|  |
| --- |
| Wstęp teoretyczny |
| Wydział: Elektrotechniki, Automatyki i Informatyki  Kierunek: Informatyka  Projekt: Technologie obiektowe |
| Michał Olejko Studia stacjonarne  Stopień II, Semestr I |

Tematem projektu jest emulacja programowania obiektowego w języku, który sam z siebie obiektowości nie wspiera. Do projektu wykorzystany zostanie strukturalny język programowania – C. Istnieje już rozwiązanie, które jako rozszerzenie tego języka podobne do *Smalltalk* wprowadza model obiektowy – Objective-C stworzone przez Toma Love i Brada Cox, oraz drugie rozszerzenie języka stworzone dzięki Bjarne Stroustrup’owi znane jako język C++. Pierwsze z nich stosowane było głównie w frameworku Cocoa w systemie OS X oraz iOS, za to C++ dzięki możliwości stosowania zarówno programowania proceduralnego jak i obiektowego w połączeniu z wysoką wydajnością kodu wynikowego zdobył pozycję jednego z najpopularniejszych języków programowania ogólnego przeznaczenia. Jak podaje wikipedia – „Na początku XXI wieku liczbę programistów języka C++ szacowano na około 3 miliony”.

Imitację obiektowości można stworzyć w języku C na kilka sposobów, jednakże prawdopodobnie wszystkie z nich opierają się na strukturach i wskaźnikach. Dla przykładu w strukturze bez problemu możemy zapisać typy proste jak int, float czy char, ale zapisanie funkcji nie jest już dozwolone. Problem można najprościej rozwiązać poprzez stworzenie wskaźnika w strukturze, ciało funkcji umieścić poza nią, a następnie przypisać do tego wskaźnika odpowiednią funkcję. W ten sposób traktując strukturę jako klasę można stworzyć na tej bazie obiekt, z którego możliwe jest odwoływanie się do zmiennych oraz funkcji. Można również wywoływać funkcje przekazując im wskaźnik do odpowiedniego ‘obiektu’, wtedy za pomocą operacji ‘->’ można odwoływać się do odpowiednich pól. Wtedy wywoływanie metod wyglądałoby następująco:

metoda(wskaźnik\_do\_obiektu, argumenty);

oraz dla porównania w C++:

objekt->metoda(argumenty);

W ten sposób można wywoływać funkcje działające na konkretnym obiekcie.

Innym, lecz podobny sposób opisany jest w książce A.-T. Schreiner, "Object - oriented Programming with ANSI - C”, 1993, gdzie tworzona jest struktura o nazwie *Class,*  a następnie tworząc strukturę, która ma być imitacją klasy jako pierwsze pole zapisuje się wskaźnik typu void\*. Metodą nazwaną *new()* alokuje się pamięć na obiekt, następnie sprawdza się to za pomocą funkcji *assert(wskaźnik na obiekt)*, następnie jawnie konwertuje się wskaźnik, który traktuje początek obiektu jako wskaźnik do struktury *Class* i ustawia się mu wskaźnik przekazany jako parametr funkcji (wskaźnik na tworzony obiekt). Następnie jeśli klasa posiada konstruktor wywołuje się go i zwraca jako wynik funkcji:

void \* new (const void \* \_class, ...)

{

const struct Class \* class = \_class;

void \* p = calloc(1, class —> size);

assert(p);

\* (const struct Class \*\*) p = class;

if (class —> ctor)

{

va\_list ap;

va\_start(ap, \_class);

p = class —> ctor(p, & ap);

va\_end(ap);

}

return p;

}

Oraz:

struct Class {

size\_t size;

void \* (\* ctor) (void \* self, va\_list \* app); //konstruktor

void \* (\* dtor) (void \* self); //destruktor

void \* (\* clone) (const void \* self); //klonowanie obiektow

int (\* differ) (const void \* self, const void \* b); //porownywanie obiektow

};

struct String {

const void \* class; /\* must be first \*/

char \* text;

};

Funkcja usuwająca obiekt również przyjmuje wskaźnik na niego i najpierw tworzy wskaźnik na wskaźnik typu struct Class i przypisuje przekazany mu adres. Jeśli przekazany obiekt posiada destruktor jest on wywoływany, a następnie zwalniana jest pamięć za pomocą funkcji *free*:

void delete (void \* self)

{

const struct Class \*\* cp = self;

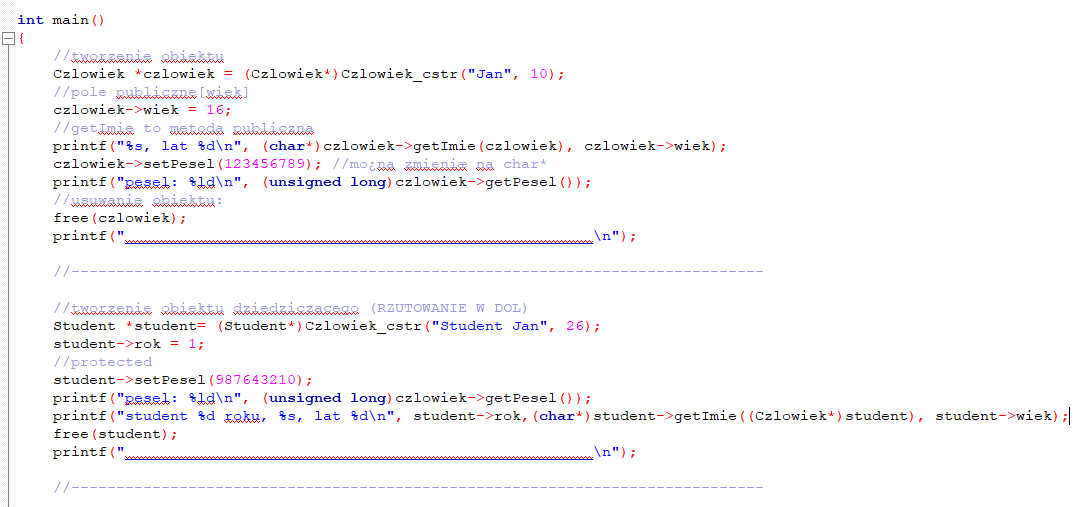
if (self && \* cp && (\* cp) —> dtor)

self = (\* cp) —> dtor(self);

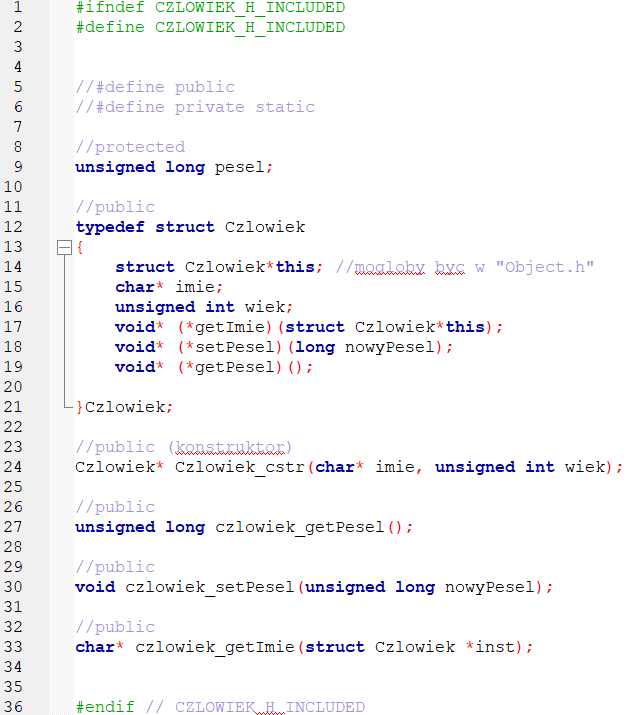
free(self);

}

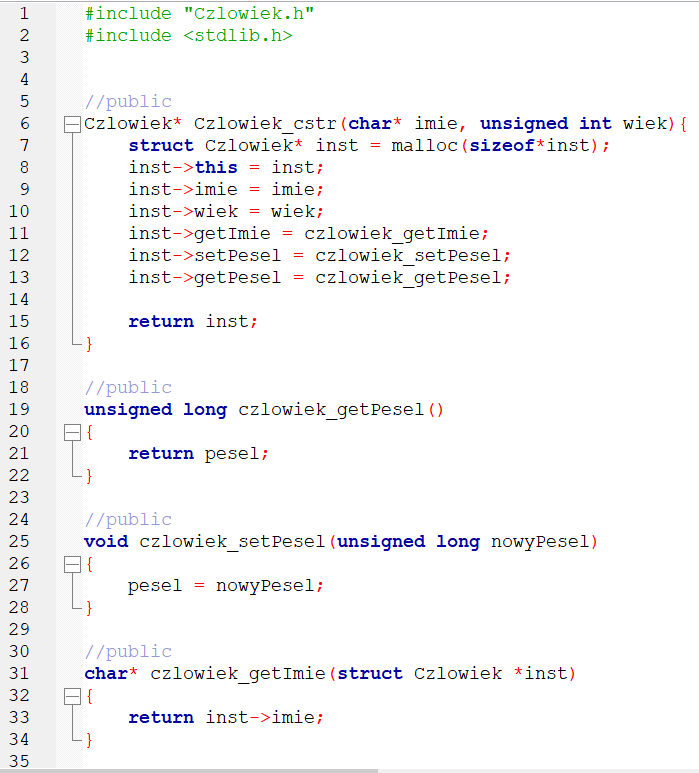
Rozwiązanie to jest bliższe realnemu tworzeniu obiektów, ale potrzeba ręcznie tworzyć konstruktor, destruktor, metodę do porównywania obiektów czy ich kopiowania na nowy obiekt.  
  
Bazując na powyższej teorii stworzono obiekty typu *Człowiek* oraz *Student*:



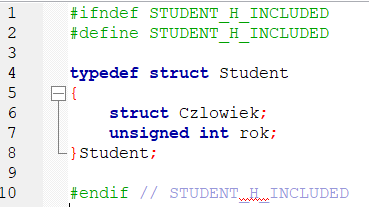
*Człowiek.h:*

**

Oraz *Człowiek.c:*

**

*Student.h:*

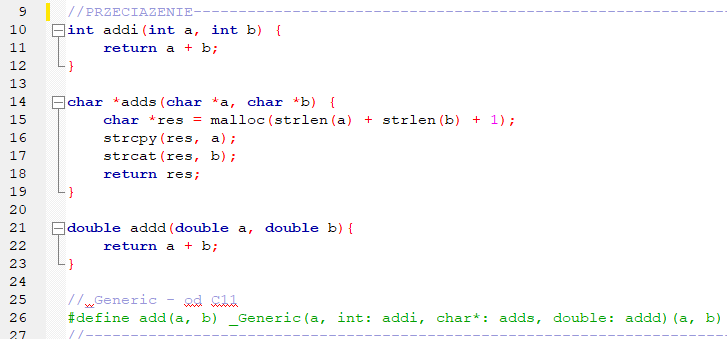
**

*Student.c:*

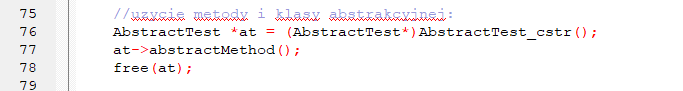
**

Jak widać *Student* rozszerza *Czlowiek* dodając polę rok, ale zachowując wszystkie pola Czlowiek oraz korzysta z jego konstruktora – rzutowanie w dół. W klasie *Czlowiek*, w pliku .h inicjalizujemy wszystkie zmienne, a do metod należało użyć wskaźników, a następnie już w pliku .c, w konstruktorze przypisać odpowiednie funkcje do tych wskaźników. W ten sposób później po stworzeniu obiektu konstruktorem, który przypisał pamięć dla obiektu możliwe jest wykonywanie metod zawartych wewnątrz obiektów oraz działanie na ich polach.

Przeciążenie można wykonać poprzez otagowane unie lub od C11 po prostu użyć słówka kluczowego \_Generic, które działa w ten sam sposób, ale wymaga o wiele mniej kody. Użycie \_Generic wygląda w następujący sposób:



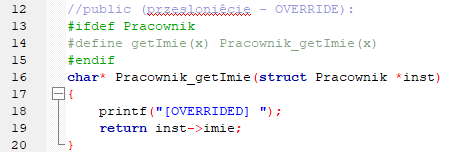
Przypisuje się tu dla wywoływania metody *add(a,b)* odpowiednie metody w zależności od typów parametrów – w tym wypadku tylko od parametru pierwszego:



Wynik:

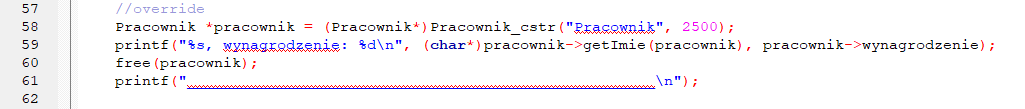


Przesłonięcie w projekcie zostało zaprezentowane na klasie typu *Pracownik*, która rozszerza klasę *Czlowiek* o pole *unsigned int wynagrodzenie*, a samo przesłonięcie występuje przy metodzie getImie. W konstruktorze pracownika przypisujemy metodzie *getImie()* metodę *Pracownik\_getImie()* oraz korzystamy z konstrukcji #ifdef - #define - #endif:



Metoda działa w ten sam sposób, lecz wyświetla napis [OVERRIDED] tak, aby pokazać że to ona została wywołana, a nie metoda z klasy wyżej – *Czlowiek*.

Przedstawienie tego rozwiązania można znaleźć w liniach 58-60 funkcji main:



Działanie:



Aby osiągnąć dostęp publiczny do metod lub pól należy je umieścić w strukturach klas, w plikach nagłówkowych .h. Dla chronionych należy umieścić je w tym samym pliku, ale już poza strukturą. Zostało to przedstawione na przykładzie *Czlowiek-Student*. Jeżeli chcemy kompletnie ukryć pola przed innymi klasami należy zmienne umieścić w pliku wykonywalnym .c. Ponieważ dziedzicząc dodaje się plik nagłówkowy .h to nie będzie możliwości uzyskania dostępu do zmiennych z plików .c.

Abstrakcyjną klasą przedstawioną w projekcie jest klasa *Abstract*, a klasa *AbstractTest* nadpisuje jej metodę.