

Politechnika Świętokrzyska

Wydział Elektrotechniki, Automatyki i Informatyki

Programowanie Imperatywne, Obiektowe i Deklaratywne – Projekt

Temat: Programowanie obiektowe w języku C

Autor: Michał Wójcik

Grupa: 1ID23A

1. Wprowadzenie

Język C++ został wprowadzony jako rozszerzenie języka C o możliwość programowania obiektowego (oraz inne nowoczesne udogodnienia, np *smart pointers*). Nie oznacza to jednak, że przed rozpowszechnieniem tego języka nie dało się programować obiektowo w C. Programiści potrzebujący korzystać z tego paradygmatu albo pisali własne narzędzia albo korzystali z bibliotek. Jedną z takich bibliotek jest **GObject** (skrót od **Glib Object System**), podstawa środowiska GNOME dla systemu GNU/Linux i wielu innych bibliotek z rodziny GNOME.

Celem tego projektu jest zbadanie możliwości programowania obiektowego w języku C z użyciem biblioteki GObject.

2. Implementacja z użyciem GObject

Deklaracja klasy w GObject jest dość skomplikowana. Rozbija się ją na pliki nagłówkowe [.h] przechowujące deklaracje oraz pliki z kodem właściwym [.c] zawierające definicje elementów klasy. Aby zadeklarować klasę należy użyć makra `G_DECLARE_FINAL_TYPE` lub `G_DECLARE_DERIVABLE_TYPE`, któremu przekazuje się kolejno:

- Nazwę klasy w formacie `NamespaceClassname`
- Nazwę klasy w formacie `namespace_classname`
- Przestrzeń nazw w formacie `NAMESPACE`
- Nazwę klasy w formacie `CLASSNAME`
- Nazwę klasy, po której klasa ma dziedziczyć w formacie `NamespaceClassname`.

Wywołanie tego makra powinno zostać poprzedzone deklaracją typu. Na przykład, dla klasy finalnej `GobjectCat`, deklaracja wygląda następująco:

```
1. #define GOBJECT_TYPE_CAT (gobject_cat_get_type())
2. G_DECLARE_FINAL_TYPE(GobjectCat, gobject_cat, GOBJECT, CAT, GObject)
```

Następnie deklarowane są widoczne publicznie metody klasy.

```
1. GObjectCat* gobject_cat_new(void);
2. void gobject_cat_meow(GobjectCat *self);
```

W pliku .c następuje definicja wszystkiego co zostało zadeklarowane przez makra. Aby wszystko działało jak trzeba, należy stosować się do konwencji nazewnictwa używanej przez GObject.

```
1. #include "GobjectCat.h"
2.
3. /* Destructor declaration */
4. static void gobject_cat_dispose (GObject *object);
5. static void gobject_cat_finalize (GObject *object);
6.
7. /* Class Definition */
8. struct _GobjectCat
9. {
10.     GObject parent_instance;
11.     gchar* name;
12. };
13.
14. G_DEFINE_TYPE(GobjectCat, gobject_cat, G_TYPE_OBJECT);
```

Na początku definiowana jest struktura klasy wraz z polami. Pola publiczne w GObject podążają za konwencją tzw. Własności (ang. *Properties*), których użycie jest definiowane w metodach ustawiających (*setter*) oraz pobierających wartości (*getter*).

```
1.  /* Properties */
2.  enum
3.  {
4.      PROP_NAME = 1,
5.      N_PROPERTIES
6.  };
7.  static GParamSpec *properties[N_PROPERTIES];
8.
9.  /* Property setter/getter definitions */
10.
11. static void
12. gobject_cat_set_property (GObject *object,
13.                          guint propertyId,
14.                          const GValue *value,
15.                          GParamSpec *paramSpec)
16. {
17.     GObjectCat *self = GOBJECT_CAT(object);
18.
19.     switch (propertyId) {
20.         case PROP_NAME:
21.             self->name = g_value_dup_string(value);
22.             g_object_notify_by_pspec(G_OBJECT(self), properties[PROP_NAME]);
23.             break;
24.
25.         default:
26.             G_OBJECT_WARN_INVALID_PROPERTY_ID(object, propertyId, paramSpec);
27.             break;
28.     }
29. }
30.
31. static void
32. gobject_cat_get_property (GObject *object,
33.                          guint propertyId,
34.                          GValue *value,
35.                          GParamSpec *paramSpec)
36. {
37.     GObjectCat *self = GOBJECT_CAT(object);
38.
39.     switch (propertyId) {
40.         case PROP_NAME:
41.             g_value_set_string(value, self->name);
42.             break;
43.
44.         default:
45.             G_OBJECT_WARN_INVALID_PROPERTY_ID(object, propertyId, paramSpec);
46.             break;
47.     }
48. }
```

Specyficzne dla GObject jest zastosowanie inicjalizatora klasy. Opisuje on jak powinna zostać utworzona klasa, m.in. tutaj „instalowane” są parametry, przypisywane są gettery, settery i destruktor. Inicjalizator uruchamia się tylko za pierwszym razem, gdy klasa jest tworzona w pamięci.

```
1.  static void
2.  gobject_cat_class_init (GObjectClass *klass)
3.  {
4.      GObjectClass *objectClass = G_OBJECT_CLASS(klass);
5.      GParamFlags default_flags = static_cast<GParamFlags>(G_PARAM_READWRITE |
6.      G_PARAM_CONSTRUCT | G_PARAM_STATIC_STRINGS | G_PARAM_EXPLICIT_NOTIFY);
7.
8.      objectClass->dispose = gobject_cat_dispose;
9.      objectClass->finalize = gobject_cat_finalize;
```

```

9.     objectClass->set_property = gobject_cat_set_property;
10.    objectClass->get_property = gobject_cat_get_property;
11.
12.    properties[PROP_NAME] = g_param_spec_string("name",
13.                                                "Cat's name",
14.                                                "A name for the cat",
15.                                                "Mr. Speckles",
16.                                                default_flags);
17.    g_object_class_install_properties(objectClass, N_PROPERTIES, properties);
18. }
19.

```

GObject wymaga też podania destruktorów. Destrukcja obiektu podzielona jest na dwie części: **dispose** odpowiada za usuwanie głęboko zagnieżdżonych obiektów związanych z biblioteką, natomiast **finalize** za czyszczenie pamięci przydzielonej w składowych, np stringów.

```

1.  static void
2.  gobject_cat_dispose(GObject *object)
3.  {
4.      G_OBJECT_CLASS(gobject_cat_parent_class)->dispose(object);
5.  }
6.
7.  static void
8.  gobject_cat_finalize(GObject *object)
9.  {
10.     GObjectCat *self = static_cast<GObjectCat
11.     *>(gobject_cat_get_instance_private(GOBJECT_CAT(object)));
12.     g_free ((gpointer) self->name);
13.     G_OBJECT_CLASS(gobject_cat_parent_class)->finalize(object);
14. }

```

Po wykonaniu tych wszystkich „przymusowych” kroków, można przejść do implementacji zadeklarowanych publicznych metod. Należy zwrócić uwagę na budowę konstruktora – nie przyjmuje on żadnych parametrów. Po utworzeniu obiektu, powinno się zmieniać właściwości ręcznie.

```

1.  /* Constructor Definition */
2.  GObjectCat *
3.  gobject_cat_new (void){
4.      return (GObjectCat*)(g_object_new (GOBJECT_TYPE_CAT, NULL));
5.  }

```

Pozostałe metody to po prostu funkcje, działające na obiekcie klasy. Poprzez zawarcie ich w tym samym pliku źródłowym, co struktura z polami, a nie w pliku nagłówkowym, mają one dostęp do tych pól, w odróżnieniu do pozostałych fragmentów kodu chcących korzystać z biblioteki.

```

1.  void
2.  gobject_cat_meow(GObjectCat *self) {
3.      //self->name only accessible here, thus: private!
4.      printf("Meow, I am %s\n", self->name);
5.  }

```

3. Wykorzystanie zaimplementowanej klasy

Jeśli chodzi o używanie przygotowanych klas, jest to o wiele prostsze niż ich tworzenie i bardzo przypomina korzystanie z klas w języku C++. Jedyną różnicę stanowi brak pól publicznych. Jest to jednak w pewnym stopniu wymuszenie dobrej praktyki, jaką jest enkapsulacja danych i dostęp wyłącznie przez gettery i settery.

Stworzenie nowego obiektu (instancji klasy) wygląda następująco:

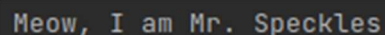
```
1. GObjectCat *gobjectCat = gobject_cat_new();
```

Wygląda więc to prawie identycznie do odpowiednika z C++, gdy tworzymy obiekt ze słowem kluczowym **new**.

```
1. CppCat *cppCat = new CppCat();
```

Wywołanie metod publicznych różni się tym, że nie są one składowymi klasy, lecz przyjmują w parametrze obiekt, na którym będą operować. Można by je w ten sposób przyrównać do metod zaprzyjaźnionych z języka C++.

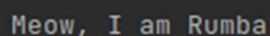
```
1. gobject_cat_meow(gobjectCat);
```



Rys.1. Wynik wywołania metody gobject_cat_meow() dla utworzonego obiektu.

Ustawienie właściwości odbywa się za pomocą funkcji **g_object_set**, przyjmującej kolejno wskaźnik do obiektu, nazwę właściwości i wartość do ustawienia.

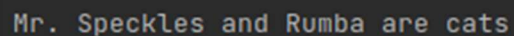
```
1. //cat with a different name
2. GObjectCat *gobjectCat1 = gobject_cat_new();
3. g_object_set(gobjectCat1,
4.             "name", "Rumba",
5.             NULL);
6. gobject_cat_meow(gobjectCat1);
```



Rys.2. Wynik wywołania metody gobject_cat_meow() dla obiektu po zmianie właściwości.

Podobnie jest w przypadku gettera, zdobycie wartości właściwości polega na wywołaniu funkcji **g_object_get**, przyjmującej wskaźnik do obiektu, nazwę właściwości i wskaźnik do zmiennej, do której zapisana zostanie wartość.

```
1.  gchar *catName, *cat1Name;
2.  g_object_get(gobjectCat,
3.              "name", &catName,
4.              NULL);
5.  g_object_get(gobjectCat1,
6.              "name", &cat1Name,
7.              NULL);
8.  printf("%s and %s are cats\n", catName, cat1Name);
9.  g_free(catName);
10. g_free(cat1Name);
```



Mr. Speckles and Rumba are cats

Rys.3. Wynik wykonania powyższego kodu.

4. Porównanie z kodem w C++

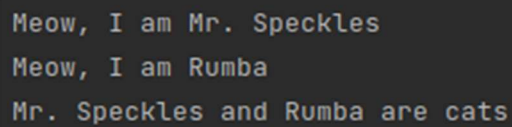
Wszystko co zostało opisane to próba stworzenia jednej klasy finalnej z jednym parametrem i jedną metodą publiczną, operującą na obiekcie. Aby uzyskać ten sam efekt w języku C++, wystarczy zaledwie kilka linii kodu:

```
1.  ///CppCat.h
2.  #ifndef PROGRAMOWANIE_CPPCAT_H
3.  #define PROGRAMOWANIE_CPPCAT_H
4.
5.  #include <string>
6.
7.  class CppCat {
8.  private:
9.      std::string name;
10. public:
11.     CppCat();
12.     void Meow();
13.     std::string getName();
14.     void setName(std::string name);
15. };
16.
17. #endif //PROGRAMOWANIE_CPPCAT_H
```

```
1.  /// CppCat.cpp
2.  #include "CppCat.h"
3.
4.  CppCat::CppCat() {
5.      name = "Mr. Speckles";
6.  }
7.
8.  void CppCat::Meow() {
9.      printf("Meow, I am %s\n", name.c_str());
10. }
11.
12. std::string CppCat::getName() {
13.     return name;
14. }
15.
16. void CppCat::setName(std::string name) {
17.     this->name = name;
18. }
```

Wykorzystanie klasy, dzięki zastosowaniu czytelnych getterów i setterów jest również prostsze i nie wymaga pamiętania nazw właściwości ani przydzielania pamięci na zmienne. Nie jest też konieczne korzystanie z dynamicznego przydzielania pamięci na same obiekty – można tworzyć je statycznie.

```
1. CppCat cppCat = CppCat();
2. cppCat.meow();
3.
4. CppCat cppCat1 = CppCat();
5. cppCat1.setName("Rumba");
6. cppCat1.meow();
7.
8. printf("%s and %s are cats\n",
9.        cppCat.getName().c_str(),
10.       cppCat1.getName().c_str());
```



```
Meow, I am Mr. Speckles
Meow, I am Rumba
Mr. Speckles and Rumba are cats
```

Rys.4. Wynik wykonania powyższego kodu.

5. Wnioski

Biblioteka GObject pozwala na użycie obiektowego paradygmatu programowania w języku C. Tworzenie klas jest czasochłonne, a wiedza którą trzeba osiąść przed rozpoczęciem korzystania z biblioteki może przytłoczyć. Większość kodu wymaganego do stworzenia klasy jest powtarzalna i zajmuje wiele linii w pliku źródłowym, co skutecznie zmniejsza czytelność kodu dla niewprawionego oka i utrudnia pisanie własnej logiki. Samo korzystanie z przygotowanej klasy nie jest o wiele trudniejsze niż w przypadku języków przeznaczonych do programowania obiektowego, jednak nadal rządzi się specyficznymi zasadami, których należy przestrzegać. Mimo wszystkich niedogodności, biblioteka dostarcza potężnego narzędzia i nowoczesnych mechanizmów dla niskopoziomowego języka C.