
WARSZTAT.GD
PROGRAMOWANIE GIER

# TWORZENIE WYDAJNEGO KODU C++ W PODEJŚCIU ZORIENTOWANYM NA DANE


Tomasz Dąbrowski


Adam Sawicki

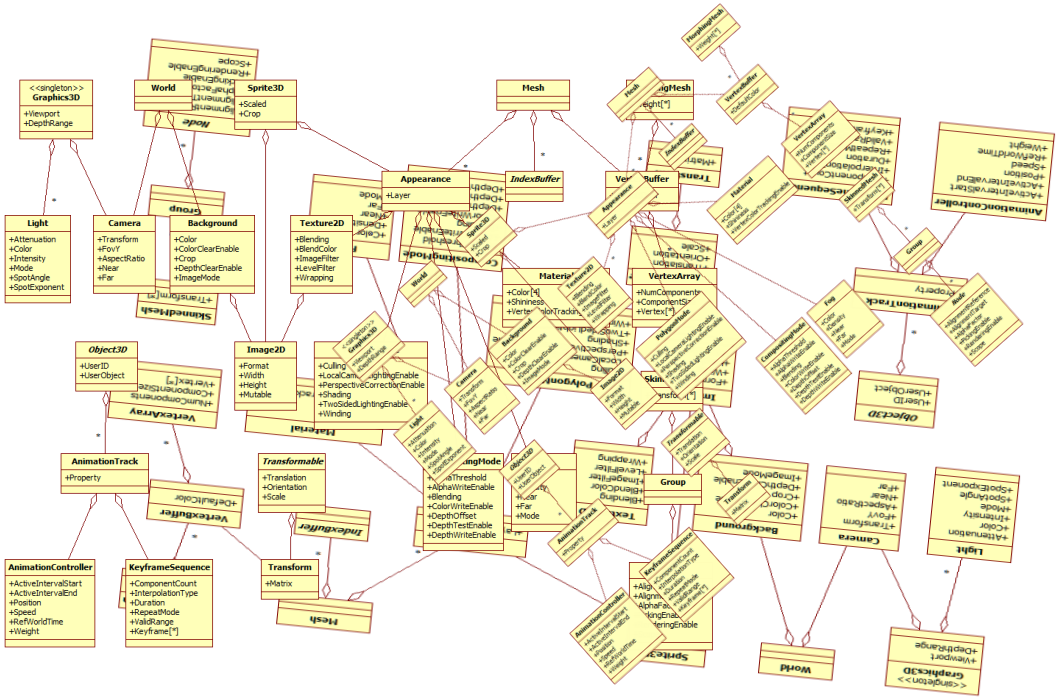
VIII Ogólnopolska Konferencja Inżynierii Gier Komputerowych Siedlce, marzec 2011

# TOMASZ DĄBROWSKI


- dabroz@scythe.pl
- http://dabroz.scythe.pl
- @dabrozPL
- Programista
  - Politechnika Warszawska – Informatyka
  - Developer iOS w inFullMobile
  - Silnik Unified Theorem Engine
  - Gra LVX Techdemo

# ADAM SAWICKI

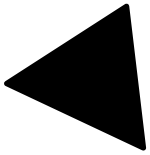
- adam@asawicki.info
- http://www.asawicki.info
- @Reg__
- Programista
  - Politechnika Częstochowska – Informatyka
  - Praktyki w siedzibie Microsoft w Redmond, USA
  - AquaFish 2 (gra wydana przez PLAY-publishing)
  - Metropolis Software (They – shooter na PC i X360)
  - Cyfrowy Polsat S.A.



# PROGRAMOWANIE OBIEKTOWE

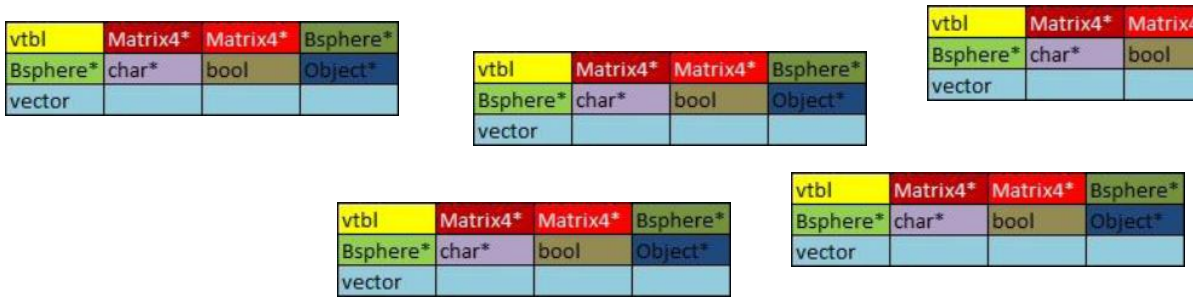
# PROGRAMOWANIE OBIEKTOWE

- Teoria: program składa się z **klas**
  - Łączą **dane** (pola) i **kod** (metody)
  - Modelują byty z dziedziny problemu
  - Są od siebie niezależne
  - Nadają się do ponownego użycia
- Założenia
  - **Enkapsulacja** (hermetyzacja) – udostępnienie interfejsu, ukrycie implementacji
  - **Polimorfizm** i dziedziczenie – używanie obiektu abstrakcyjnego bez rozróżniania konkretnych typów


# PROGRAMOWANIE OBIEKTOWE

- Wady
  - Czy mogą istnieć klasy całkowicie niezależne od innych?
  - Czy zdarzyło ci się użyć klasy w innym projekcie bez żadnych zmian?
  - Rzadko interesuje nas pojedynczy obiekt, zwykle mamy ich wiele
  - Nie mamy kontroli nad tym, co się gdzie odwołuje
  - Dane są porozrzucane po pamięci – słaba **lokalność odwołań**
  - Obiektowy kod trudno się **zrównolegla**




[Pitfalls]

# PROGRAMOWANIE OBIEKTOWE

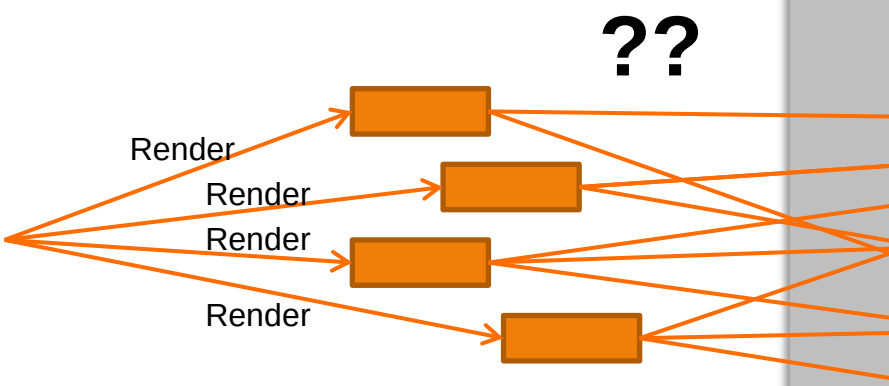
- Rozwiązanie
  - Myśl o **danych**
  - Używaj dobrze dobranych **struktur danych**
  - Organizuj kod tak, aby jawnie **przetwarzał dane** wejściowe na wyjściowe


[Pitfalls]

# Przykład

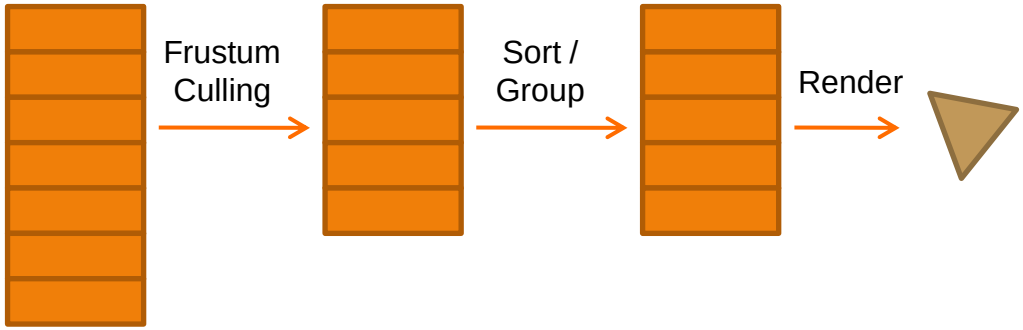
:(

```
class BaseObject {
private:
  float3 position;
public:
  virtual void Render() = 0;
};
```




:)

```
struct RenderBatch {
  float3 position;
  Mesh *mesh;
  Material *material;
};
```



# AOS KONTRA SOA

## AoS – Array of Structures

```
struct Particle
{
  float4 position;
  float4 velocity;
  float4 color;
  float size;
  float orientation;
};

struct ParticleSystem
{
  uint32_t count;
  Particle *particles;
};
```

## SoA – Structure of Arrays

```
struct ParticleSystem
{
  uint32_t count;
  float4 *positions;
  float4 *velocities;
  float4 *colors;
  float *sizes;
  float *orientations;
};
```

- SoA może być szybsze dzięki lepszemu wykorzystaniu pamięci cache
  - Grupuje dane według wzorca użycia
  - Oddziela dane „gorące” od „zimnych”

[DivergentCoder]

# AOS KONTRA SOA

Rozwiązanie pośrednie: oddzielenie danych „gorących” od „zimnych”

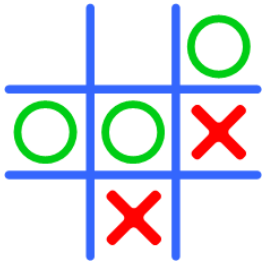
```
struct Entity {
  float4 position;
  float4 velocity;
  EntityEx *extra_data;
};

struct EntityEx {
  std::string name;
  std::vector<Mesh*> meshes;
  std::map<std::string, boost::any> properties;
};
```

# PROGRAMOWANIE OBIEKTOWE – PRZYKŁAD

**Gra w kółko i krzyżyk** – podejście obiektowe

1. Znajdujemy klasy identyfikując rzeczowniki



Krzyżyk

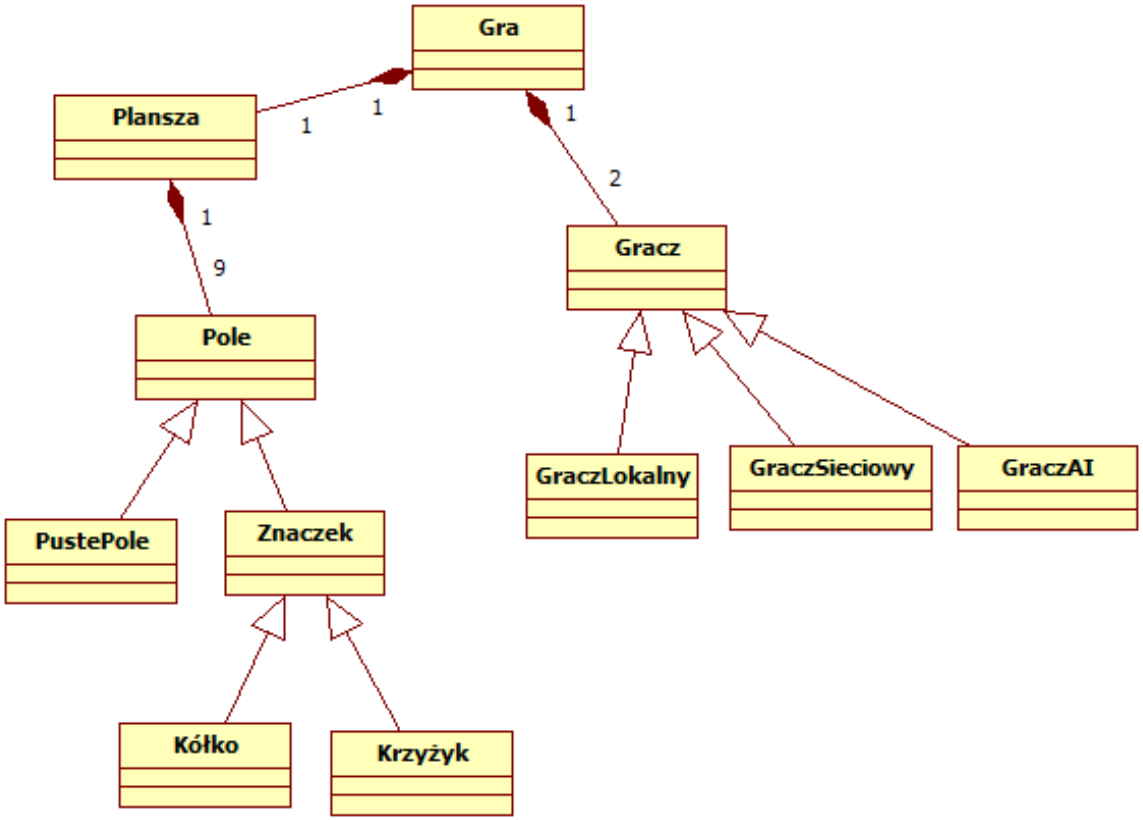
Gracz Plansza

Kółko

Gra

# PROGRAMOWANIE OBIEKTOWE – PRZYKŁAD

2. Projektujemy powiązania między klasami



# PROGRAMOWANIE OBIEKTOWE – PRZYKŁAD

3. Implementujemy klasy

- Co umieścić w tych klasach, żeby nie były puste ??
- W której klasie umieścić potrzebne dane i kod ??

- **Analysis Paralysis** – poszukiwanie idealnego rozwiązania

# PROGRAMOWANIE OBIEKTOWE – PRZYKŁAD

3. Implementujemy klasy

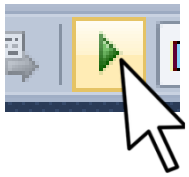
- Co umieścić w tych klasach, żeby nie były puste ??
- W której klasie umieścić potrzebne dane i kod ??

- **Analysis Paralysis** – poszukiwanie idealnego rozwiązania

# DATA-ORIENTED DESIGN

- Alternatywne podejście: **Data-Oriented Design (DOD)**
  - Myśl o strukturze danych w pamięci
- Pytanie: Jakie dane powinna przechowywać gra?
  - Tablica 3 x 3 pól
  - Każde pole jest w jednym ze stanów: puste, kółko, krzyżyk
  - Czyj jest teraz ruch: gracza 1 lub 2

```
enum POLE {
  POLE_PUSTE,
  POLE_KOLKO,
  POLE_KRZYZYK,
};
POLE Plansza[3][3];
int CzyjRuch;
```

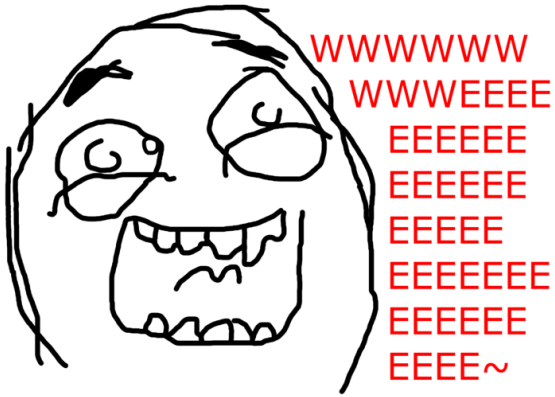


# DOD – PRZYKŁAD

- Następnie pomyśl, co po kolei kod powinien robić z tymi danymi
  - Wykonanie ruchu przez gracza
  - Sprawdzenie, czy ruch jest prawidłowy
  - Wstawienie symbolu do tablicy
  - Sprawdzenie, czy gracz wygrał
  - Rozpoczęcie tury przeciwnego gracza

# DOD – PRZYKŁAD

- Następnie pomyśl, co po kolei kod powinien robić z tymi danymi
  - Wykonanie ruchu przez gracza
  - Sprawdzenie, czy ruch jest prawidłowy
  - Wstawienie symbolu do tablicy
  - Sprawdzenie, czy gracz wygrał
  - Rozpoczęcie tury przeciwnego gracza



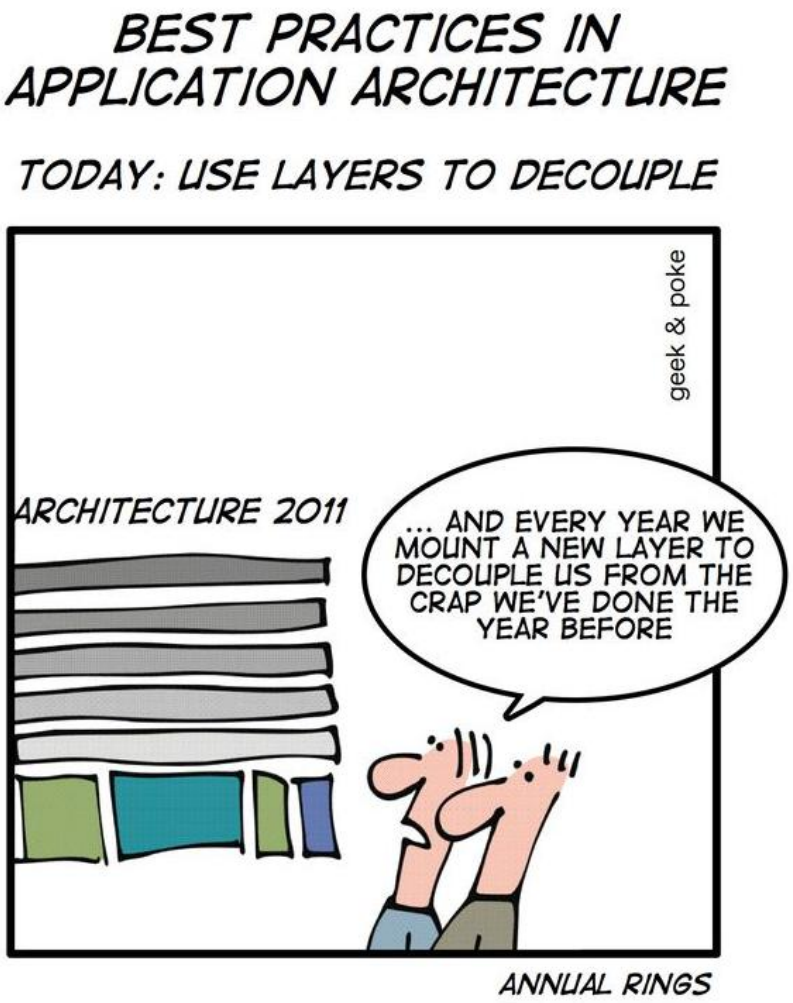
# PROGRAMOWANIE OBIEKTOWE – WNIOSKI

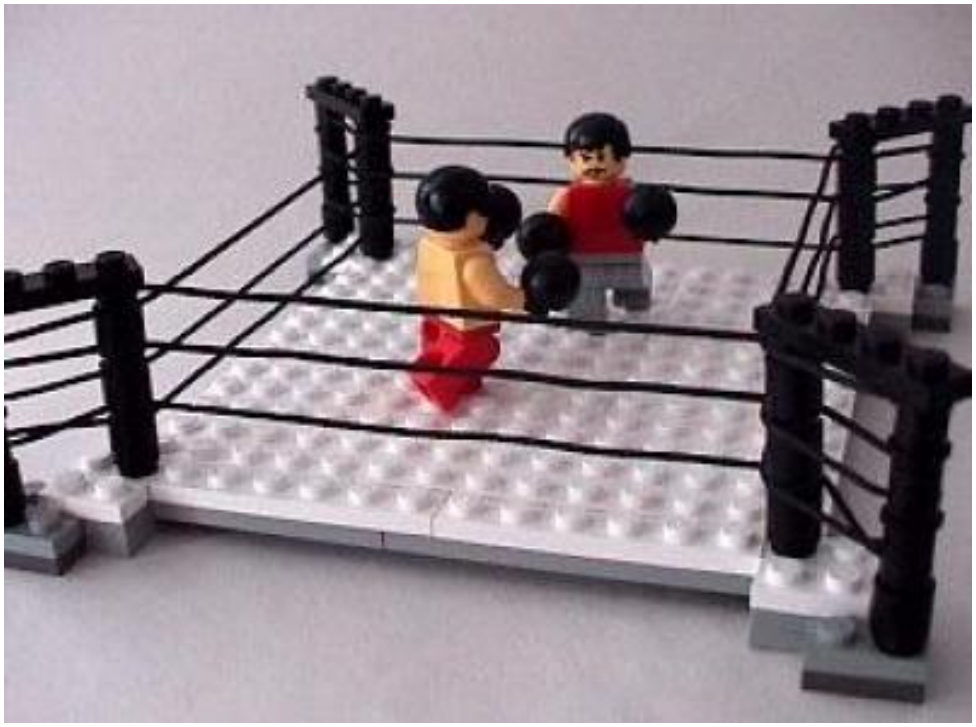
- Programowanie obiektowe **nie jest idealne**
- Można nawet stwierdzić, że **nie spełniło swoich założeń**
  - Modelowanie rzeczywistych obiektów? Co modeluje manager, helper, listener, obeserver, locker i inne -ery?
  - Źle użyte działa przeciwko wydajności, jak też prostocie i czytelności kodu
- Warto je stosować, ale **z rozwagą**
  - Znaj i doceniaj inne możliwości
  - Nie szukaj gotowych „klocków” uciekając od myślenia

# WARSTWY

*The object-oriented version of "Spaghetti code" is, of course, "Lasagna code". (Too many layers.)* – Roberto Waltman

*All problems in computer science can be solved by another level of indirection… Except for the problem of too many layers of indirection.* – David Wheeler

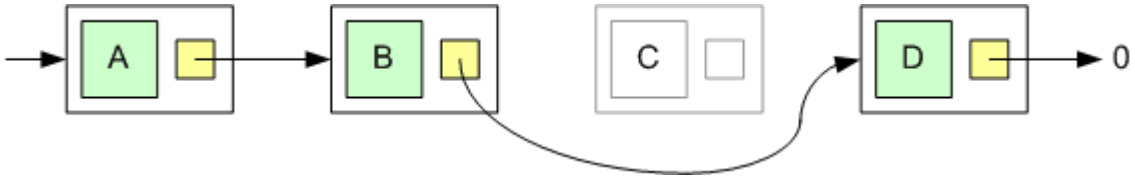




# LISTA VS WEKTOR

# LISTA CZY WEKTOR

- Potrzebujemy kolekcji, którą:
  - będziemy często iterowali,
  - będziemy czasem dodawali nowe elementy,
  - będziemy czasem usuwali dowolne elementy,
  - kolejność nie jest istotna.
- „Oczywistym” wyborem wydaje się lista łączona (np. std::list) (ze względu na usuwanie w O(1))



# LISTA CZY WEKTOR

- Lista łączona ma wady:
  - Dynamiczna alokacja osobno każdego elementu (alokacja jest wolna)
  - Elementy są rozrzucone po pamięci (cache miss)
- Rozwiązanie: tablica (np. std::vector)

# LISTA KONTRA WEKTOR

- Program testowy:
  - N elementów
  - 2000 powtórzeń następującej sekwencji:
    - Przejście przez całą kolekcję
    - Dodanie jednego elementu
    - Usunięcie jednego elementu (losowego)
- Testowe klasy:
  - 4 bajtowa
  - 513 bajtowa

# ZAWODNICY

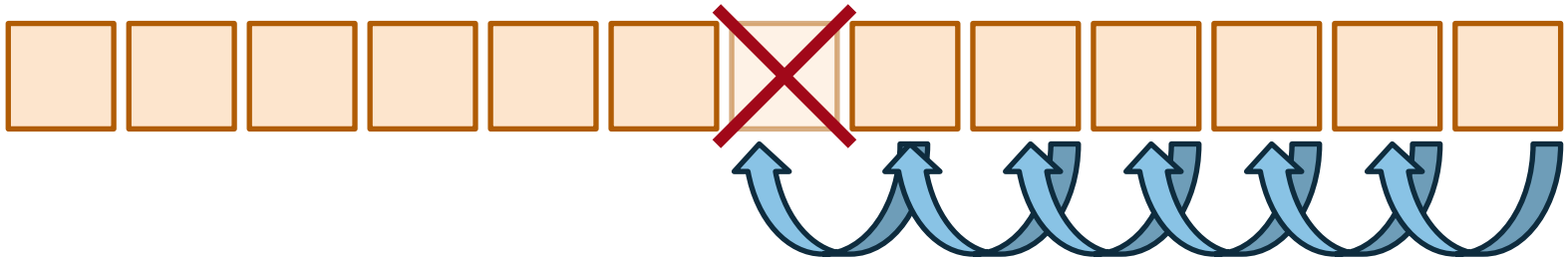
- Wszystkie testy na MSVC 2010, Release, wszystkie optymalizacje włączone
- #1: Zwykły wektor (std::vector z STL)
- #2: Zwykła lista (std::list z STL)
- A także…

# #3: LISTA Z ALOKACJAMI

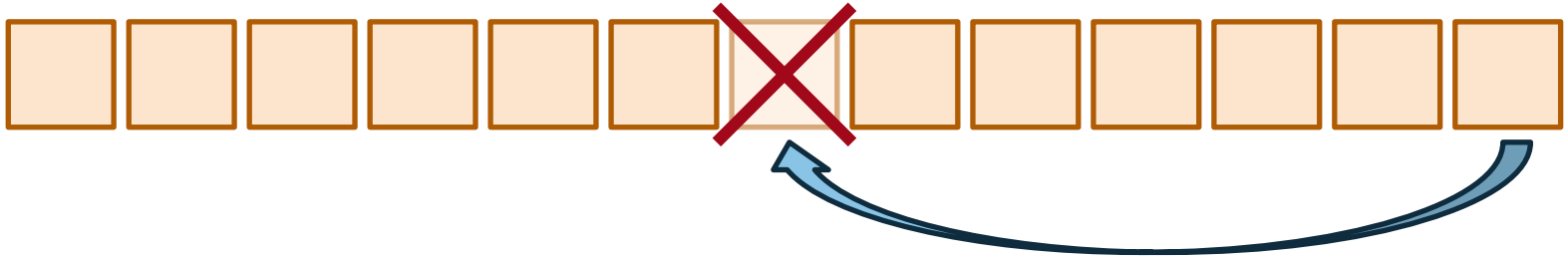
- Lista z alokacjami pomiędzy elementami
- Cel:
  - pogorszenie lokalności odwołań do cache
  - symulacja rzeczywistych warunków w grze

# #4: WEKTOR Z ZAMIANĄ

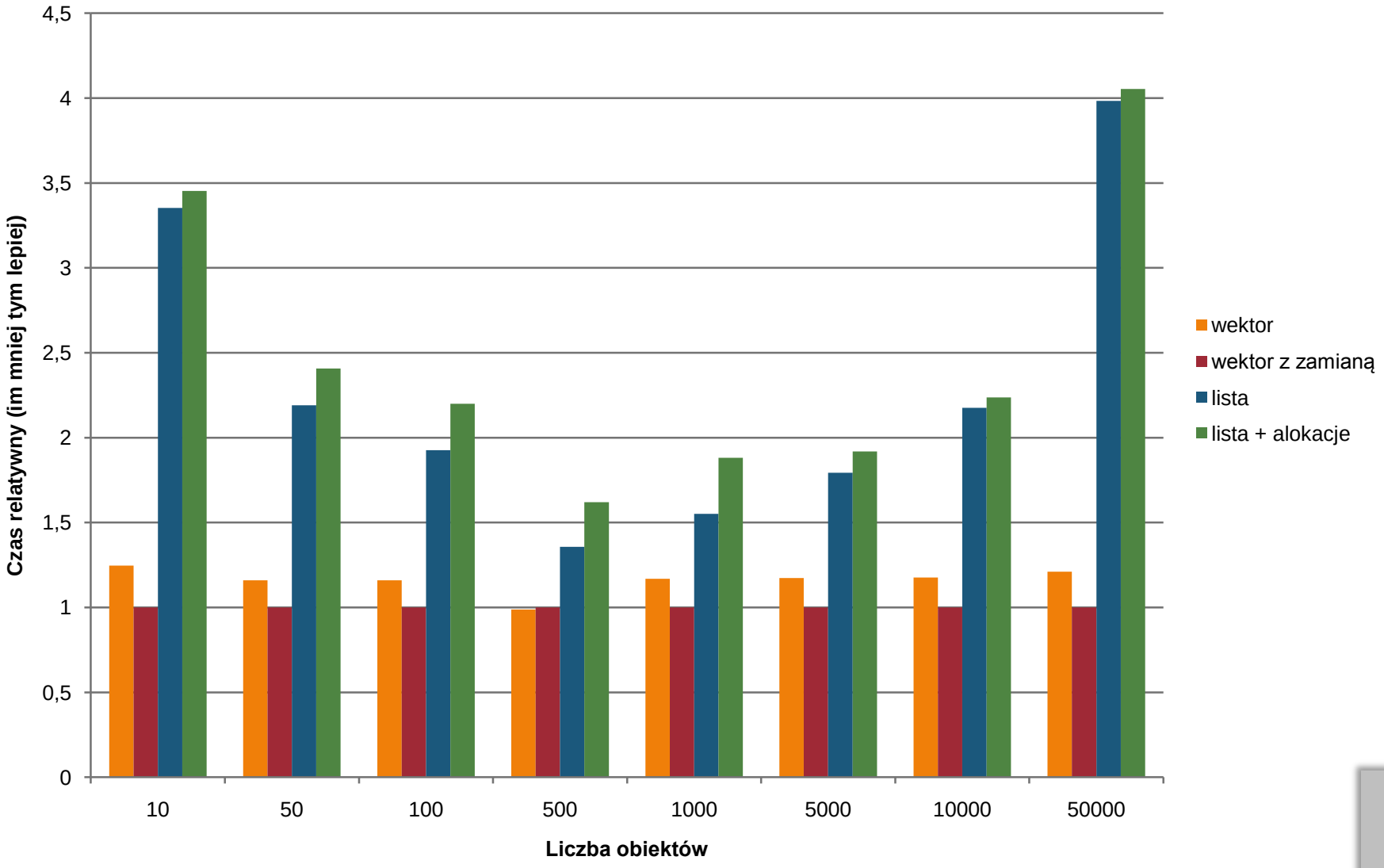
- Usuwanie w zwykłym wektorze



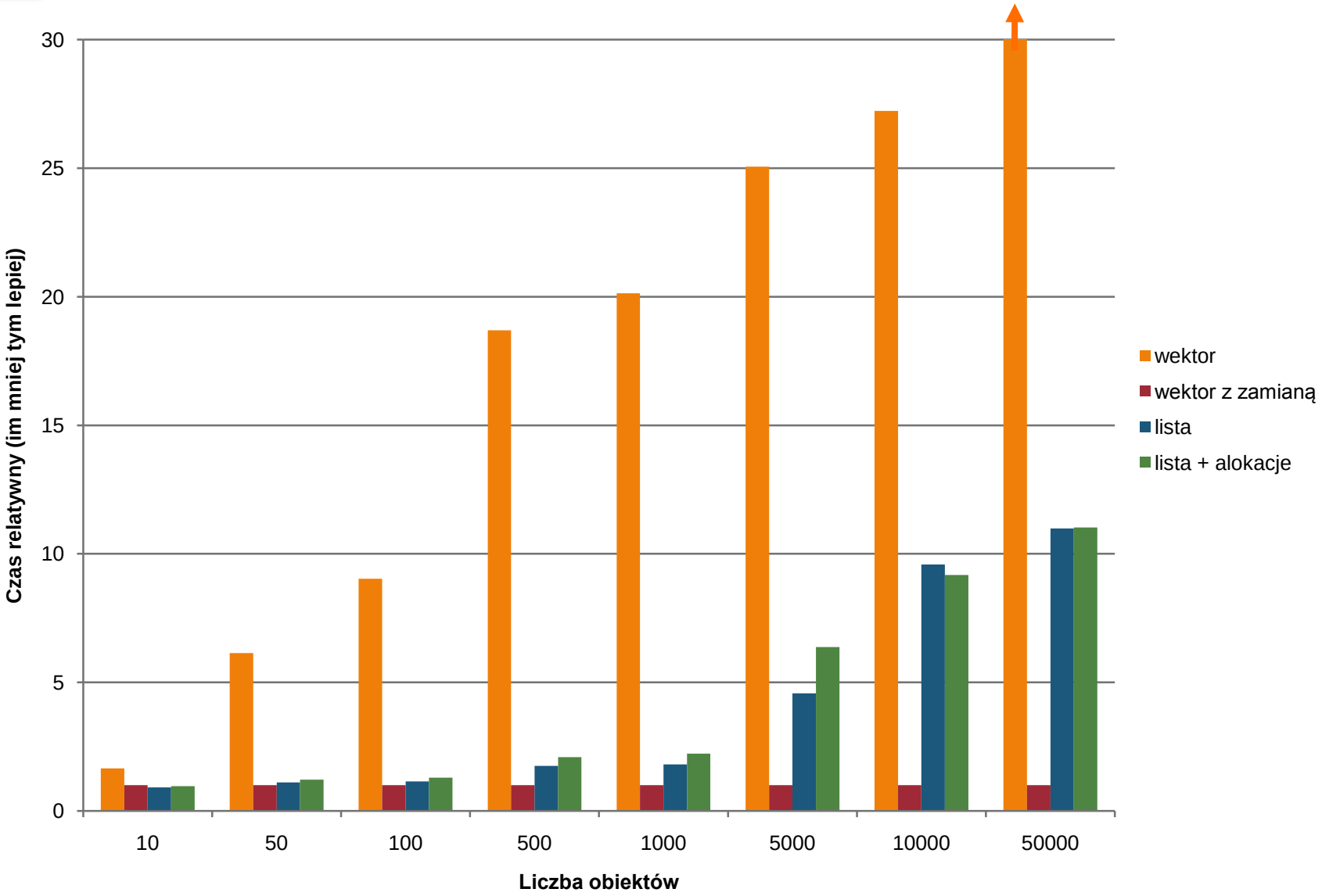
- Jeżeli nie zależy nam na kolejności, wystarczy zamienić usuwany element z ostatnim



# WYNIKI – OBIEKTY 4 BAJTOWE



# WYNIKI – OBIEKTY 513 BAJTOWE



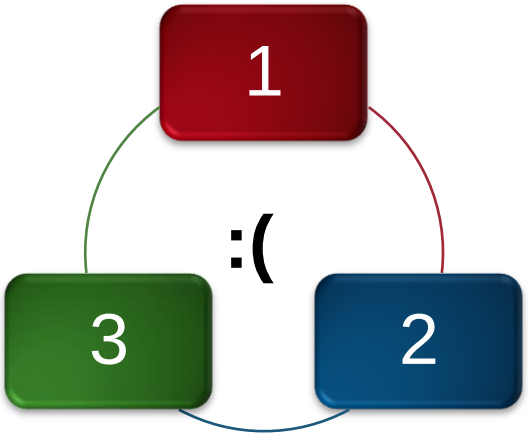
*Cat++, the cat with the seven legs and the wing, is a better cat, but also a scalable leg-oriented animal, not to mention excellent support for generic limbs.*

*Upgrade to a modern pet with more features today!*

[C++FQA]


# GENERALIZACJA

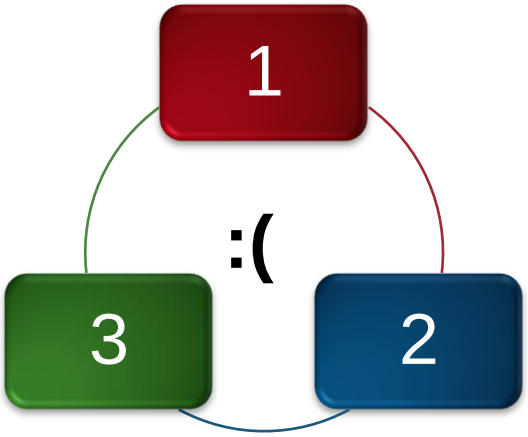
# GENERALIZACJA – BŁĘDNE KOŁO




1. Chcemy, żeby każdy pisany kod był jak najbardziej ogólny i uniwersalny,

   aby nie trzeba było wprowadzać w nim zmian, kiedy zmienią się wymagania.

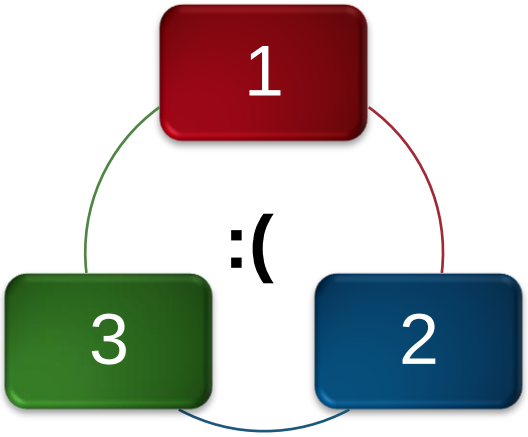
# GENERALIZACJA – BŁĘDNE KOŁO




2. Unikamy wprowadzania zmian w kodzie,

   bo każda, nawet miała zmiana, wymaga bardzo dużo pracy.

# GENERALIZACJA – BŁĘDNE KOŁO




3. Każda zmiana w kodzie wymaga bardzo dużo pracy,

   ponieważ wszystko piszemy jak najbardziej ogólnie i uniwersalnie.

# GENERALIZACJA – BŁĘDNE KOŁO

- Rozwiązanie
  - Pisz w sposób prosty, bezpośrednio to co ma być napisane i nie więcej.
  - Nie wyprzedzaj przyszłości. Kiedy pojawi się taka potrzeba, wtedy przerobisz kod na bardziej ogólny.
- Artyści i projektanci poziomów domagają się możliwości szybkiego testowania nowych pomysłów.
  - Programiści też powinni mieć do tego prawo! [C0DE517E]

# DZIEDZICZENIE

# DZIEDZICZENIE

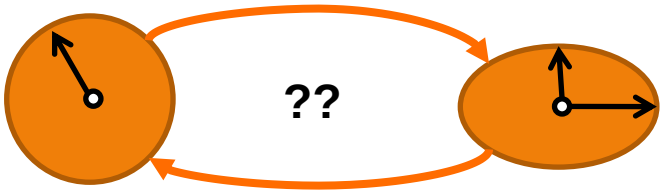
- Teoria
  - Modeluje relację „jest rodzajem”
  - Umożliwia abstrakcję i polimorfizm
  - Dzieli obszerny kod na fragmenty mające część wspólną

# DZIEDZICZENIE

- Wady
  - Kod jednej funkcjonalności jest **rozproszony** między wiele klas i plików
  - Tworzy ścisłe **powiązania** między klasami
  - Zmiana w jednym miejscu powoduje **nieoczekiwane efekty** w innych miejscach
  - Często **trudno zaprojektować** dobry układ klas, ich pól, metod i zależności między nimi

# DZIEDZICZENIE

- Przykład: **okrąg** i **elipsa**
  - **Matematyk** mówi: okrąg jest rodzajem elipsy, która ma dwa promienie równe!
  - **Programista obiektowy** mówi: elipsa jest rodzajem okręgu, który dodaje drugi promień!

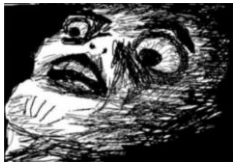


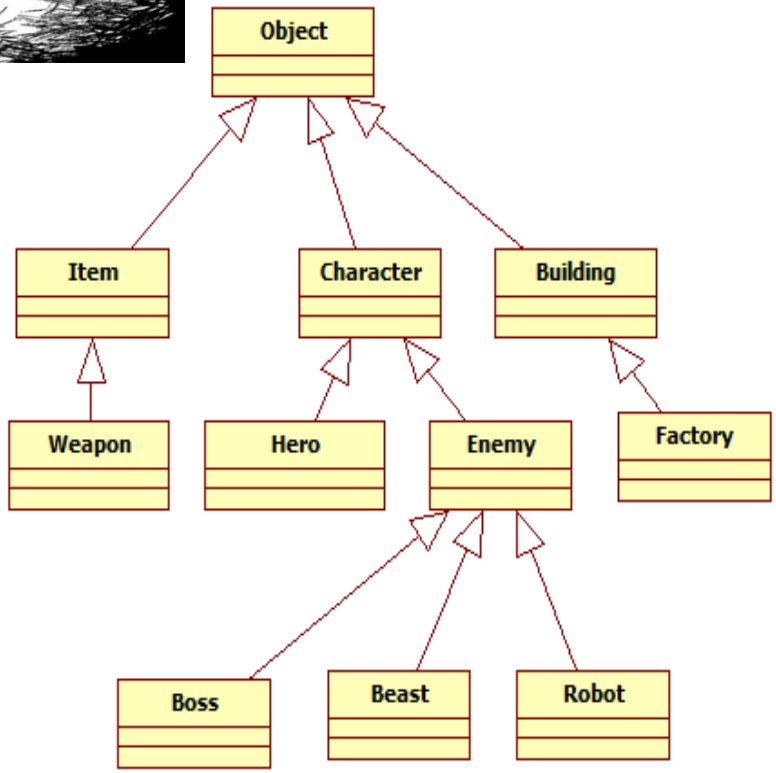
# DZIEDZICZENIE

- Rozwiązanie
  - Myśl o **danych**
  - Rozważ inne możliwości
    - enum Type
    - Kompozycja (agregacja)
    - Podejście sterowane danymi (data-driven)
    - Architektura komponentowa
  - Stosuj dziedziczenie jak najrzadziej, nie jak najczęściej
  - Patrz na dziedziczenie jak na mechanizm językowy, nie jak na ideę do modelowania każdej relacji „jest rodzajem”

# DATA-DRIVEN

## Podejście sterowane danymi






```
"Boss": {
  "Life": 1000,
  "Armor": 200,
  "Weapon": "Laser"
},

"Laser": {
  "ParticleEffect":
    "LaserEffect02",
  "Damage": 1000,
  "Duration": 2.5,
  "Cooldown": 0.7
}
```

# ARCHITEKTURA KOMPONENTOWA

- Encja jest prostym, beztypowym kontenerem na komponenty
  - Posiada tylko podstawowe dane opisujące każdy obiekt – nazwa, transformacja
  - Przechowuje opcjonalne komponenty różnego rodzaju
- Komponent danego rodzaju
  - Odpowiada za konkretny aspekt działania encji – np. rendering, dźwięk, fizyka, AI
  - Możliwe różne implementacje – od hierarchii klas po tablice struktur POD
  - Lepsza organizacja kodu, możliwość optymalizacji i zrównoleglania
- Przykład: Unity

[EvolveYourHierarchy]

# STATYCZNE STRUKTURY DANYCH

- Niekoniecznie **POD**
- Łatwa **serializacja**, kopiowanie przez memcpy
- **ID** zamiast wskaźników, stringi o **statycznej** długości

[Duck]

# STATYCZNE STRUKTURY DANYCH

- Niekoniecznie **POD**
- Łatwa **serializacja**, kopiowanie przez memcpy
- **ID** zamiast wskaźników, stringi o **statycznej** długości

**:(**

```
struct Object
{
  float4 position;
  std::string name;
  Object * parent;
  Material * material;
};
```

[Duck]

# STATYCZNE STRUKTURY DANYCH

- Niekoniecznie **POD**
- Łatwa **serializacja**, kopiowanie przez memcpy
- **ID** zamiast wskaźników, stringi o **statycznej** długości

**:(**

```
struct Object
{
  float4 position;
  std::string name;
  Object * parent;
  Material * material;
};
```

**:)**

```
struct Object
{
  float4 position;
  char name[32];
  uint32_t parentID;
  uint32_t materialID;
};
```

[Duck]

# ENKAPSULACJA

# ENKAPSULACJA

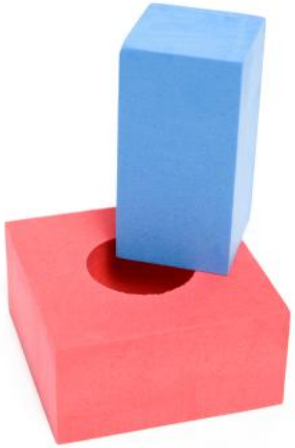
- Teoria
  - Udostępnia **interfejs**, ukrywa szczegóły **implementacji**
  - Pozwala **zmienić** implementację bez zmian w interfejsie i w innych częściach kodu

```
class Foo {
public:
  int GetValue() { return Value; }
  void SetValue(int v) { Value = v; }
private:
  int Value;
};
```

# ENKAPSULACJA

- Problem
  - Czym się różni **getter + setter** od uczynienia pola publicznym?
  - Czy naprawdę wierzysz, że możesz zmienić implementację tego pola (zmienić jego typ, wyliczać za każdym razem, pobierać z innego obiektu) **bez zmian** w interfejsie i w innych częściach kodu?

```
class Foo {
public:
  int GetValue() { return (int)Value; }
  void SetValue(int v) { Value = (float)v; }
private:
  float Value;
};
```



# ENKAPSULACJA

- Rozwiązanie
  - Nie bój się używać **struktur** z publicznymi polami tam, gdzie chodzi o paczkę danych
  - Rozważ inne metody dostępu do danych, np. parametry przekazywane podczas inicjalizacji, a potem dostępne **tylko do odczytu**
    - Descriptor, immutability

```
struct FooDesc {
  int Value;
};

class Foo {
public:
  Foo(const FooDesc &params);
  void GetParams(FooDesc &out_params);
};
```

# PRZEKAZYWANIE DANYCH

# PRZEKAZYWANIE PRZEZ WARTOŚĆ

```
struct Foo { char foo[N]; };

Foo fun_value(Foo a, Foo b)
{
	Foo ret;
	for (int i = 0; i < SIZE; i++)
		ret.foo[i] = a.foo[i] + b.foo[i];
	return ret;
}
```

# PRZEKAZYWANIE PRZEZ WARTOŚĆ

```
struct Foo { char foo[N]; };

Foo fun_value(Foo a, Foo b)
{
        Foo ret;
        for (int i = 0; i < SIZE; i++)
                ret.foo[i] = a.foo[i] + b.foo[i];
        return ret;
}

Foo a = ..., b = ...;
Foo c = fun_value(a, b);
```

# PRZEKAZYWANIE PRZEZ WARTOŚĆ

```
struct Foo { char foo[N]; };

Foo fun_value(Foo a, Foo b)
{
        Foo ret;
        for (int i = 0; i < SIZE; i++)
                ret.foo[i] = a.foo[i] + b.foo[i];
        return ret;
}

Foo a = ..., b = ...;
Foo c = fun_value(a, b);

Foo a = ..., b = ..., c = ...;
Foo d = fun_value(fun_value(a, b), c);
```

# PRZEKAZYWANIE PRZEZ REFERENCJĘ

```
struct Foo { char foo[N]; };

void fun_ref(const Foo &a, const Foo &b, Foo & ret)
{
	for (int i = 0; i < SIZE; i++)
		ret.foo[i] = a.foo[i] + b.foo[i];
}
```

# PRZEKAZYWANIE PRZEZ REFERENCJĘ

```
struct Foo { char foo[N]; };

void fun_ref(const Foo &a, const Foo &b, Foo & ret)
{
        for (int i = 0; i < SIZE; i++)
                ret.foo[i] = a.foo[i] + b.foo[i];
}


Foo a = ..., b = ..., c;
fun_ref(a, b, c);
```

# PRZEKAZYWANIE PRZEZ REFERENCJĘ

```
struct Foo { char foo[N]; };

void fun_ref(const Foo &a, const Foo &b, Foo & ret)
{
        for (int i = 0; i < SIZE; i++)
                ret.foo[i] = a.foo[i] + b.foo[i];
}


Foo a = ..., b = ..., c;
fun_ref(a, b, c);

Foo a = ..., b = ..., c = ..., d;
Foo tmp;
fun_ref(a, b, tmp);
fun_ref(tmp, d, c);
```

# TRYB MIESZANY + OPERATOR

```
struct Foo { char foo[N]; };

Foo operator + (const Foo & a, const Foo &b)
{
	Foo ret;
	for (int i = 0; i < SIZE; i++)
		ret.foo[i] = a.foo[i] + b.foo[i];
	return ret;
}
```

# TRYB MIESZANY + OPERATOR

```
struct Foo { char foo[N]; };

Foo operator + (const Foo & a, const Foo &b)
{
        Foo ret;
        for (int i = 0; i < SIZE; i++)
                ret.foo[i] = a.foo[i] + b.foo[i];
        return ret;
}

Foo a = ..., b = ...;
Foo c = a + b;
```

# TRYB MIESZANY + OPERATOR

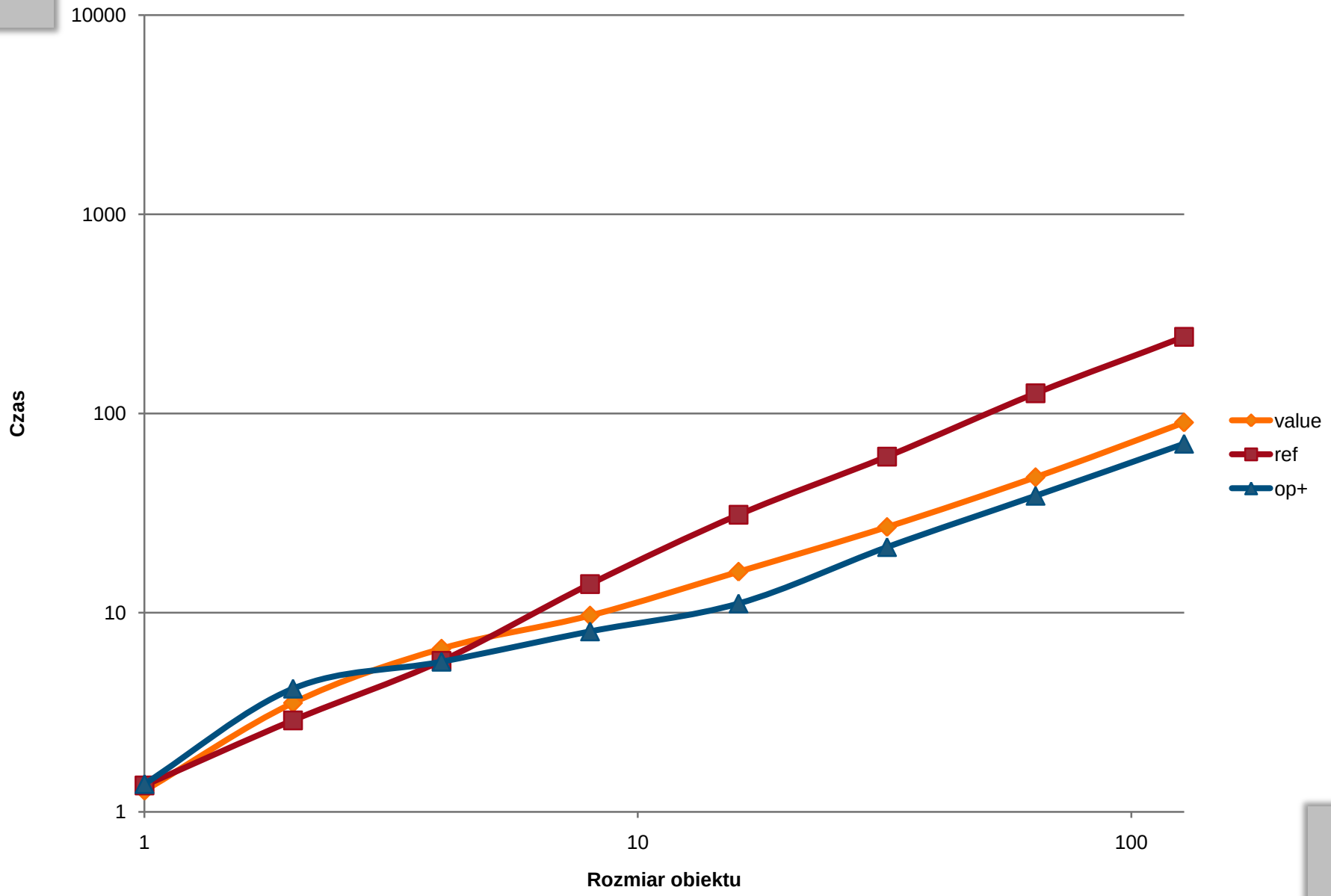
```
struct Foo { char foo[N]; };

Foo operator + (const Foo & a, const Foo &b)
{
        Foo ret;
        for (int i = 0; i < SIZE; i++)
                ret.foo[i] = a.foo[i] + b.foo[i];
        return ret;
}

Foo a = ..., b = ...;
Foo c = a + b;

Foo a = ..., b = ..., c = ...;
Foo d = a + b + c;
```

# WYNIKI



# VIRTUAL

- Dodatkowy narzut przy wywołaniu funkcji (pomijalny na X86, bolesny na konsolach)
- Brak możliwości **inline**
  - Teoretycznie inline możliwe przy instrumentalizacji (PGO), ale jest to bardzo wolny sposób, do tego wyłącznie dla buildów release
  - Często brak wstawienia danego fragmentu kodu powoduje lawinowe efekty: kompilator nie może dokonać obliczeń w czasie kompilacji i usunięcia zbędnego kodu
  - W jednym z naszych eksperymentów kompilator po usunięciu virtual wyciągnął **przed pętlę** skomplikowane obliczenia – efekt: czas wykonania mniejszy o kilkanaście rzędów wielkości
- Na poziomie koncepcyjnym: nie wiadomo jaki kod zostanie wywołany


# WZORCE PROJEKTOWE: SINGLETON

# SINGLETON

- Teoria
  - Tylko jedna instancja klasy
  - Dostępna globalnie
  - Inicjalizowana przy pierwszym użyciu

```
SoundSystem::GetInstance().PlaySound("Boom!");
```

- Wady
  - Brak jawnej kontroli nad kolejnością inicjalizacji i finalizacji
  - Narzut na każde wywołanie
  - Problem z bezpieczeństwem wątkowym
- **Czy na pewno system dźwiękowy powinien się inicjalizować w momencie, kiedy chcemy odegrać pierwszy dźwięk?**

# SINGLETON

- Rozwiązanie
  - Jawnie tworzyć i niszczyć obiekty globalne w określonym miejscu programu

```
g_SoundSystem = new SoundSystem();
```

*Every time you make a singleton, God kills a startup. Two if you think you've made it thread-safe.* – popularna sentencja z Twittera

# C++:
# KONSTRUKTOR

# KONSTRUKTOR

- Wada 1: Nie można stworzyć obiektu niezainicjowanego
  - Nie można ponownie użyć obiektu

```
class File {
public:
  File(const char *name) {
    buffer = new char[BUF_SIZE];
    file = fopen(name, "rb");
  }
  ~File() {
    fclose(file);
    delete [] buffer;
  }
private:
  char *buffer;
  FILE *file;
};
```

```
for (int i = 0; i < num_files; ++i) {
  File file(file_names[i]);
  file.Read(...);
}
```

- Bufor jest alokowany za każdym razem!

# KONSTRUKTOR

- Rozwiązanie: Osobna metoda inicjalizująca

```
class File {
public:
  File(const char *name)
    : buffer(new char[BUF_SIZE]) { }
  ~File() { delete [] buffer; }
  void open(const char *name) (
    file = fopen(name, "rb");
  }
  void close() { fclose(file); }
private:
  char *buffer;
  FILE *file;
};
```

```
File file;
for (int i = 0; i < num_files; ++i) {
  file.open(file_names[i]);
  file.Read(...);
  file.close();
}
```

- Ponowne użycie obiektu pozwala na zaalokowanie bufora tylko raz.

# KONSTRUKTOR

- Wada 2: Nie można zwrócić błędu

```
File::File(const char *name)
{
  file = fopen(name, "rb");
  if (file == NULL)
    return false; // Error!
}
```

```
File(const char *name)
{
  file = fopen(name, "rb");
  if (file == NULL)
    succeeded = false;
}

File file(file_name);
if (!file.get_succeeded())
  ...
```

- Trzeba rzucić wyjątek albo zapisać stan błędu w polach obiektu.

# KONSTRUKTOR

- Rozwiązanie: Osobna metoda inicjalizująca

```
bool File::open(const char *name) {
  file = fopen(name, "rb");
  return file != NULL;
}
```

```
File file;
if (!file.open(file_name))
  ...
```

- Zwykła metoda może zwrócić kod błędu.

# KONSTRUKTOR

- Wada 3: Dużo parametrów, nie można rozbić inicjalizacji na etapy

```
Edit *login_edit = new Edit(
  float2(16, 16), // position
  float2(128, 24), // size
  AUTO_ID, // id
  0, // flags
  COLOR_WHITE, // background color
  COLOR_BLACK, // text color
  "Login"); // watermark text
```

# KONSTRUKTOR

- Rozwiązanie: Osobne metody ustawiające parametry

```
Edit *login_edit = new Edit();
login_edit->set_position(float2(16, 16));
login_edit->set_size(float2(128, 24));
login_edit->set_watermark_text("Login");
```

- Parametry nieustawione po prostu przyjmują wartości domyślne.
- Kod jest bardziej opisowy – widzimy nazwy parametrów.

# KONSTRUKTOR

- Wada 4: Parametry trzeba przekazywać do klasy bazowej

```
Edit::Edit(const float2 &position, const float2 &size, uint id,
  uint flags, COLOR background_color, COLOR text_color,
  const char *watermark_text)
: Window(position, size, id, flags, background_color, text_color)
, watermark_text_(watermark_text)
{
  ...
}
```

- Nie można inicjalizować pól klasy bazowej na liście inicjalizacyjnej konstruktora klasy pochodnej.

# KONSTRUKTOR

- Rozwiązanie: Osobne metoda inicjalizująca lub metody ustawiające parametry

```
// Konstruktor Edit jest prosty
Edit *login_edit = new Edit();

// Metody klasy bazowej Window
login_edit->set_position(float2(16, 16));
login_edit->set_size(float2(128, 24));

// Metoda klasy Edit
login_edit->set_watermark_text("Login");
```

# PRZEDWCZESNA OPTYMALIZACJA

# PRZEDWCZESNA OPTYMALIZACJA

- *Premature optimization is the root of all evil* – Donald Knuth
  - Nie śmiem podważać słów Profesora
  - Jednak kiedy zdanie jest wyjęte z kontekstu, ucięte i opatrzone odpowiednim komentarzem, jego znaczenie zostaje wypaczone
  - Wielu rozumie je jako „nie warto przejmować się optymalizacją”
- Pełny cytat brzmi: ***We should forget about small efficiencies, say about 97% of the time. Premature optimization is the root of all evil***

[MatureOptimization]

# OPTYMALIZACJA

- Popularny stereotyp: Optymalizacja to przepisywanie procedur na assembler i zaciemnianie kodu.

- Tymczasem dobrze napisany program często jest równocześnie
  - bardziej **przejrzysty**
  - bardziej **wydajny**

# OPTYMALIZACJA

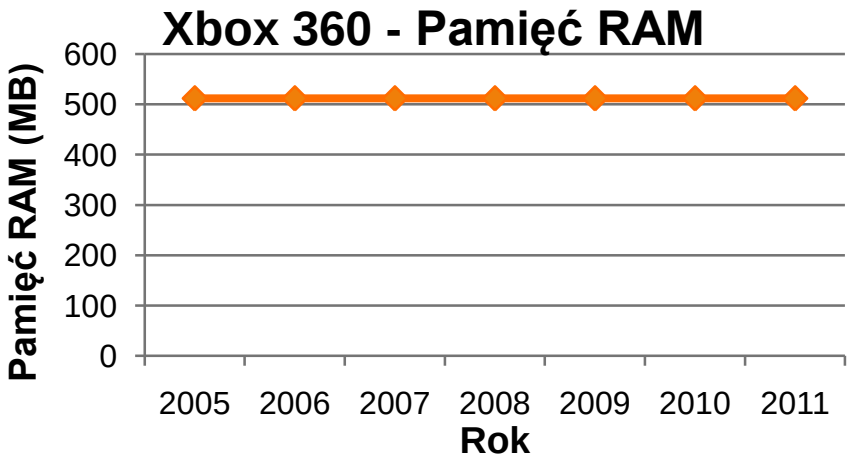
- Optymalizacja dotyczy każdego poziomu tworzonego programu
  - Niemożliwe jest poprawić wszystko w kodzie na sam koniec
  - Na wysokim poziomie ważny jest dobór struktur danych
  - Na niskim poziomie ważne są dobre nawyki (np. definiowane zmiennych poza pętlą)
  - Często nie ma jednego fragmentu kodu, który pochłania 90% czasu
- Rozwiązanie: ***Pisz w porządku od początku!***

# NIE WARTO OPTYMALIZOWAĆ?

- Argumenty
  - *Warto poświęcić wydajność programu dla wygody programowania*
  - *Szybszy sprzęt jest tańszy niż lepszy programista*
  - *Sprzęt staje się coraz szybszy*
- To może jest prawdą, kiedy piszesz program dla serwera, ale nie kiedy piszesz grę.

# NIE WARTO OPTYMALIZOWAĆ?

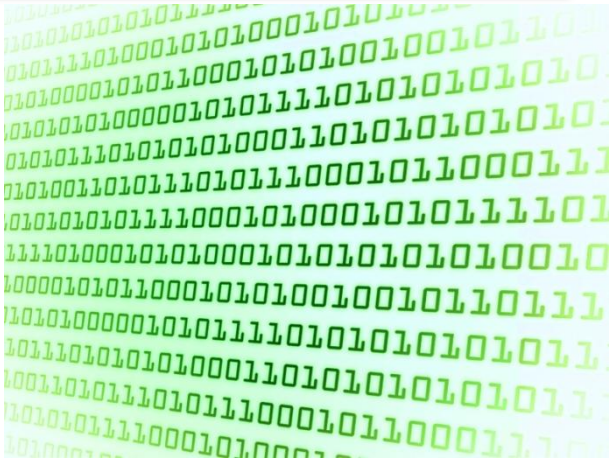
- Argumenty
  - *Warto poświęcić wydajność programu dla wygody programowania*
  - *Szybszy sprzęt jest tańszy niż lepszy programista*
  - *Sprzęt staje się coraz szybszy*
- To może jest prawdą, kiedy piszesz program dla serwera, ale nie kiedy piszesz grę.



# NIE WARTO OPTYMALIZOWAĆ?

- Konsola to sprzęt o ograniczonych możliwościach – określona wydajność, ilość pamięci itd.
  - Tak jak router, dekoder i inne systemy wbudowane
- PC to platforma ograniczona parametrami sprzętu, jaki aktualnie posiada przeciętny gracz
  - Na inne wymagania może sobie pozwolić Crysis 2, a na inne gra casual


# NIE WARTO OPTYMALIZOWAĆ?



- Konieczne jest bezkompromisowe wykorzystanie maksimum dostępnej mocy obliczeniowej, dlatego:
  - Używamy C++, a nie języków interpretowanych, z wirtualną maszyną czy garbage collectorem
  - Wyłączamy obsługę wyjątków i RTTI
  - Piszemy własne alokatory pamięci
  - Unikamy STL

# KOMPILATOR ZOPTYMALIZUJE?

- Często powtarza się, że kompilator jest lepszy w optymalizacji kodu od przeciętnego programisty
- Tymczasem…

# KOMPILATOR ZOPTYMALIZUJE?

- Często powtarza się, że kompilator jest lepszy w optymalizacji kodu od przeciętnego programisty
- Tymczasem…

```
// przeciętny programista
int y = x / x;

00401018 mov eax, 1
```

# KOMPILATOR ZOPTYMALIZUJE?

- Często powtarza się, że kompilator jest lepszy w optymalizacji kodu od przeciętnego programisty
- Tymczasem…

```
// MSVC 2008 Release (/O2)
int y = x / x;

00401018 mov ecx,dword ptr [esp+8]
0040101C mov eax,ecx
0040101E cdq
0040101F idiv eax,ecx
```

```
// przeciętny programista
int y = x / x;

00401018 mov eax, 1
```

# WIEMY WIĘCEJ NIŻ KOMPILATOR

- x / x w większości przypadków jest równe 1…
- …ale dla x = 0 zostanie wygenerowany wyjątek dzielenia przez 0 (dla liczb całkowitych) lub NaN (dla liczb zmiennoprzecinkowych)
- Ten przypadek możemy wykryć i obsłużyć wcześniej (na etapie inicjalizacji)
- Piszmy **wydajny** kod dla 99% przypadków, sytuacje **wyjątkowe** nie mogą rzutować na wydajność całego programu

# OPTYMALIZACJA KOMPILATORA

- Kompilator może nawet nie wyliczyć stałej, tylko wygenerować kod do jej obliczania wykonywany przed main()

```
1  const float CONST_1 = 3.f;
2  const float CONST_2 = 2.f * CONST_1;
3  const float CONST_3 = 1.f / CONST_2;
```

```
1  ??__ECONST_3@@YAXXZ PROC                    ; `dynamic initializer for 'CONST_3'', COMDAT
2  ; 251   : const float CONST_3 = 1.f / CONST_2;
3  movss     xmm0, DWORD PTR __real@3f800000
4  divss     xmm0, DWORD PTR _CONST_2
5  movss     DWORD PTR _CONST_3, xmm0
6  ret 0
7  ??__ECONST_3@@YAXXZ ENDP                    ; `dynamic initializer for 'CONST_3''
8  _CONST_3 DD 01H DUP (?)
9  ...
10 _CONST_1 DD 040400000r          ; 3
11 _CONST_2 DD 040c00000r          ; 6
```

- Rozwiązania:
  - CONST_3 = 1.f / (2.f * CONST_1)
  - #define zamiast const

[yarpen]

# WYNAJDYWANIE KOŁA NA NOWO?

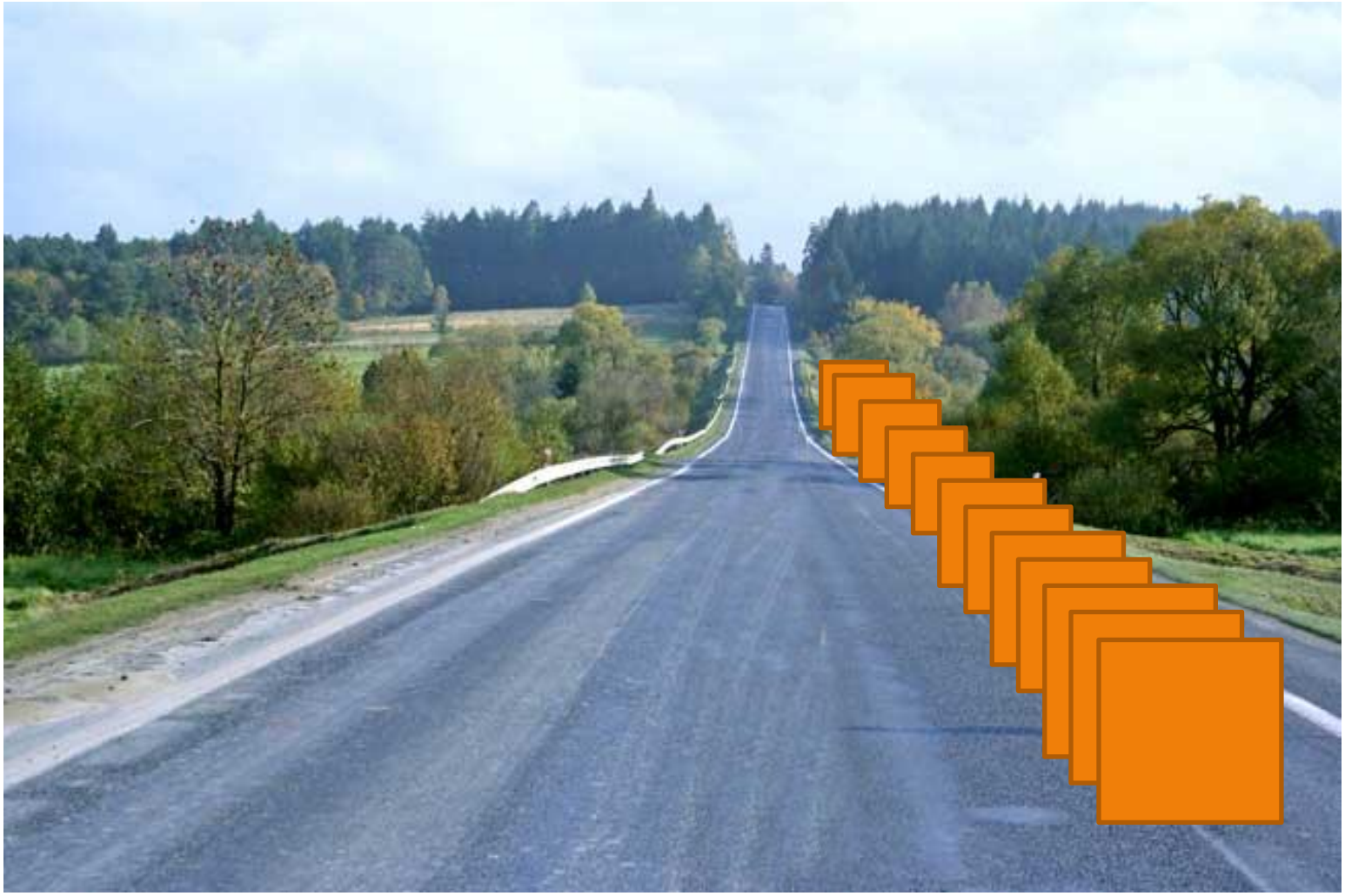
- *Nie pisz własnych bibliotek, tylko używaj gotowych. I tak nie napiszesz lepszych.*
  - Na pewno?
- Znasz swój szczególny przypadek lepiej, niż twórcy uniwersalnej biblioteki, np.:
  - Alokator pamięci ma stały rozmiar obiektu albo nie musi być bezpieczny wątkowo
  - Macierz zawsze będzie float4x4
- Możesz napisać kod wydajniejszy i lepiej dostosowany do swoich potrzeb
  - Jeśli nie, to przynajmniej dużo się przy tym nauczysz :)

# RÓWNOLEGŁOŚĆ

# LINIOWY KOD PRZETWARZAJĄCY DANE

```
int transform()
{
    for each (Object o)
    {
        ...
    }
}
```

# LINIOWY KOD PRZETWARZAJĄCY DANE

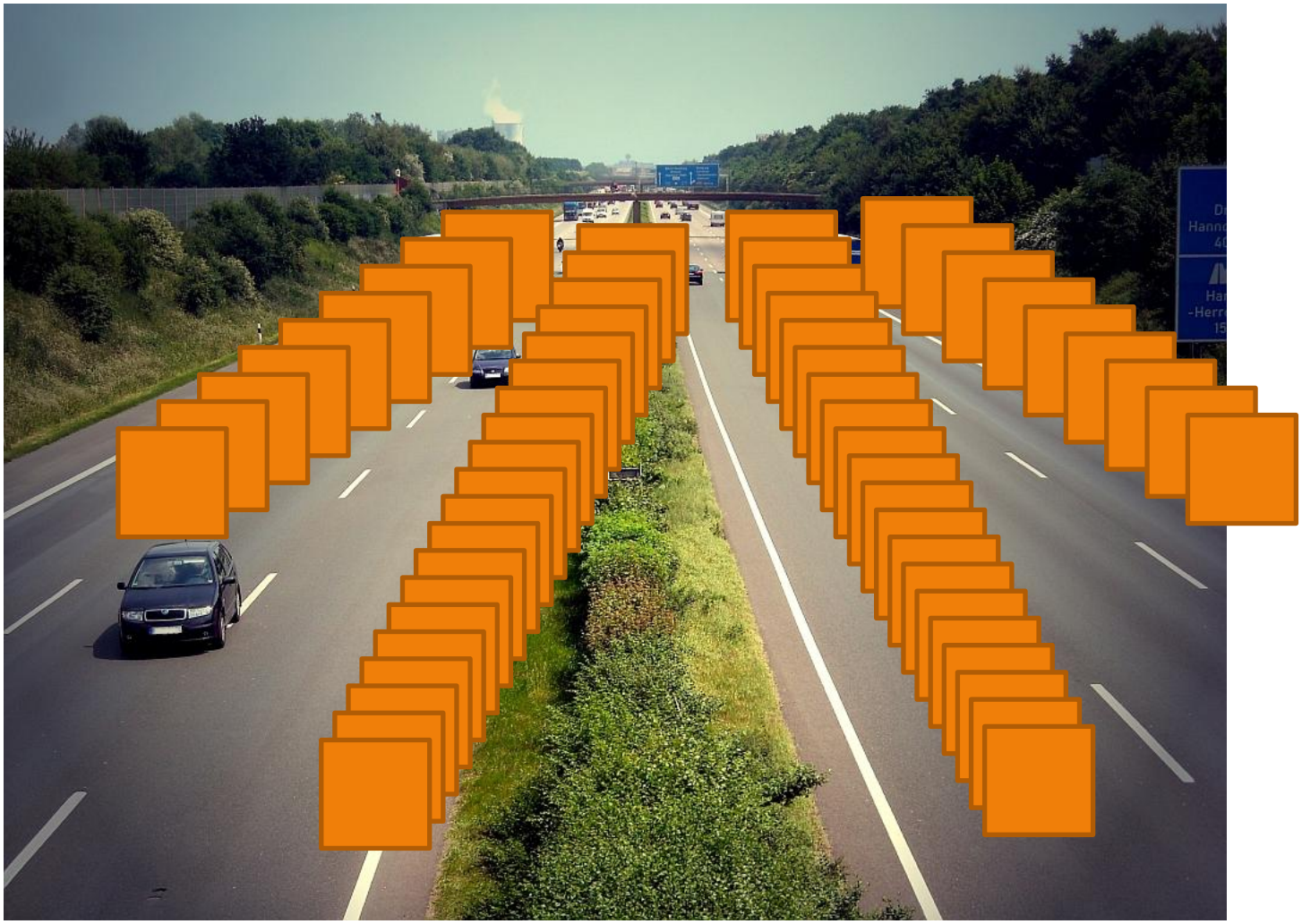


# LINIOWY KOD PRZETWARZAJĄCY DANE + RÓWNOLEGŁOŚĆ

- int transform()
- {
- #pragma omp parallel for
  - for (Object o)
  - {
  - ….
  - }

}

# LINIOWY KOD PRZETWARZAJĄCY DANE – RÓWNOLEGŁOŚĆ

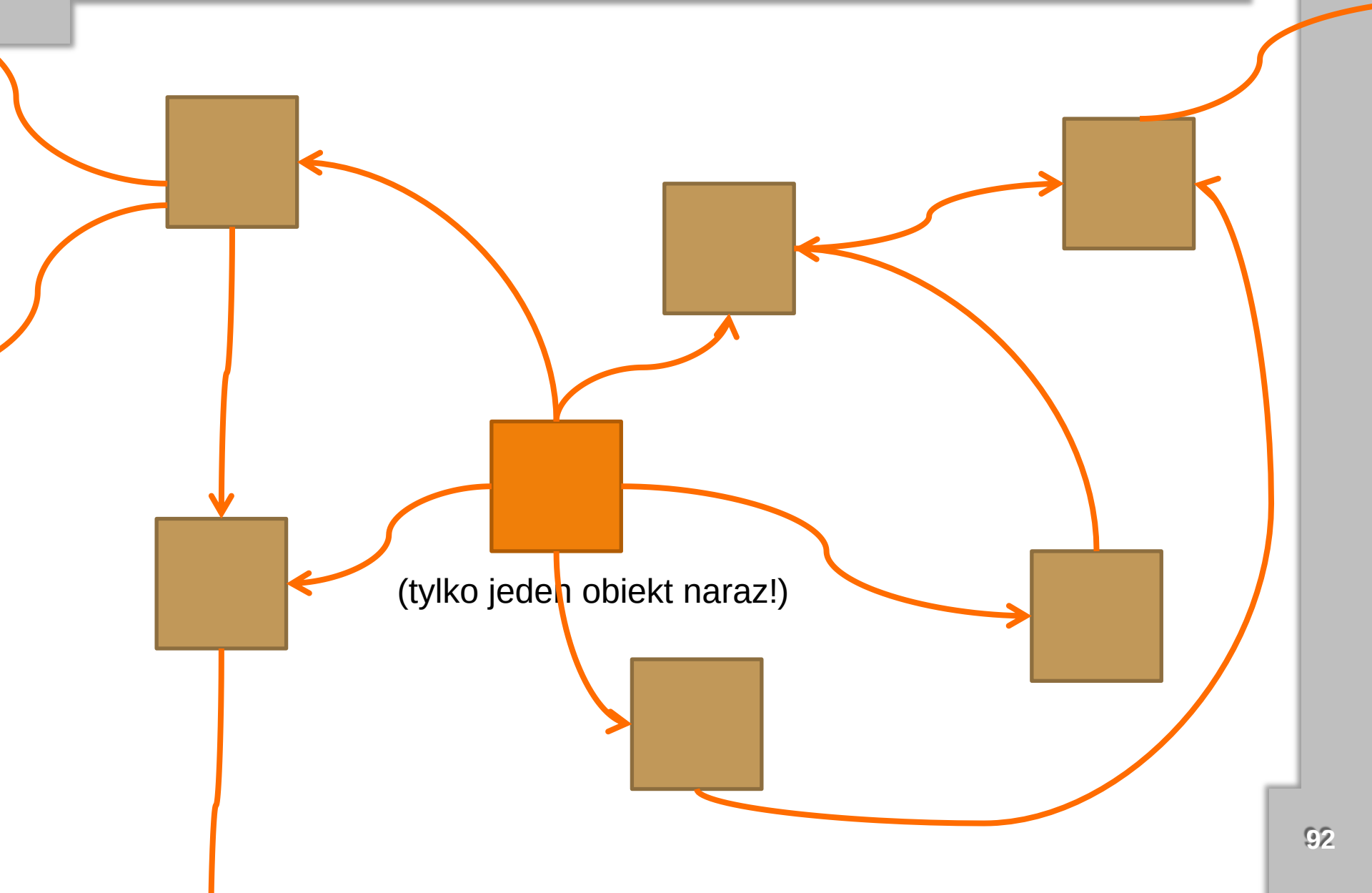


# KOD OBIEKTOWY

- *An OO program walks into a bar and says to the bartender, Alice asked Bob to ask Charlie to ask Dave to ask Eddie to ask Frank to ask…*

(sentencja z Twittera)

# KOD OBIEKTOWY



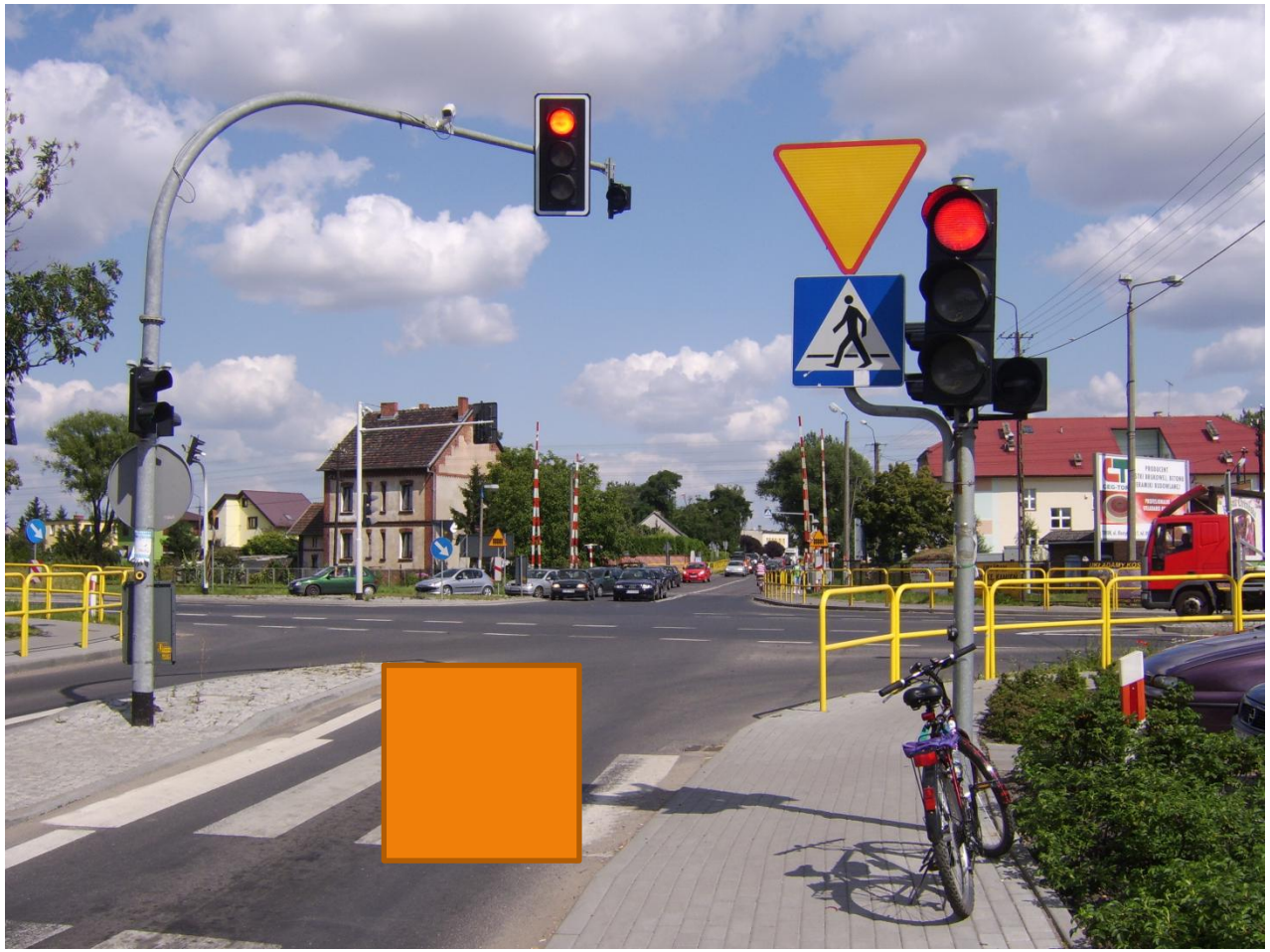
# PROGRAMOWANIE RÓWNOLEGŁE

- Muteksy to nie jest rozwiązanie.
- DOD jest dobre, jeśli chodzi o łatwość zrównoleglania.
- virtual void update() = 0;
  - Hmm, czy ta funkcja będzie thread-safe? Do czego ona się odwołuje?
- DOD – od razu widać co się dzieje
  - Zrównoleglić #pragmą OpenMP

# WYKONYWANIE WARUNKOWE

```
void foo(ptr * ob)
{
  if (ob->isBar())
  {
    ...
  }
  else
  {
    ...
  }
}
```

# WYKONYWANIE WARUNKOWE



# DODATKOWE ODWOŁANIA DO PAMIĘCI/OBIEKTÓW

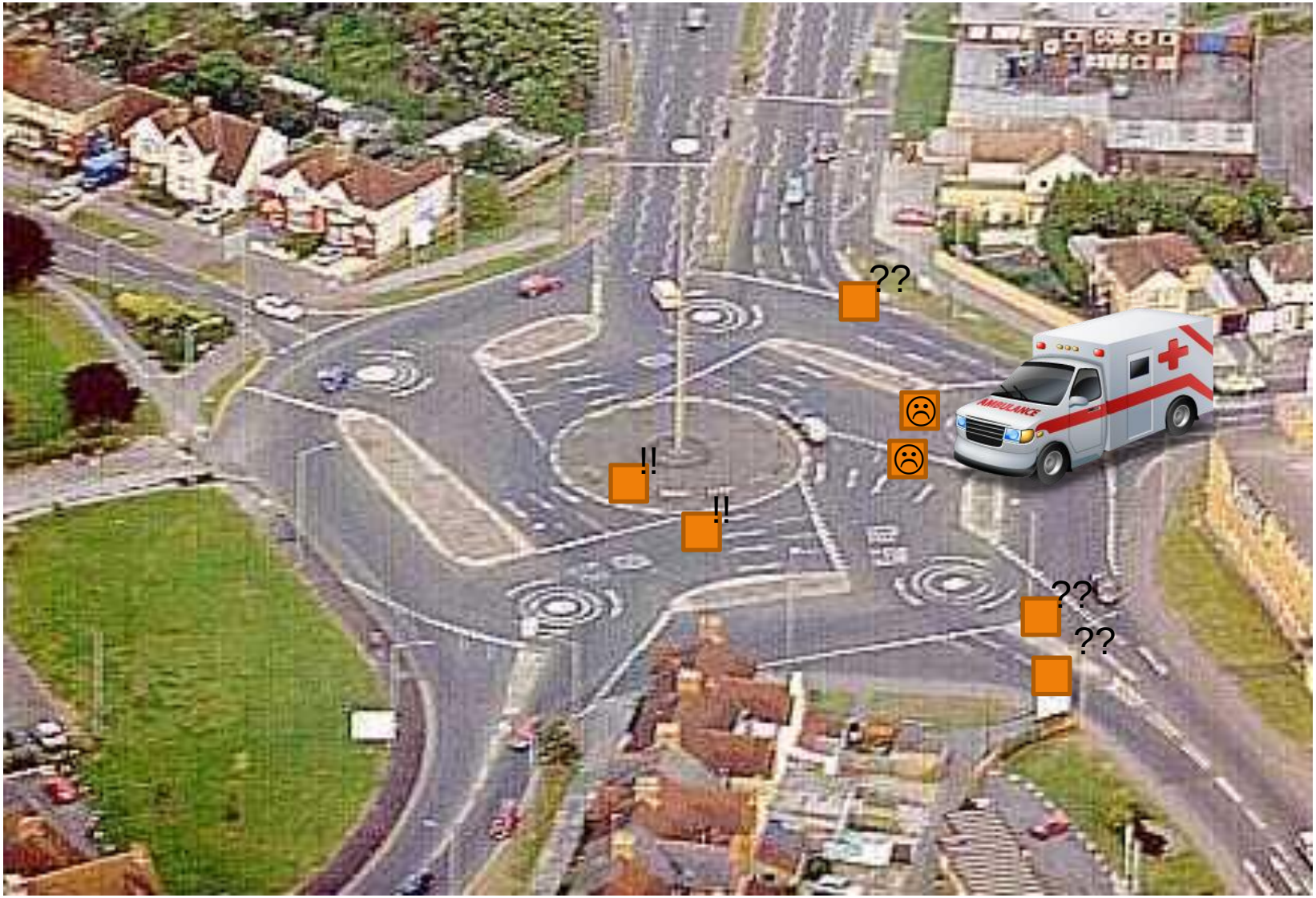
```
void processObject(ob)
{
  object2 a = ob->parent;
  object3 b = ob->son->brother->mother->sister;
  if (a->funVirtualFun(b, ob, a+b * (c-a))
  {
    ...
  }
}
```

# DODATKOWE ODWOŁANIA DO PAMIĘCI/OBIEKTÓW

# UŻYCIE WZORCÓW PROJEKTOWYCH, DODATKOWYCH WARSTW ABSTRAKCJI

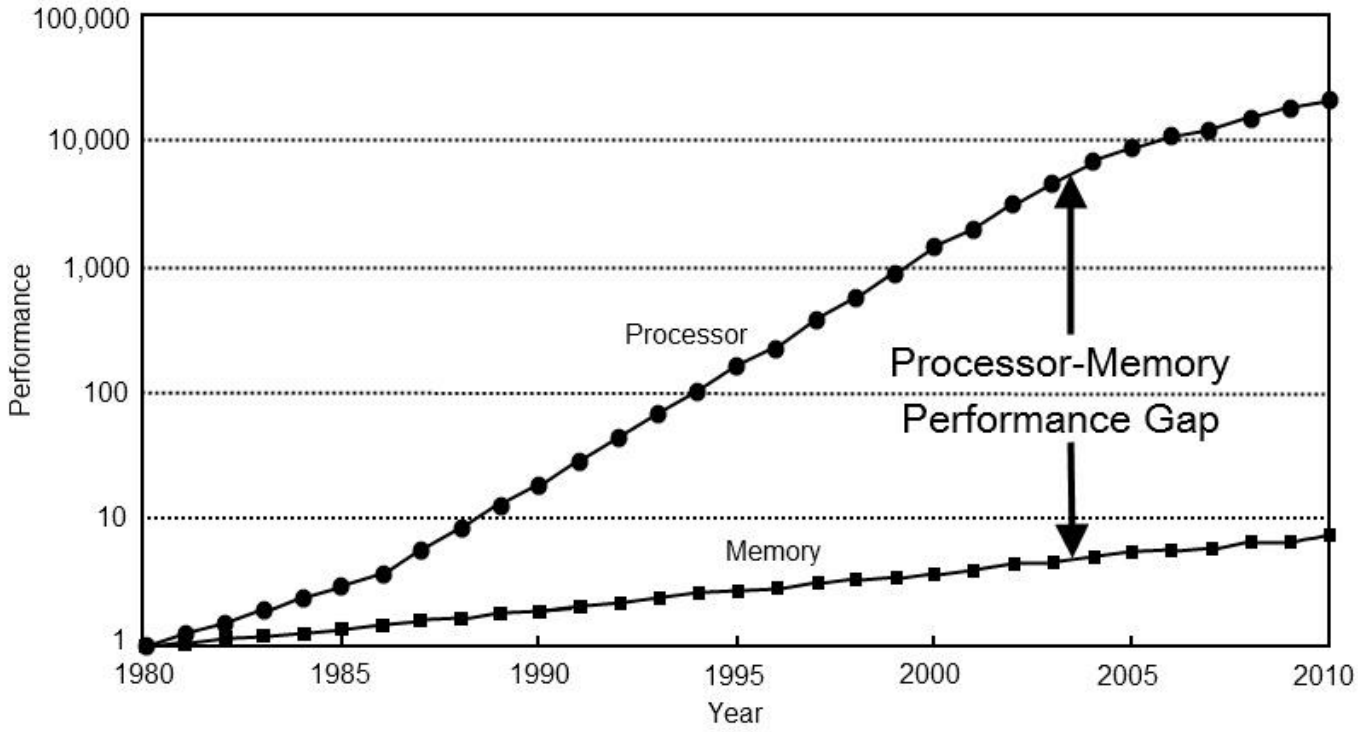
# IDEALNY OBIEKTOWY DESIGN

# IDEALNY OBIEKTOWY DESIGN + MT



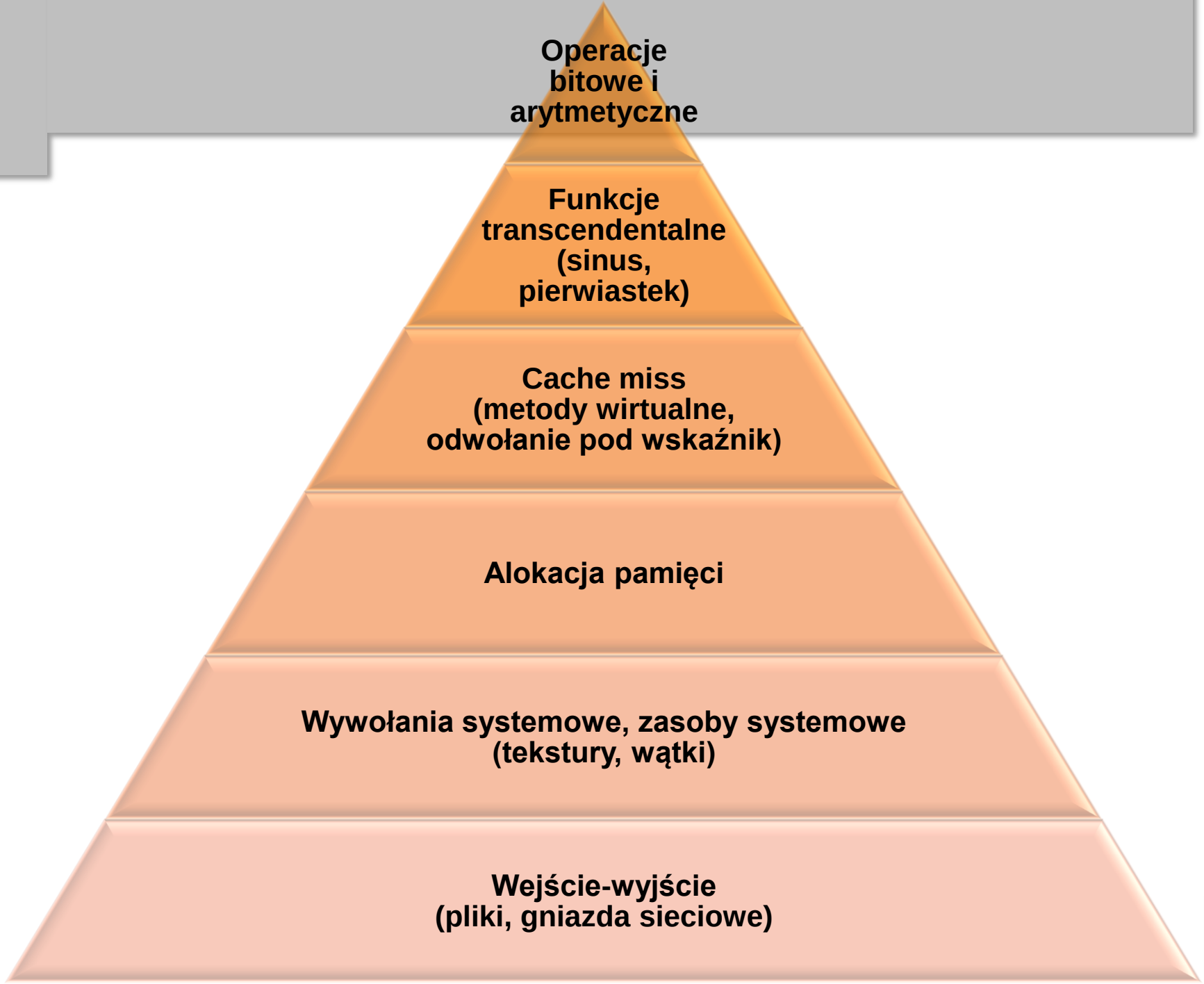
# WYDAJNOŚĆ

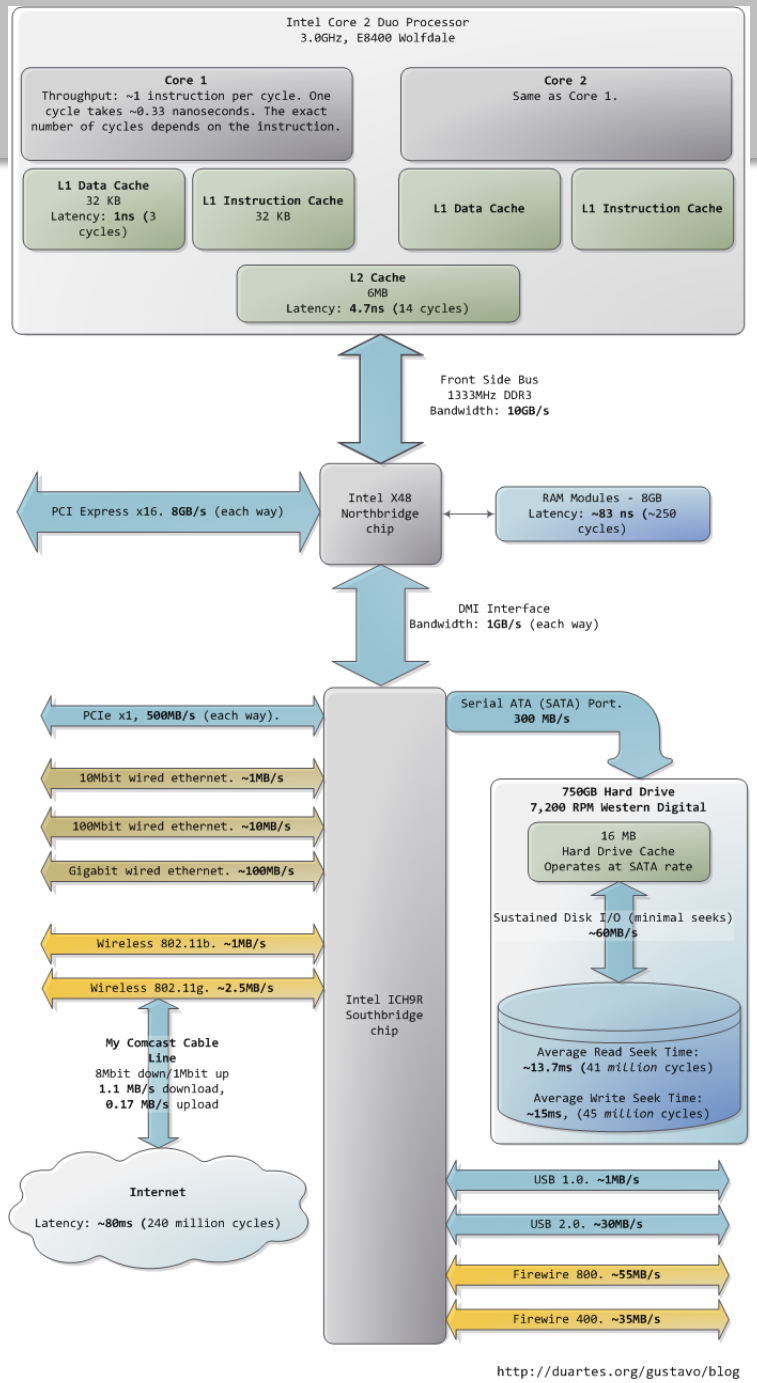
# PAMIĘĆ WOLNIEJSZA NIŻ PROCESOR



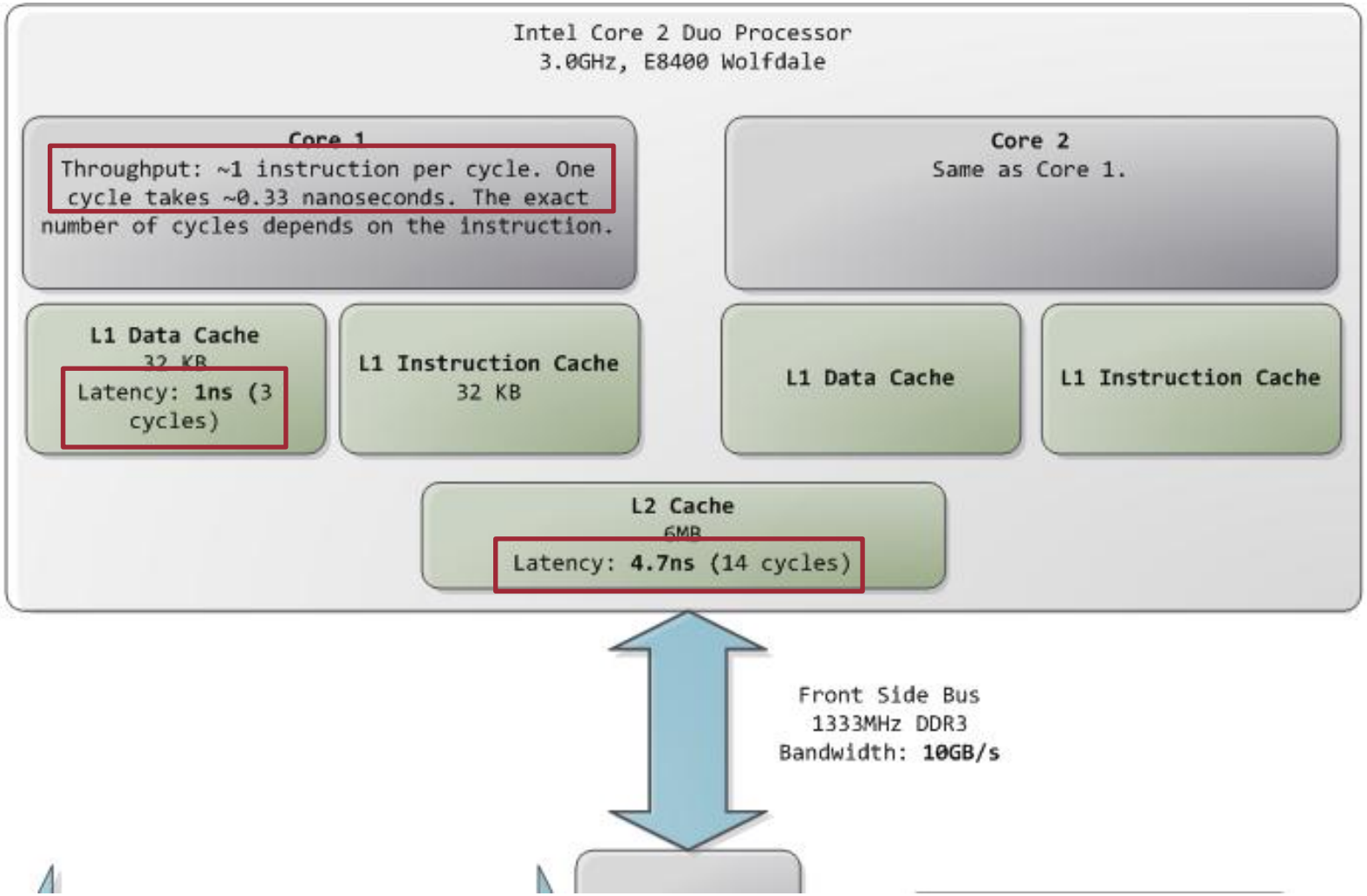

- Nietrafienie w pamięć podręczną – **cache miss**
  - 1980: Odwołanie do pamięci RAM ≈ 1 cykl procesora
  - 2009: Odwołanie do pamięci RAM ≥ **400** cykli procesora

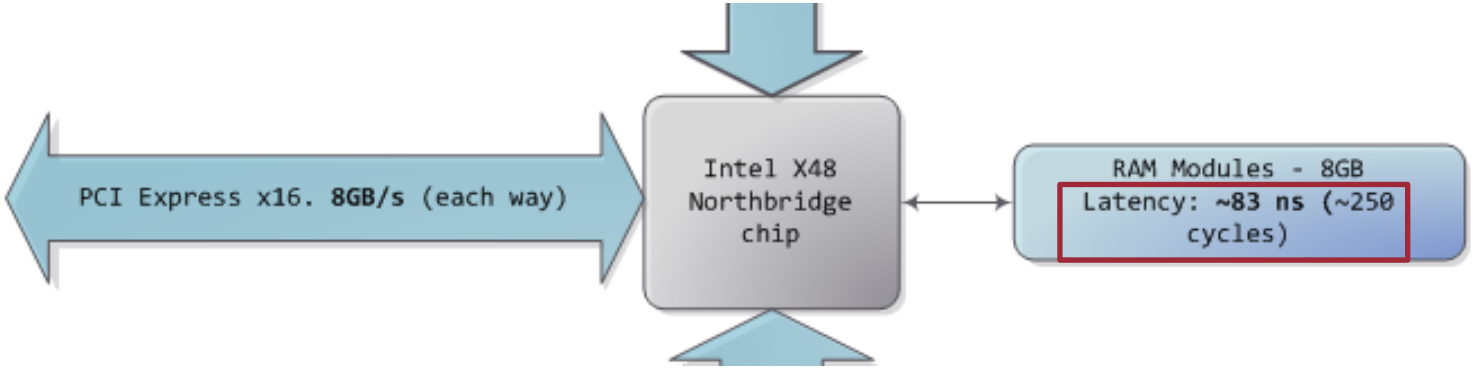
[Pitfalls]

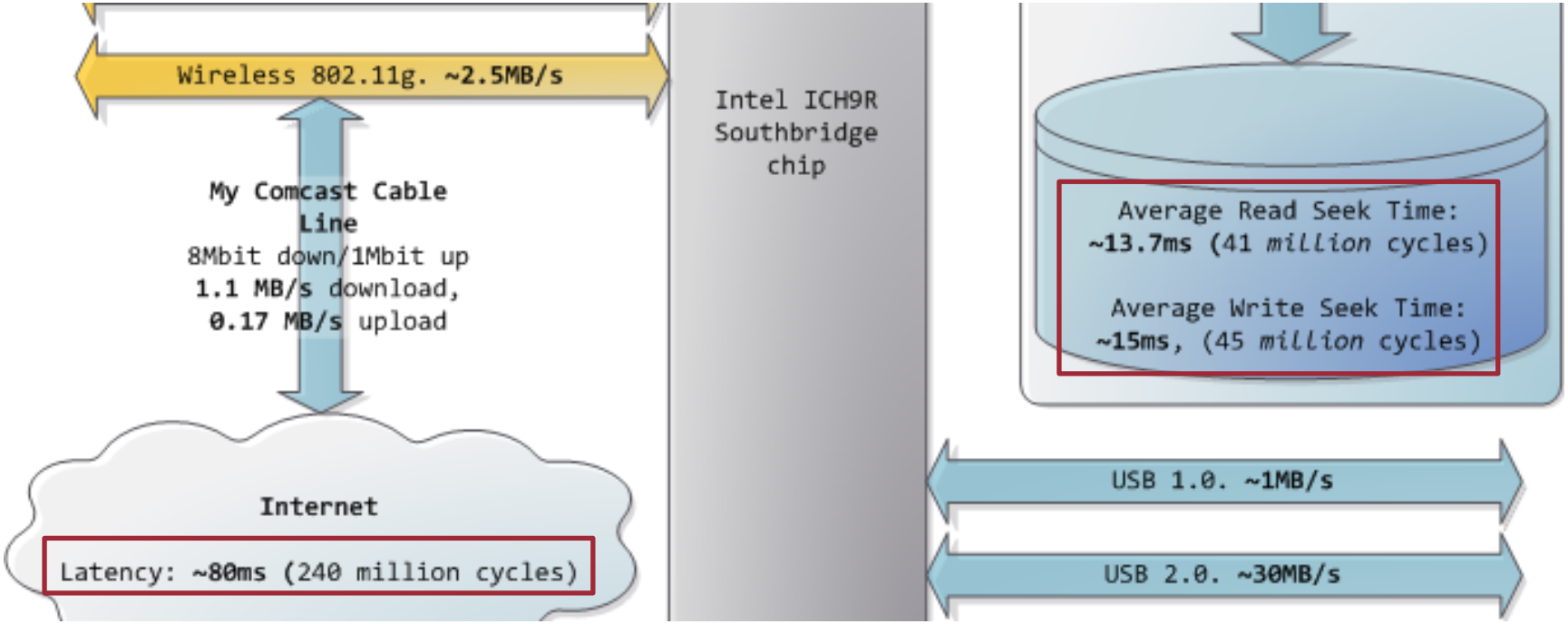

Operacje bitowe i arytmetyczne
Funkcje transcendentalne (sinus, pierwiastek)
Cache miss (metody wirtualne, odwołanie pod wskaźnik)
Alokacja pamięci
Wywołania systemowe, zasoby systemowe (tekstury, wątki)
Wejście-wyjście (pliki, gniazda sieciowe)




[GustavoDuarte]


Intel Core 2 Duo Processor
3.0GHz, E8400 Wolfdale
Core 1
Throughput: ~1 instruction per cycle. One cycle takes ~0.33 nanoseconds. The exact number of cycles depends on the instruction.
Core 2
Same as Core 1.
L1 Data Cache
32 KB
Latency: 1ns (3 cycles)
L1 Instruction Cache
32 KB
L1 Data Cache
L1 Instruction Cache
L2 Cache
6MB
Latency: 4.7ns (14 cycles)
Front Side Bus
1333MHz DDR3
Bandwidth: 10GB/s


PCI Express x16. 8GB/s (each way)
Intel X48 Northbridge chip
RAM Modules - 8GB
Latency: ~83 ns (~250 cycles)


Wireless 802.11g. ~2.5MB/s
My Comcast Cable Line
8Mbit down/1Mbit up
1.1 MB/s download,
0.17 MB/s upload
Internet
Latency: ~80ms (240 million cycles)
Intel ICH9R Southbridge chip
Average Read Seek Time:
~13.7ms (41 million cycles)
Average Write Seek Time:
~15ms, (45 million cycles)
USB 1.0. ~1MB/s
USB 2.0. ~30MB/s

# PRECALC

- ***Najszybszy kod to taki, który nigdy się nie wykonuje***
- **Precalc** – przygotuj dane wcześniej, nie licz za każdym razem
  - Podczas wczytywania gry NIE **parsuj** plików tekstowych, XML, modeli OBJ, tekstur PNG, nie **kompiluj** shaderów
  - Edytor lub inne narzędzia powinny przygotować assety w **docelowym** formacie: modele do wczytania **prosto** do VB/VBO, tekstury w DDS (ewentualnie JPEG), własne formaty plików, VFS
  - Podczas działania gry NIE licz **w każdej klatce** tego, co można wyliczyć **raz** lub **co jakiś czas** (np. AI)
- Stosuj culling i podział przestrzeni, aby nie przetwarzać tego, czego nie widać lub co nie jest istotne
- LOD – Level of Detail – kiedy mimo wszystko trzeba przetwarzać dużo danych, można mniej szczegółowo

# ALIASING

- Jednym z poważnych problemów wydajnościowych jest **aliasing**

```
struct Foo
{
  int q;
};
int bar(Foo &q, int *data)
{
  foo.q = 5;
  data[0] = 11;
  return foo.q; // == 5 ??
}
```

# ALIASING

- Jednym z poważnych problemów wydajnościowych jest **aliasing**

```
struct Foo
{
  int q;
};
int bar(Foo &q, int *data)
{
  foo.q = 5;
  data[0] = 11;
  return foo.q; // == 5 ??
}
```

Intuicja podpowiada że tak. W praktyce data może wskazywać na `foo.q`:

```
Foo foo;
int a = bar(foo, &foo.q);
```

# ALIASING

- Jednym z poważnych problemów wydajnościowych jest **aliasing**

```
struct Foo
{
  int q;
};
int bar(Foo &q, int *data)
{
  foo.q = 5;
  data[0] = 11;
  return foo.q; // == 5 ??
}
```

Intuicja podpowiada że tak. W praktyce data może wskazywać na `foo.q`:

```
Foo foo;
int a = bar(foo, &foo.q);
```

Rozwiązanie? `__restrict` – informacja dla kompilatora że dany wskaźnik jest **jedynym** odniesieniem do pewnych danych.

zmienne lokalne >>> zmienne składowe >>> zmienne globalne

# CODE BLOAT

- #1: preprocessing
  - 3 MB z wszystkimi nagłówkami STL
  - ??? z Boostem
- #2: pliki obiektowe
  - Konkretyzacja szablonów dla każdego typu…
  - …w każdej jednostce translacji

| | | | | |
|---|---|---|---|---|
| algorithm | complex | exception | list | stack |
| bitset | csetjmp | fstream | locale | stdexcept |
| cassert | csignal | functional | map | strstream |
| cctype | cstdarg | iomanip | memory | streambuf |
| cerrno | cstddef | ios | new | string |
| cfloat | cstdio | iosfwd | numeric | typeinfo |
| ciso646 | cstdlib | iostream | ostream | utility |
| climits | cstring | istream | queue | valarray |
| clocale | ctime | iterator | set | vector |
| cmath | deque | limits | sstream | |

# POLECANE OPCJE KOMPILATORA

- Wyłączyć **wyjątki**
  - Dodatkowy narzut nawet wtedy, kiedy nie potrzebujemy ich używać
- Wyłączyć **RTTI**
  - Typeinfo oraz dynamic_cast
- Najwyższy poziom **ostrzeżeń** (trzeba zwłaszcza uważać na konwersje typów: konwersja int/float jest wolna)
- Włączyć instrukcje **wektorowe** (SSE) oraz funkcje **intrincsic**
- Jeżeli nie ma przeciwwskazań, przestawić obliczenia zmiennoprzecinkowe na **fast**
- Włączyć najwyższy poziom **optymalizacji**, LTCG (ew. PGO) oraz WPO

# BIBLIOGRAFIA


- [MatureOptimization] Mature Optimization - Mick West, Game Developer Magazine, 2006
  - http://bit.ly/13Kkx0
- [GustavoDuarte] What Your Computer Does While You Wait, Gustavo Duarte
  - http://bit.ly/fB6r
- [Pitfalls] Pitfalls of Object Oriented Programming, Tony Albrecht (Sony Computer Entertainment Europe)
  - http://bit.ly/90fCdE
- [EvolveYourHierarchy] Evolve Your Hierarchy, Mick West
  - http://bit.ly/cah6k
- [yarpen] Attack of dynamic initializers, Maciej Siniło „yarpen”
  - http://bit.ly/fSqbLh
- [DivergentCoder] AoS & SoA Explorations
  - http://divergentcoder.com/programming/aos-soa-explorations-part-1/
  - http://divergentcoder.com/programming/aos-soa-explorations-part-2/
  - http://divergentcoder.com/programming/aos-soa-explorations-part-3/
- [C0DE517E] International movement for code rights
  - http://bit.ly/aHC56I
- [C++FQA] Yossi Kreinin
  - http://yosefk.com/c++fqa/
- [Duck] Managing Decoupling Part 3 - C++ Duck Typing, Niklas Frykholm
  - http://altdevblogaday.org/2011/02/25/managing-decoupling-part-3-c-duck-typing/
- Stanley B. Lippman *Model obiektu w C++*
- Dov Bulka, David Mayhew *Efektywne programowanie w C++* (Mikom 2001)

# ŹRÓDŁA WIEDZY (MAŁY OFF-TOPIC)

- Skąd czerpiemy wiedzę?
  - Strony WWW
    - Google
    - Konkretne adresy
  - **Twitter**
  - **Blogi** (np. altdevblogaday.org)
  - W języku angielskim!
- **Forum** to nie jest efektywne źródło wiedzy
  - Mały stosunek sygnału do szumu
  - Forum – czytasz wypowiedzi różnych osób
    Twitter i blogi – czytasz tylko tych, których chcecz

# PYTANIA?