Przeładowanie nazw funkcji

- Zanim zajmiemy się pojęciem klasy opiszemy tak zwane przeładowanie nazw funkcji. C++ pozwala definiować różne funkcje o tej samej nazwie, pod warunkiem, że różnią się sposobem wywołania. Mogą mieć różne ilości argumentów, argumenty różnych typów, albo różną kolejność argumentów. Na przykład, wyobraźmy sobie, że mamy w programie dwie struktury

```
struct osoba
{
 char nazwisko[50];
 char imie[20];
}

struct etat
{
 char nazwa[100];
 int pensja;
}
```

Przeładowanie nazw funkcji

- Chcielibyśmy zdefiniować sposób wypisywania tych struktur. Tworzymy funkcją, która przyjmuje strukturę i ją wypisuje na ekran. Naturalnie chcielibyśmy nazwać ją `wypisz()`. Możemy utworzyć dwie takie funkcje:

```
void wypisz(osoba x);
void wypisz(etat x);
```

- Każda z tych funkcji może mieć inną definicję. Kompilator rozpozna, po sposobie wywołania, którą z definicji zastosować.

Przeładowanie nazw funkcji

- Na przykład:

```
void wypisz(osoba x)
{
 cout<<"Pracownik: "<<x.imie<<" "<<x.nazwisko<<endl;
}
```

- Podobnie:

```
void wypisz(etat x)
{
 cout<<"Stanowisko: "<<x.etat<<endl;
 cout<<"Uposażenie: "<<x.pensja<<endl;
}
```

- W sumie mamy więc dwie różne funkcje, o tej samej nazwie. Wbrew pozorom jest to bardzo wygodne, i często stosowane. Zamiast przeładowania moglibyśmy nadać tym dwóm funkcjom różne nazwy. Ale podczas pisania programu niektóre nazwy są naturalne. Nie chcemy wnikać w to jak dana funkcja sobie konkretnie radzi z konkretnym wywołaniem, interesuje nas finalny wynik.

Przeładowanie nazw funkcji

- Przypomnijmy operator «. To nie jest funkcja, tylko operator, ale operatory też podlegają przeładowaniu na tej samej zasadzie. Jeżeli kompilator zobaczy

```
int x = 10;
x = x << 2;
```

to podstawi za `x` wartość 40. Jest to bitowe przesunięcie w lewo o 2 pozycje (czyli mnożenie przez 4). Jeżeli natomiast zobaczy

```
cout << "Jakiś napis";
```

po prostu wypisze treść napisu na ekran. Co ma zrobić pozna po kontekście. Jeżeli po lewej stronie operatora jest zmienna, to zastosuje operator przesunięcia bitowego. Jeżeli po lewej stronie jest strumień wyjściowy `cout`, wykona na zmiennej po prawej stronie operację wyjścia.

Przeładowanie nazw funkcji

- Jeżeli kompilator zobaczy

```
float x;
cout << x;
```

  to wypisze wartość zmiennej `x` jako liczby zmiennoprzecinkowej. Tym razem dodatkowy kontekst polega na typie zmiennej do wypisania.
- Oczywiście nie ma powodu, żeby wszystkie funkcje nazywały się tak samo. Powinniśmy nadawać nazwy w sposób logiczny. Za moment, mówiąc o pojęciu klasy zobaczymy naturalne miejsce do przeładowania.
- Jeżeli chcemy korzystać z przeładowania funkcji, pamiętajmy o ważnych szczegółach.
- Funkcje przeładowane muszą różnić się sposobem wywołania. Albo ilość argumentów musi się różnić, albo ich typ, albo kolejność.

Przeładowanie nazw funkcji

- Typ zwracanej wartości nie ma znaczenia. Następująca sytuacja będzie zgłoszona przez kompilator jako błąd

```
void funkcja(int x)
}
...
}
int funkcja(int y)
{
...
}
```

- Obie definicje różnią się tylko typem zwracanej wartości, więc to nie jest przeładowanie. Zastanówmy się, żeby wiedzieć przy wywołaniu którą definicję zastosować, kompilator musiałby „zapuścić" obie definicje na konkretnych danych, i sprawdzić, jakiego typu wychodzi wynik.

Przeładowanie nazw funkcji

- Przypomnijmy, że funkcje mogą posiadać argumenty domyślne. Na przykład, możemy zdefiniować funkcję następująco:

```
void funkcja(int x, int y = 0)
}
...
}
```

- Wtedy możemy wywołać funkcję z dwoma argumentami:

```
funkcja(a,b);
```

  albo z jednym argumentem

```
funkcja(a);
```

- W tym drugim przypadku za drugi argument zostanie podstawiona wartość domyślna 0. Nie możemy więc zdefiniować innej funkcji przeładowanej

```
void funkcja(int x)
}
...
}
```

Przeładowanie nazw funkcji

- To byłby błąd. Kompilator nie wiedziałby, czy przy wywołaniu z jednym parametrem zastosować wersję przeładowaną, czy wersję z argumentem domyślnym.
- Tak naprawdę o argumentach domyślnych można właśnie myśleć jako o przeładowanych wersjach funkcji. Błąd pokazany powyżej kompilator przypuszczalnie zgłosi jako podwójną definicję.
- Kolejną sprawą o której trzeba pamiętać przy przeładowaniu jest to, że C++ może w miarę potrzeby sam rzutować typy zmiennych (jeżeli się da).

Przeładowanie nazw funkcji

- Na przykład, jeżeli mamy definicję:

```
void funkcja(int x)
{
...
}
```

to przy wywołaniu

```
funkcja(0.1);
```

kompilator najpierw zrzutuje `0.1` na `int`, a następnie podstawi jako argument (w tym przypadku podstawi 0). Jeżeli gdzieś, w tym samym zakresie ważności nazw będzie też definicja

```
void funkcja(double x)
{
...
}
```

to kompilator nie będzie zmiennej rzutował, tylko podstawi równo `0.1`, ale do definicji przeładowanej.

Przeładowanie nazw funkcji

- Trzeba o tym pamiętać. Widząc wywołanie funkcji, kompilator najpierw szuka wersji definicji z dokładnie takimi samymi typami.
- W drugiej kolejności, jeżeli kompilator nie znajdzie defnicji funkcji z dokładnie identycznymi typami, próbuje dokonać tak zwanej konwersji trywialnej. Jak pamiętamy niektóre typy są pozornie różne, ale jednak identyczne. Na przykład obie zmienne, `tablica` i `tablica_ptr`

```
char tablica[10];
char *tablica_ptr;
```

są tak naprawdę tego samego typu.

Przeładowanie nazw funkcji

- Jeżeli konwersja trywialna nie wystarcza, kompilator próbuje tak zwanej promocji. Rzutuje typ zmiennej na podobną, ale o większej pojemności. Na przykład, jeżeli wywołanie jest z argumentem typu `char`, a kompilator znalazł definicję funkcji z argumentem typu `int` to dokona promocji i podstawi. Podobnie typ `float` może być promowany do `double`.
- W końcu kompilator próbuje rzutowania standardowego. Tak jak powyżej widzieliśmy, typ `double` może być rzutowany na `int` (przez zaokrąglenie).
- Jeżeli argumentów jest wiele, sytuacja jeszcze bardziej się komplikuje. Zamiast pamiętać wszystkie szczegóły warto więc pamiętać zasadę. Kompilator powinien w miejscu wywołania wiedzieć, którą definicję zastosować.

Klasy

- Używaliśmy poprzednio typu zmiennej definiowanej przez użytkownika, tak zwanej struktury. Zawierała ona wiele składników w jednej całości. Składniki były innymi zmiennymi. Klasa to taka bardziej ogólna struktura. Oprócz różnych zmiennych może też zawierać funkcje.
- Pojęcie klasy czyli typu zmiennej, oraz obiektu, czyli zmiennej takiego typu jest obecnie centralnym pojęciem w wielu językach programowania.
- Wyobraźmy sobie, że w ramach naszego programu chcielibyśmy mieć funkcję, która wyświetla okno, z jakąś zawartością, gdzieś na ekranie. Funkcja, wyświetlająca takie okno musi wiedzieć, w którym miejscu je wyświetlić, jaki ma być rozmiar, jaki kolor tła, jaka czcionka, skąd wziąć zawartość.
- Z takim oknem związane są różne funkcje, na przykład przesuń okno, zmniejsz okno, nadpisz zawartość itp.

Klasy

- Logiczne jest zgromadzenie tych wszystkich danych oraz operujących na nich funkcji (tak zwanych metod klasy) w jedną całość. Wyobraźmy sobie, że strukturę, reprezentującą książki, którą tworzyliśmy poprzednio chcemy rozbudować do klasy. Robimy to następująco:

```
class book
{
    char tytul[1000];
    char autor_nazw[50];
    char autor_imie[20];
    char wydawnictwo[1000];
    int cena;
    int rok_wyd;
    book *nast;
    book *poprz;
};
```

- Jak na razie nie jest to nic zawiłego. Jeżeli chcemy, możemy posługiwać się taką klasą tak jak strukturą. Ale możemy do niej dodać funkcje. Taką typową funkcją jest zapisanie konkretnej zawartości do zmiennych będących składnikami klasy.

Klasy

- Zauważmy że poprzednio, posługując się strukturą, musieliśmy dobrze wiedzieć, jak dane wewnątrz struktury są przechowywane, jak się nazywają. Zastąpimy to jedną funkcją

```
class book
{
  ...
  void zapisz_ksiazke(char tytul[], char autor_nazw[],
     char autor_imie[], char wydawnictwo[], int cena, int rok_wyd);
};
```

- Zauważmy, że wewnątrz definicji klasy funkcję składową tylko zadeklarowaliśmy. Definicja będzie gdzieś dalej. Tak się z reguły postępuje. Można jednak samą definicję też umieścić od razu w definicji klasy. Tak robimy, jeżeli taka definicja jest krótka, rzędu kilku linijek.
- Funkcje, których definicja znajduje się wewnątrz klasy traktowane są przez kompilator jako `inline`. Jeżeli nie chcemy funkcji `inline`, wynieśmy definicję na zewnątrz.

Klasy

- Jeżeli definicja funkcji składowej klasy znajduje się poza klasą, z reguły gdzieś poniżej, wygląda to następująco:

```
class book
{
 ...
 void zapisz_ksiazke(char tyt[], char nazw[],
      char imie[], char wyd[], int c, int rok);
};


 ...


book::zapisz_ksiazke(char tytul[], char autor_nazw[],
  char autor_imie[], char wydawnictwo[], int cena, int rok_wyd)
{
    strcpy(tytul, tyt);
    strcpy(autor_nazw, nazw);
    strcpy(autor_imie, imie);
    strcpy(wydawnictwo, wyd);
    cena = c;
    rok_wyd = rok;
}
```

Klasy

- Zwróćmy uwagę na operator przestrzeni nazw: `book::`. Jest jasne, jaka jest jego rola. Mówi nam, że następująca po nim nazwa odpowiada tej z klasy `book`. Konieczność stosowania tego operatora bierze się stąd, że zakres ważności nazw składników klasy (zmiennych i funkcji) ograniczony jest do danej klasy.
- Jest w tym różnica w porównaniu do zwykłych funkcji (nie będących składnikami klas). Dla zwykłych funkcji zakres ważności nazwy to cały plik, w którym funkcja jest zadeklarowana (od miejsca deklaracji).

Klasy

- Mając zdefiniowaną klasę możemy tworzyć zmienne tego typu. Takie zmienne, których typ jest klasą nazywają się obiektami. Obiekty, tak jak inne zmienne, mogą być statyczne lub dynamiczne. Na przykład, jeżeli wcześniej zdefiniowaliśmy klasę `book`, to możemy napisać

```
book a,b;
  ...
book *c;
  ...
c = new book;
```

- Utworzone zostały dwie zmienne typu `book` statyczne a i b, oraz zmienna dynamiczna, wskazywana przez wskaźnik c.
- Mając utworzone zmienne, możemy odnosić się do ich części składowych tak jak to robiliśmy w przypadku struktur:

```
a.rok_wyd = 1587;
(*c).cena = 45000;
strcpy((*c).tytul,"Teoria Grawitacji");
b.zapisz_ksiazke( ... );
```

Klasy

- Uwaga: możemy odnosić się tylko do składników tak zwanych publicznych. Składniki mogą być też prywatne, nie są wtedy w ogóle widoczne poza obiektem. To jest ważny temat, i zajmiemy się nim niedługo.
- Przypomnijmy, że funkcje będące częściami składowymi klas nazywają się metodami klas.
- Wywołanie metody klasy zawsze zawiera w sobie obiekt wywołujący:

```
b.zapisz_ksiazke( ... );
```

- To jest zupełnie oczywiste. Funkcja musi wiedzieć, na jakich konkretnych danych ma działać. W powyższym przykładzie funkcji zapisującej konkretne wartości w zmiennych zawartych w obiekcie, funkcja musi wiedzieć, o który konkretny obiekt chodzi. Wszystkie obiekty tej samej klasy mają składowe o tych samych nazwach.

Klasy

- Metody klasy są jej częściami składowymi. Jeżeli tworzymy więc obiekty jakiejś klasy każdy z tych obiektów zawiera swoje części składowe, dane, oraz metody. Jeżeli chodzi o dane, to sprawa jest jasna, każdy obiekt ma swoje dane, konkretna wartość jest zapewne różna w różnych obiektach. Ale po co tym wszystkim obiektom własne kopie funkcji, które przecież są wszystkie identyczne?
- Oczywiście C++ bierze to pod uwagę. Nie jest to widoczne dla użytkownika, ale metody obiektów danej klasy przechowywane są w pamięci tylko w jednym egzemplarzu (inaczej niż dane, które każdy obiekt ma swoje). Kiedy wywoływana jest metoda dla dowolnego obiektu, tak naprawdę wywoływany jest ten jeden wspólny egzemplarz.

Klasy

- W takim razie pojawia się problem. Przecież metoda musi wiedzieć, który konkretny obiekt ją wywołał. Jeżeli jest to metoda `zapisz_ksiazke( ... )`, to metoda musi wiedzieć, gdzie konkretnie przekazane dane zapisać. Rozwiązanie jest takie, że metoda, oprócz wszystkich argumentów, dostaje też „po cichu" wskaźnik na obiekt, który ją wywołał. Ten wskaźnik nazywa się `this`, i można się nim posługiwać.
- Wewnątrz klasy można stosować etykiety `public`, `private` oraz `protected`. Każdy składnik należy do jednej z tych kategorii. Składowe kategorii `public` są widoczne na zewnątrz klasy, tak jak składowe struktury. Składowe kategorii `private` i `protected` nie są widoczne na zewnątrz, i tylko metody danej klasy mogą na nich operować (zapisać, odczytać, czy wywołać, jeżeli jest to metoda).

Klasy

- Różnica pomiędzy `private` a `protected` jest subtelna, i nie będziemy jej teraz zgłębiać.
- Domyślna jest etykieta `private`. Etykiety można stosować wielokrotnie, każda jest aktywna aż do następnej.
- Cel tych etykiet jest jasny. Składowe publiczne, to interface, którym klasa komunikuje się z programem. Składowe prywatne to części mechanizmów wewnętrznych, które nie dotyczą zewnętrznego programu.
- Jeżeli chcemy docenić zalety pojęcia klasy wyobraźmy sobie duży program, który jest pisany przez wiele osób, i który musi być wspierany i unowocześniany cały czas. Jeżeli główne elementy programu zorganizowane są w obiektach, to różne klasy mogą pisać i się nimi zajmować różni ludzie. Jedynymi elementami klas, które trzeba konsultować pomiędzy sobą są elementy publiczne.

Klasy

- Podobnie, przy nowych wersjach programu, jeżeli klasa zachowuje niezmienione składowe publiczne (czyli swój interfejs z programem), to można ją w programie podmienić. Nic się nie zmieni z punktu widzenia programu (chociaż nowa klasa może na przykład szybciej działać). Programista ma swobodę zmiany składowych prywatnych bez oglądania się na resztę programu.
- Stosowanie etykiet jest następujące:

```
class book
{
private:
 char tytul[100];
 char autor_nazw[50];
   ...
public:
 void zapisz_ksiazke( ... );
   ...
}
```

Klasy

- Składowe `tytul` i `autor_nazw` są prywatne (etykieta `private` nie jest nawet konieczna, prywatne jest domyślne), natomiast funkcja `zapisz_ksiazke()` jest publiczna. Program może zapisywać dane do obiektu posługując się tą funkcją, i tylko tak. Taki sposób komunikacji z obiektem jest przewidywalny i bezpieczny.
- Klasy z reguły zawierają specjalne funkcje, tak zwane konstruktor i destruktor. Nazwy nie oddają dokładnie funkcjonalności. Wiadomo, jakie instrukcje tworzą obiekty, i jakie je likwidują. Konstruktor jest funkcją wywoływaną automatycznie w momencie tworzenia obiektu. Z reguły jest wykorzystywana do nadania początkowych wartości zmiennym, także do ew. tworzenia potrzebnych zmiennych dynamicznych, będących składowymi klasy. Destruktor z kolei jest wywoływany automatycznie przy likwidacji obiektu. Typowo zajmuje się likwidacją zmiennych dynamicznych, które mogłyby być składowymi klasy.

Klasy

- Konstruktor ma taką samą nazwę jak klasa, a destruktor taką samą nazwę jak klasa, poprzedzoną znakiem ~. Obie metody muszą być publiczne. Inaczej niż inne funkcje nie mają typu zwracanego. Na przykład:

```
class book
{
private:
 char tytul[100];
 char autor_nazw[50];
  ...
public:
 book();
 book(char tyt[]);
 book(char tyt[], char nazw[], char im[]);
 ~book(void);
 void zapisz_ksiazke( ... );

  ...
}
```

- Jak zwykle, definicje konstruktorów i destruktora znajdą się gdzieś poniżej.

Klasy

- Zauważmy, że w powyższym mamy kilka wersji konstruktora. Konstruktor klasy to funkcja która z reguły podlega przeładowaniu. Mając powyższe przeładowane wersje konstruktora możemy definiować obiekty tej klasy na różne sposoby:

```
book a; //powstaje obiekt bez żadnej inicjalizacji

book b("Teoria grawitacji"); //powstaje obiekt z zapisanym tytułem

book c("teoria Grawitacji", "Newton", "Isaac");
                //powstaje obiekt z zapisanym tytułem i autorem
```

- Oczywiście wymaga to odpowiedniego zdefiniowania wszystkich wersji konstruktora:

Klasy

```
book::book(char tyt[])
{
 strcpy(tytul, tyt);
}
book::book(char tyt[], char nazw[], char im[])
{
 strcpy(tytul, tyt);
 strcpy(autor_nazw, nazw);
 strcpy(autor_imie, im);
}
```

- W naszej klasie `book` nie ma potrzeby definiowania destruktora. Niczego specjalnego nie trzeba robić przy usuwaniu z pamięci obiektu tej klasy.
- Do tematu klas wrócimy w drugiej części tego kursu. Jest to kluczowe pojęcie w językach programowania.
- Posługując się klasami warto poszczególne klasy wydzielić do osobnych plików `*.cpp`. Nazwa pliku powinna być taka sama, jak nazwa klasy, i powinny mieć własne pliki nagłówkowe `*.h`.