    int y;
public:
    Point(int xx, int yy) : x{xx}, y{yy} { }
    int get_x() const { return x; }
    void set_x(int xx) { x = xx; }
    int get_y() const { return y; }
    void set_y(int yy) { y = yy; }
    // no behavioral member functions
};
```

Nauka mechanizmów nowoczesnego C++

- Nowe mechanizmy – dla wszystkich wersji nowoczesnego C++
- Baza pojęć
- Proste i zrozumiałe przykłady

ES.71: Prefer a range-for-statement to a for-statement when there is a choice

```
for (int i = 0; i < list.size(); ++i)
{
    list[i].foo(); //For-statement
}
```

```
for (auto &x : list)
{
    x.foo(); //Range-for-statement
}
```

Reason Readability. Error prevention. Efficiency.

Unikanie błędów

```
for (int i = 0; i < list.size(); ++i)
{
    list[i + 1].foo(); //Index out of range
}
```

Czytelność

```
for (auto &x : list)
{
    x.foo(); //Range-for-statement
}
```

```
for (const auto& x : list)
```

```
for (auto x : list)
```

```
for (const auto x : list)
```

Wydajność

TRADITIONAL

```
sub rcx, rdx ; rcx = end-begin
mov rax, rcx
shr rax, 2   ; (end-begin)/4
je .L4
add rcx, rdx
xor eax, eax
```

RANGE

```
xor eax, eax
cmp rdx, rcx ; begin==end?
je .L4
```

CppCon 2017: Matt Godbolt “What Has My Compiler Done for Me Lately? Unbolting the Compiler's Lid”

ES.23: Prefer the {}-initializer syntax

- **Couple of ways to initialize an int:**

```
int i1;                 // undefined value
int i2 = 42;            // note: inits with 42
int i3(42);             // inits with 42
int i4 = int();         // inits with 0
int i5{42};             // inits with 42
int i7{};               // inits with 0
int i6 = {42};          // inits with 42
int i8 = {};            // inits with 0
auto i9 = 42;           // inits int with 42
auto i10{42};           // C++11: std::initializer_list<int> , C++14: int
auto i11 = {42};        // inits std::initializer_list<int> with 42
auto i12 = int{42};     // inits int with 42
int i13();              // declares a function
int i14(7, 9);          // compile-time error
int i15 = (7, 9);       // OK, inits int with 9 (com
int i16 = int(7, 9);    // compile-time error
auto i17(7, 9);         // compile-time error
auto i18 = (7, 9);      // OK, inits int with 9 (comma operator)
auto i19 = int(7, 9);   // compile-time error
```

don't use ( ) in initializations

CppCon 2018: Nicolai Josuttis “The Nightmare of Initialization in C++”

Initializacja kontenerów

```cpp
void foo()
{
    // Empty vector
    std::vector<int> vec();

    // Creates 42 zero-initialized elements
    std::vector<int> vec(42);

    // Creates 10 elements with value '2'
    std::vector<int> vec(10, 2);

    // Creates vector with values: '2', '10', '42'
    std::vector<int> vec{ 2, 10, 42 };
}
```

- Uniform initialization zapewnia, że nie zostanie wywołany konstruktor
- Wymagany do stworzenia std::vector lub std::array o 1-2 elementach

Niejawna konstrukcja typów zwracanych

```
struct Point
{
    int x{ 0 };
    int y{ 0 };

    Point() = default;
    Point(int newX, int newY) : x(newX), y(newY)
    {
    }
};
```

```
Point getPoint()
{
    return Point(10, 5);
}
```

```
Point getPointUniform()
{
    return{ 10, 5 };
}
```

- Pominięcie nazwy typu przy wywołaniu konstruktora
- Krótszy kod
- Przydatne do skrócenia długości argumentów funkcji przy wywołaniu

```
// Function declaration
Point pointFunction();

// Variables
Point pointA(10, 10);
Point pointB{};
Point pointC;
```

- Zmienne globalne – błędna deklaracja funkcji
- IDE zaproponuje nam zdefiniowanie funkcji – rzadkie źródło błędów

Poprawienie jakości kodu

- Pisanie czystego kodu
- Unikanie nieoptymalnych rozwiązań
- Unikanie podstawowych pomyłek przy projekcie
- Wykorzystywanie dostępnych mechanizmów zamiast „odkrywać koło na nowo”

I.23: Keep the number of function arguments low

Discussion The two most common reasons why functions have too many parameters are:

1. *Missing an abstraction*. There is an abstraction missing, so that a compound value is being passed as individual elements instead of as a single object that enforces an invariant. This not only expands the parameter list, but it leads to errors because the component values are no longer protected by an enforced invariant.

void SetEnemyBaseBlackboardValues(APawn* EnemyCharacter, APawn* PlayerCharacter, EEnemyState EnemyCharacterState, EEnemyBasicBehavior EnemyCharacterBasicBehavior,EEnemyPatrolType EnemyCharacterPatrolType, bool IsPlayerCharacterVisible, bool IsPlayerCharacterAudible, const FVector& LastSeenPlayerCharacterLocation,const FVector& LastHeardPlayerCharacterLocation, float DistanceToPlayerCharacter, const FVector& NextEnemyPatrolLocation,float EnemyWeaponRange, float EnemyAlertTimeBeforeAbleToAttack, float EnemyWeaponChargeTime);

void SetEnemyBaseBlackboardValues(FEnemyBlackboardData Data);

F.15: Prefer simple and conventional ways of passing information

<table>
<tr><th></th><th>Cheap or impossible to copy (e.g., int, unique_ptr)</th><th>Cheap to move (e.g., vector<T>, string)
or Moderate cost to move (e.g., array<vector>, BigPOD)
or Don’t know (e.g., unfamiliar type, template)</th><th>Expensive to move
(e.g., BigPOD[],
array<BigPOD>)</th></tr>
<tr><td>Out</td><td colspan="2">X f()</td><td rowspan="2"></td></tr>
<tr><td>In/Out</td><td colspan="2">f(X&)</td></tr>
<tr><td>In</td><td rowspan="2">f(X)</td><td rowspan="2">f(const X&)</td><td rowspan="2"></td></tr>
<tr><td>In & retain “copy”</td></tr>
</table>

“Cheap” ≈ a handful of hot int copies
“Moderate cost” ≈ memcpy hot/contiguous ~1KB and no allocation

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

CppCon 2018: Kate Gregory “What Do We Mean When We Say Nothing At All?”

Źródła i inspiracje

- https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

- CppCon 2017: Kate Gregory “10 Core Guidelines You Need to Start Using Now” - https://www.youtube.com/watch?v=XkDEzfpdcSg
- CppCon 2015: Kate Gregory “Stop Teaching C" - https://www.youtube.com/watch?v=YnWhqhNdYyk
- [ES. 23] CppCon 2018: Nicolai Josuttis “The Nightmare of Initialization in C++” - https://www.youtube.com/watch?v=7DTlWPgX6zs
- [F.15] CppCon 2018: Kate Gregory “What Do We Mean When We Say Nothing At All?” - https://www.youtube.com/watch?v=kYVxGyido9g
- [ES.71] CppCon 2017: Matt Godbolt “What Has My Compiler Done for Me Lately? Unbolting the Compiler's Lid” + https://www.youtube.com/watch?v=bSkpMdDe4g4

Dziękuję!

Kamil Grodecki
k.grodecki@omlgames.com

https://github.com/komilll/SpreadIT