

# (Średnio)zaawansowane programowanie w C++

Wykład 6 - wtyczki; łączenie C++ z C, Python, Rust; powtórzenie.

Robert Nowak

25L

# Plan wykładu

- ▶ translacja w C++,
- ▶ dynamiczna informacja o typie,
- ▶ wtyczki,
- ▶ tworzenie aplikacji z modułami w różnych językach programowania
  - ▶ łączenie C i C++,
  - ▶ łączenie C++ i Pythona,
  - ▶ łączenie Rust z C++, Rust z Pythonem.
- ▶ powtórzenie.

# *Translacja w C++*

- ▶ obróbka wstępna (preprocessing)
- ▶ kompilacja
- ▶ konsolidacja

# Preprocesor (unikać wykorzystania preprocesora)

Preprocesora należy używać jedynie do:

- ▶ dołączania plików nagłówkowych (`#include`)
- ▶ zabezp. nagłówków przed wielokrotnym dołączaniem
- ▶ kompilacji warunkowej

Makrodefinicje (**`#define`**)

- zamiast prostych stosować stałe
- zamiast złożonych szablony funkcji

```
const int MAX = 20; //stała globalna
class Foo {
    static const int SIZE = 30; //stała globalna, dostępna dla metod
};
template<typename T>
inline void max(const T& a, const T& b) {
    if(a > b) return a;
    else return b;
}
```

# Biblioteki

- ▶ biblioteki dostępne w formie źródeł
  - ▶ wykorzystują szablony
  - ▶ brak problemów z łączeniem (konsolidacją)
- ▶ biblioteki binarne
  - ▶ muszą być zgodne z aplikacją
  - ▶ kod może być ukryty

## Biblioteki binarne:

- ▶ statyczne (Windows: .lib, Linux: .a): - skompresowane wyniki kompilacji + tablica symboli
- ▶ dynamiczne (Windows: .dll, Linux: .so): - kod PIC (ang. *position independent code*) + tablica symboli
  - ▶ ładowane podczas startu aplikacji
  - ▶ ładowane podczas pracy aplikacji (np. wtyczki)

# Biblioteki ładowane dynamicznie

- ▶ możliwe wykorzystywanie typów, które nie są dostępne w czasie kompilacji
- ▶ interfejs musi być znany w czasie kompilacji

Biblioteka upraszczająca wykorzystanie wtyczek zawiera:

- ▶ zarządca wtyczek (rejestracja, inicjacja, finalizacja)
- ▶ wtyczka
- ▶ klasy pomocnicze (np. klasa reprezentująca bibliotekę dynamiczną)

Przykłady: QT (plugin), Boost.DLL, Boost.Extensions, Plugg, Gigi Sayfan framework (Dr Dobb's Journal).

# Przykład - zarządca wtyczek

```
class PluginManager { //fabryka obiektów
public:
    static PluginManager& getInstance(); //singleton
    bool loadAll(const std::string& directory); //ładowuje i inicjuje
    IPlugin* getObject(const std::string& id); //dostarcza uchwyt
};
```

Zadania:

- ▶ przechowuje uchwyty do dostępnych wtyczek
- ▶ pozwala ładować/zwalniać wtyczki
- ▶ w zależności od platformy wykorzystuje różne funkcje do ładowania bibliotek dynamicznych

# Reprezentacja wtyczki - bez użycia refleksji

```
class Selectable {  
public:  
    virtual bool isSelectable() = 0; //czy obiekt może być wyróżniany  
    virtual void select() = 0; //wyróżnia obiekt  
    virtual void unselect() = 0; //kasuje wyróżnienie obiektu  
};
```

**Wada:** typy, które nie dostarczają interfejsu muszą go implementować

```
class Background : public Selectable { //typ bez możliwości wyróżnienia  
public:  
    bool isSelectable() override { return false; } //nie można wybierać  
    void select() override {} //wymagana pusta implementacja  
    void unselect() override {} //wymagana pusta implementacja  
};
```



# Reprezentacja wtyczki - wykorzystanie refleksji

Mechanizmy refleksji : rzutowanie dynamiczne

- ▶ umożliwiają badanie obecności interfejsu
- ▶ upraszczają obsługę typów dostarczanych przez biblioteki dynamiczne

```
class Selectable {//interfejs dla typów dających możliwość wyróżniania
public:
    virtual void select() = 0;
    virtual void unselect() = 0;
};

class Button : public Selectable { //implementuje interfejs
public:
    void select() override { /* wyróżnienie obiektu */ }
    void unselect() override { /* kasowanie wyróżnienia */ }
};

class Background { //nie implementuje interfejsu
    //...
};
```

# Rzutowanie dynamiczne do badania interfejsu

```
Object* obj = /* inicjacja obiektu */ ;  
Selectable* sel = dynamic_cast<Selectable*>(obj);  
if(sel) //badanie, czy obiekt dostarcza interfejsu  
    sel->select(); //wyróżnienie obiektu
```

- ▶ klasy, które nie dostarczają interfejsu nie muszą go implementować
- ▶ badanie następuje za pomocą mechanizmów języka
- ▶ interfejs musi być znany
- ▶ typy mogą być implementowane wewnątrz bibliotek dynamicznych

# *Łączenie modułów C++ z modułami w innych językach programowania*

# wykład 1 - moduły w różnych językach programowania

- ▶ nie istnieje jeden, najlepszy język programowania
- ▶ stosowanie zawsze jednego języka programowania - niezbyt wyrafinowane rozwiązania

Warstwy pośrednie, aplikacje rozproszone:

- ▶ CORBA (Common Object Request Broker Architecture, OMG)
- ▶ RPC (Remote Procedure Call, Sun)
- ▶ .NET (Microsoft)

# Łączenie C i C++

Kompilatory muszą być zgodne (w taki sam sposób reprezentować typy wbudowane)

problemy:

- ▶ dekorowanie nazw dla linkera
- ▶ struktury danych
- ▶ funkcja main
- ▶ operacje na sterpie

# dekorowanie nazw

- linker
- ▶ w C nazwy nie mogą być przeciążane
  - ▶ w C++ mogą

nazwa funkcji jest dekorowana przez kompilator C++ (name mangling, name decoration)

```
extern "C" { //Zapobiega dekorowaniu nazw
    int funkcja(int a, int b) /* ... */
}
//Zapobiega dekorowaniu nazw jeżeli jest kompilowane przez C++
#ifdef __cplusplus
extern "C" {
#endif
    int funkcja( int a, int b) /* ... */
#ifdef __cplusplus
}
#endif
```

# funkcja main i struktury danych

Należy wybierać implementację main z C++, ponieważ

- ▶ zapewnia ona prawidłową inicjację składowych statycznych
- ▶ zapewnia wołanie destruktorów dla składowych statycznych

Struktury danych (tylko te, które są dostępne w C):

- ▶ typy wbudowane, POD
- ▶ struktury, które nie mają funkcji wirtualnych
- ▶ obiekty, które zostały powołane przez `new` powinny być zwalniane przez `delete`
- ▶ pamięć alokowana przez `malloc` powinna być zwalniana przez `free`

# Python

- ▶ interpretowany, interaktywny język programowania
- ▶ Python Software Foundation, [www.python.org](http://www.python.org)
- ▶ programowanie **funkcyjne**, obiektowe i strukturalne
- ▶ dynamiczna kontrola typów
- ▶ brak enkapsulacji
- ▶ zarządzanie pamięcią przez garbage collection
- ▶ dokumentacja w kodzie źródłowym
- ▶ zmienna liczba argumentów funkcji i metod
- ▶ zaznaczanie bloków przez wcięcia



# Potrzeba użycia różnych języków w aplikacjach

(patrz wykład 1)

System komputerowy zawsze:

- ▶ ma ograniczenia czasowe, więc tworzenie całości powinno być możliwie szybkie
- ▶ posiada pewne elementy, które są „wąskim gardłem” - powinny być zaimplementowane wydajnie (20% kodu)

System komputerowy często:

- ▶ posiada pewne elementy, których autor nie chce udostępniać (kod ukryty przed użytkownikiem)
- ▶ posiada pewne fragmenty, które powinny być dostępne dla użytkownika (aby dostosować aplikację do indywidualnych potrzeb, np. konfiguracja)

# Łączenie C++ i Pythona

- ▶ powody rozszerzania Pythona w C++
  - ▶ kod wykonywany jest szybciej
  - ▶ wykorzystanie istniejącego kodu
  - ▶ kod jest ukryty przed użytkownikiem
- ▶ powody osadzania Pythona w C++
  - ▶ brak potrzeby kompilacji
  - ▶ wykorzystanie bibliotek Pythona
  - ▶ kod dostępny dla użytkownika

Interfejsy:

- ▶ Python C API
- ▶ Boost Python

# Przykłady

Rozszerzanie Pythona w C++:

- ▶ utworzenie biblioteki dzielonej zawierającej funkcje w C++
- ▶ import biblioteki do Pythona i wykonanie

Przykład: hello world

Osadzanie Pythona w C++ (Boost Python):

Przykład: embedding

# Łączenie Rust i C++

Wołanie kodu C++ wewnątrz Rust, kod jest 'unsafe':

- ▶ `cty` - typy zgodne z C i C++
  - `c_char`, `c_uchar`
  - `c_short`, `c_ushort`, `c_int`, ..., `c_ulongong`
  - `int8_t`, `uint8_t`, ..., `uint64_t`
  - `c_float`, `c_double`

```
extern "C" { //plik foo.h
    int funkcja(int a, int b) /* ... */
}
//plik foo.rs
#[repr(C)]
extern "C" {
    pub fn funkcja( a: cty::c_int, b: cty::c_int) -> cty::c_int;
}
```

Następnie kompilujemy kod w C/C++ (np. do `.a` lub `.lib`).

# Łączenie Python i Rust

Najczęściej rozszerzanie Pythona w Rust (tworzenie kodu w Rust, który jest wołany w Python) ponieważ:

- ▶ wydajność (kod kompilowany),
- ▶ współbieżność.

Dlaczego Rust, nie C++?

- ▶ gwarancja poprawnego zarządzania pamięcią,
- ▶ gwarancja braku wyścigów.

Narzędzia:

- ▶ `PyO3` - biblioteka Rust, pozwala tworzyć kod wołany z Pythona,
- ▶ `Maturin` - pakiet Pythona (w `pip`) upraszczający używanie modułów w Rust tworzonych przez `PyO3`

# *Powtórzenie*

# Powtórzenie

- ▶ polimorfizm, funkcje wirtualne, dziedziczenie wielobazowe
- ▶ dynamiczna informacja o typie
- ▶ różne strategie obsługi błędów, mechanizm wyjątków
- ▶ sprytnie wskaźniki:
  - ▶ `std::unique_ptr`,
  - ▶ `std::shared_ptr`,
  - ▶ `std::weak_ptr`,
  - ▶ `boost::intrusive_ptr`
- ▶ referencja do r-wartości w C++11 (r-value reference)
- ▶ Rust: podstawy, zarządzanie zasobami

# *Przykładowe zadania*



## Zadanie (Guru of The Week, 2), niepotrzebne obiekty

```
string FindAddr( list<Employee> l, string name ) {  
    for( list<Employee>::iterator i = l.begin(); i != l.end(); i++ )  
        if( *i == name )  
            return (*i).addr;  
    return "";  
}
```

//poprawne: algorytmy i lambda z C++11

```
auto it=find_if(l.begin(),l.end(),[&](Employee& e){return e.name==name;});
```

## Zadanie (Guru of The Week, 2), niepotrzebne obiekty

```
string FindAddr( list<Employee> l, string name ) {  
    for( list<Employee>::iterator i = l.begin(); i != l.end(); i++ )  
        if( *i == name )  
            return (*i).addr;  
    return "";  
}  
  
string FindAddr( const list<Employee>& l, const string& name ) {  
    string addr;  
    for( list<Employee>::iterator i = l.begin(); i != l.end(); ++i )  
        if( i->name() == name ){  
            addr = (*i).addr; break; }  
    return addr;  
}  
  
//poprawne: algorytmy i lambda z C++11  
auto it=find_if(l.begin(),l.end(),[&](Employee& e){return e.name==name;});
```

# Zadanie, brak zwalniania obiektów

```
class List {  
public:  
    using PNode = shared_ptr<Node>;  
    struct Node {  
        Node(const string& s):s_(s){}  
        string s_;  
        PNode prev_;  
        PNode next_;  
    };  
    void push_back(const string& s);  
private:  
    PNode head_;  
};
```

```
void List::push_back(const string& s)  
    if( head_ ) { //not empty  
        PNode tail( new Node(s) );  
        tail->prev_ = head_->prev_;  
        tail->next_ = head_;  
        head_->prev_->next_ = tail;  
        head_->prev_ = tail;  
    }  
    else { //empty  
        head_ = PNode(new Node(s));  
        head_->prev_ = head_;  
        head_->next_ = head_;  
    }  
}
```

# Zadanie, brak zwalniania obiektów - rozwiązanie

```
class List {  
public:  
    using PNode = shared_ptr<Node>;  
    using PWNode = weak_ptr<Node>;  
    struct Node {  
        Node(const string& s):s_(s){}  
        string s_;  
        PWNode prev_; //zmiana  
        PNode next_;  
    };  
    void ~List() { //zmiana  
        //usuniecie zal. cyklicznej  
        if( head_ )  
            head_->prev_->next_.reset();  
    }  
    void push_back(const string& s);  
private:  
    PNode head_;  
};
```

```
void List::push_back(const string& s)  
    if( head_ ) { //not empty  
        PNode tail( new Node(s) );  
        tail->prev_ = head_->prev_;  
        tail->next_ = head_;  
        head_->prev_.lock()->next_ = tail;  
        head_->prev_ = tail;  
    }  
    else { //empty  
        head_ = PNode(new Node(s));  
        head_->prev_ = head_;  
        head_->next_ = head_;  
    }  
}
```

*Dziękuję*