Beata Jankowska
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

V. Przeciążanie nazw funkcji i operatorów. Biblioteki wejścia-wyjścia w języku C++

1. Przeciążanie nazw funkcji

1.1. W wielu językach dopuszcza się możliwość definiowania w tym samym zakresie widoczności funkcji o identycznych nazwach, lecz różnych sygnaturach. Tę możliwość wykorzystuje się zwykle w odniesieniu do funkcji o takiej samej (podobnej) semantyce, lecz różnych typach parametrów. Taki sposób definiowania funkcji nazywa się **przeciążaniem nazw funkcji**. Wybór właściwej funkcji ze zbioru wszystkich funkcji o przeciążonej nazwie odbywa się **na podstawie analizy jej parametrów aktualnych** i ma miejsce **na etapie kompilacji programu**.

1.2. Przeciążanie nazw funkcji dopuszczono, między innymi, w języku C++. Oto przykładowy zestaw przeciążonych funkcji o nazwie `min`:

```
int min(int x, int y)
{return (x<=y?x:y);}

float min(float x, float y)
{return (x<=y?x:y);}

typedef struct l_zesp
{double rzecz;
 double uroj;} zespolona;
```

```
#include <math>
zespolona min(zespolona x, zespolona y)
{if((pow(x.rzecz,2) + pow(x.uroj,2)) <=
    (pow(y.rzecz,2) + pow(y.uroj,2)))
   return x;
 else
   return y;}

int min(int x, int y, int z)
{int najm=x;
 if (y<najm) najm=y;
 if (z<najm) najm=z;
 return (najm);}
```

Powyższych funkcji użyto w następujących instrukcjach:

```
int a=4, b=9, c1, c2;
float f=3.9, g=-5.6, h;
zespolona z1={-3.0,2.5}, z2={4.0,-1.5}, z3;
c1 = min(a,b);    //c1=4;
c2 = min(a,b,-2); //c2=-2;
h  = min(f,g);    //h=-5.6;
z3 = min(z1,z2);  //z3={-3.0,2.5};
```

1.3. Jeśli w programie zdefiniowano dwie funkcje o tym samym identyfikatorze i jednakowych sygnaturach, to kompilator sygnalizuje błąd (zawsze gdy funkcje są różnych typów; na ogół – gdy funkcje są tego samego typu – jak MS Visual C++; w tym drugim wypadku może ewentualnie zachodzić radykalne zasłanianie).
Przykładowo, przeciążanie nazw funkcji nie zachodzi w następujących sytuacjach:

```
int min(int x, int y)
{return (x<=y?x:y);}
float min(int x, int y)
{return float(x<=y?x:y);} //sygnalizacja błedu
```

```
int min(int x, int y)
{return (x<=y?x:y);}
int min (int x, int y)
{int najm=0;
 if (x<0) najm=x;
 if (y<0) najm=y;
 return najm;} //sygnalizacja bledu
               //(m.in. w MS Visual C++),
               //ew. radykalne zasloniecie
```

1.4. W sytuacji, gdy funkcja została poprawnie przeciążona, lecz nie można dopasować (analiza list parametrów formalnych i aktualnych!) żadnej definicji tej funkcji do pewnego jej wywołania, kompilator sygnalizuje błąd, np.

```
int min(int x, int y)
{return (x<=y?x:y);}
float min(float x, float y)
{return (x<=y?x:y);}
int min(int x, int y, int z)
{int najm=x;
 if (y<najm) najm=y;
 if (z<najm) najm=z;
 return (najm);}
float a=1.0, b=-2.5, c=7.22, d;
zespolona z1={2.5,-1}, z2={-2.1,3.0}, z3={-1.1,-2.7}, z4;
d=min(a,b,c);     //zgodność list parametrów,
                  //dopasowanie po konwersji zawężającej
z4=min(z1,z2,z3); //niezgodność list parametrów,
                  //sygnalizacja błędu
```

Beata Jankowska
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

2. Przeciążanie operatorów

2.1. W języku C++ oprócz nazw funkcji można przeciążyć także większość standardowych operatorów (ta możliwość nie obejmuje, między innymi, operatorów: `.`, `.*`, `::`, `?:`, `sizeof`). Nie wolno natomiast definiować własnych operatorów.

2.2. W ogólności, przeciążanie operatorów odbywa się za pomocą funkcji o charakterze globalnym lub w powiązaniu z pewną klasą. Przy przeciążaniu obowiązują następujące zasady:
- przeciążany operator musi zachować arność operatora standardowego,
- przeciążony operator zachowuje dotychczasowy priorytet oraz właściwość łączności (jeśli operator standardowy ją posiada),
- przeciążone operatory `&&`, `||` i `,` (przecinek) tracą swoje specjalne właściwości (skrócone wartościowanie),
- w wypadku przeciążania przy użyciu funkcji globalnej, jeden z argumentów funkcji musi być typu klasowego, strukturalnego lub wyliczeniowego,
- w wypadku przeciążania w powiązaniu z pewną klasą, dozwolone operatory można przeciążyć na dwa sposoby: przy użyciu funkcji składowej tej klasy lub przy użyciu funkcji zaprzyjaźnionej z tą klasą; w nowej definicji przeciążonego operatora przynajmniej jeden z jego argumentów powinien być obiektem przedmiotowej klasy; w szczególności, w definicji sformułowanej przy użyciu funkcji składowej klasy – musi nim być pierwszy argument operatora (pierwszy, ukryty parametr funkcji),

2.3. Operatory `=`, `[]`, `()` oraz `->` można przeciążyć jedynie za pomocą (niestatycznych) funkcji składowych klasy.

2.4. Nową definicję operatora o symbolicznej nazwie @ można więc zadać:

za pomocą **funkcji globalnej** z nagłówkiem w postaci:
- *typ_funkcji* `operator@ (`*typ_arg1 arg1*`,` *typ_arg2 arg2*`)`, w wypadku, gdy @ jest operatorem binarnym różnym od `=`, `->`, `()` oraz `[]`, a przynajmniej jeden z typów argumentów (*typ_arg1*, *typ_arg2*) jest typem klasowym, strukturalnym lub wyliczeniowym;
- *typ_funkcji* `operator@ (`*typ_arg arg*`)`, w wypadku, gdy @ jest operatorem unarnym, a typ argumentu (*typ_arg*) jest typem klasowym, strukturalnym lub wyliczeniowym;

za pomocą **funkcji składowej klasy** z nagłówkiem w postaci:
- *typ_funkcji* `operator@ (`*typ_arg arg*`)`, w wypadku, gdy @ jest operatorem binarnym (operatory `new`, `new[]`, `delete`, `delete[]` są przy tym traktowane jak metody statyczne),
- *typ_funkcji* `operator@ ()`, w wypadku, gdy @ jest operatorem unarnym;
  (domyślnym argumentem operatora @ jest w obu powyższych wypadkach przedmiotowy obiekt `(*this)`);

za pomocą **funkcji zaprzyjaźnionej z klasą**, z nagłówkiem w postaci:
- `friend` *typ_funkcji* `operator@ (`*typ_arg1 arg1*`,`
  typ_arg2 arg2`)`,
  w wypadku, gdy @ jest operatorem binarnym różnym od `=`, `->`, `()`, oraz `[]`,
- `friend` *typ_funkcji* `operator@ (`*typ_arg arg*`)`,
  w wypadku, gdy @ jest operatorem unarnym,
  gdzie *typ_arg* oraz (*typ_arg1* lub *typ_arg2*) są typami przedmiotowej klasy.

https://pl.wikibooks.org/wiki/C%2B%2B/Przeci%C4%85%C5%BCanie_operator%C3%B3w

2.5. Zdefiniowana niżej klasa `Zespolone` służy do manipulacji na wartościach zespolonych.

```
typedef struct l_zesp
{double rzecz;
 double uroj;} zespolona;

class Zespolone
{private:
 zespolona liczba;
 public:
 Zespolone()
 {liczba.rzecz=0.0;
  liczba.uroj=0.0;}
 ~Zespolone()
 {}
 void Zapisz_liczbe(double x, double y)
 {liczba = {x,y};}
 zespolona Podaj_liczbe()
 {return liczba;}
 void Wyświetl_liczbe()
 {if (liczba.uroj>=0)
      printf("%f+%f\n",
             liczba.rzecz,abs(liczba.uroj));
  else
      printf("%f-%f\n",
             liczba.rzecz,abs(liczba.uroj));}
 Zespolone& Dodaj(zespolona zesp_arg) //def.1a
 {Zapisz_liczbe(liczba.rzecz+zesp_arg.rzecz,
                liczba.uroj+zesp_arg.uroj);
  return *this;}
 Zespolone& Przeciwna()//def.1b
 {Zapisz_liczbe(-liczba.rzecz, -liczba.uroj);
  return *this;}
}//koniec definicji klasy Zespolone
```

2.6. W miejsce funkcji składowych `Dodaj` i `Przeciwna` można w klasie `Zespolone` zdefiniować ich odpowiedniki operatorowe, przeciążające operatory: 2-argumentowy dodawania z przypisaniem `+=` i 1-argumentowy zamiany na liczbę przeciwną `-`.

```
Zespolone& operator+=(zespolona zesp_arg)  //def.2a
{Zapisz_liczbe(liczba.rzecz+zesp_arg.rzecz,
               liczba.uroj+zesp_arg.uroj);
 return *this;}

Zespolone& operator-()                     //def.2b
{Zapisz_liczbe(-liczba.rzecz, -liczba.uroj);
 return *this;}
```

Oba przeciążone operatory różnią się od swych odpowiedników-metod jedynie nazwą: ich typy, sygnatury oraz szczegółowe definicje mają identyczną postać. To proste zastąpienie było możliwe dzięki temu, że ukrytym argumentem metod `Dodaj` i `Przeciwna` oraz operatorów `+=` i `-` jest obiekt przedmiotowej klasy (odwołania do pól `liczba.rzecz` i `liczba.uroj` są w istocie odwołaniami `(*this).liczba.rzecz` i `(*this).liczba.uroj`).

2.7. Równoważne przeciążenia operatorów dodawania z przypisaniem `+=` i zamiany na liczbę przeciwną `-` można też zdefiniować przy użyciu funkcji zaprzyjaźnionych z klasą `Zespolone`:

```
//w ciele klasy
friend Zespolone& operator+=
               (Zespolone&, zespolona);//dekl.3a
friend Zespolone& operator-
               (Zespolone&);           //dekl.3b
```

```
//poza cialem klasy
Zespolone& operator+= (Zespolone& zsp,
                       zespolona lz)    //def.3a
{zsp.Zapisz_liczbe
            (zsp.liczba.rzecz+lz.rzecz,
             zsp.liczba.uroj+lz.uroj);
 return zsp;}
Zespolone& operator- (Zespolone& zsp)   //def.3b
{zsp.Zapisz_liczbe
            (-zsp.liczba.rzecz,
             -zsp.liczba.uroj);
 return zsp;}
```

2.8. Przy założeniu następujących deklaracji i instrukcji tworzenia obiektów klasy `Zespolone`:

```
zespolona lz = {13.0,-5.2}
Zespolone kz1 = Zespolone();
Zespolone kz2 = Zespolone();
kz1.Zapisz_liczbe(2.5,-7.6);
kz2.Zapisz_liczbe(-4.61,-0.02);
```

poniższe działania są sobie równoważne:

```
kz1.Dodaj(lz).Wyświetl_liczbe(); //(1a)
kz2.Przeciwna().Wyświetl_liczbe(); //(1b)

(kz1+=lz).Wyświetl_liczbe(); //(2a)
kz1.operator+=(lz).Wyświetl_liczbe();
(-kz2).Wyświetl_liczbe(); //(2b)
kz2.operator-().Wyświetl_liczbe();

(kz1+=lz).Wyświetl_liczbe(); //(3a)
operator +=(kz1,lz).Wyświetl_liczbe();
(-kz2).Wyświetl_liczbe(); //(3b)
operator -(kz2).Wyświetl_liczbe();
```

i dają rezultat w postaci:

```
15.5-12.8i
4.61+0.02i
```

2.9. W celu przeciążenia unarnego operatora postfiksowego (np. `x++`), należy funkcję określającą przeciążenie zdefiniować jako funkcję w postaci:

typ_funkcji `operator@ (int)`,

z dodatkowym argumentem typu `int`, o znaczeniu wyłącznie rozpoznawczym.

3. Biblioteki wejścia-wyjścia w języku C++

3.1. Operacje wejścia-wyjścia należą do tej grupy instrukcji, bez których nie da się napisać "sensownie działającego" programu. W języku C++ mamy do dyspozycji:

- zestaw **klasycznych funkcji wejścia-wyjścia**, tworzących **bibliotekę standardową** dla programów sformułowanych w językach C i C++ (`'stdio'`) oraz **biblioteki dodatkowe**, charakterystyczne dla rozważanego środowiska uruchomieniowego (np. `'conio'`); powyższe funkcje operują na urządzeniach fizycznych (klawiatura, monitor) i plikach fizycznych (bajtowych); używają do tego celu abstrakcyjnych strumieni, reprezentowanych w postaci wskaźników do obiektów typu `FILE`;
- zestaw **obiektowych funkcji i operatorów wejścia-wyjścia**, zdefiniowanych w **bibliotekach strumieni wejściowo-wyjściowych**, dostarczanych z oprogramowaniem podstawowym języka C++.

3.2. W pliku źródłowym programu korzystającego z klasycznych bibliotek wejścia-wyjścia (dostarczanych wraz z oprogramowaniem) trzeba zamieścić dyrektywy:

```
#include <stdio.h>
#include <conio.h>  .
```

3.3. Oto zestaw najczęściej używanych klasycznych funkcji wejścia-wyjścia, przeznaczonych do obsługi standardowego wejścia, standardowego wyjścia i plików dyskowych:

Nazwa	Typ	Parametry	Wynik	Plik nagłówkowy
Obsługa standardowego wejścia				
`getchar`	`int`	brak	kod ASCII znaku związanego z pierwszym klawiszem z bufora klawiatury	`conio.h`
`getche`	`int`	brak	jak wyżej, z dodatkowym wyświetleniem znaku na ekranie	`conio.h`
`scanf`	`int`	**1.** `char*` **2.** *typ_wsk1* **3.** *typ_wsk2* …..	liczba pól (ciagów znaków) wczytanych poprawnie, zgodnie z zadanym formatem (**1**), i podstawionych za zmienne wskazywane przez (**2**), (**3**), …	`cstdio.h`
`gets`	`char*`	**1.** `char*`	wskaźnik do obszaru (**1**), w którym umieszczono wiersz (ciąg znaków zakończony znakiem końca wiersza) wprowadzony ze standardowego wejścia i dodatkowo wyświetlony na ekranie	`cstdio.h`

Nazwa	Typ	Parametry	Wynik	Plik nagłówkowy
Obsługa standardowego wyjścia				
`putchar`	`int`	**1.** `int`	kod ASCII znaku (**1**) wyprowadzonego na standardowe wyjście (ekran)	`cstdio.h`
`puts`	`int`	**1.** `char*`	kod ASCII ostatniego znaku spośród wszystkich wyprowadzonych na standardowe wyjście znaków ciągu (**1**)	`cstdio.h`
`printf`	`int`	**1.** `char*` **2.** *typ1* **3.** *typ2* …..	liczba wyprowadzonych na standardowe wyjście, zgodnie z zadanym formatem (**1**), znaków reprezentujących wartości wyrażeń (**2**), (**3**), …	`cstdio.h`
`clrscr`	`void`	brak	brak rezultatu; skutek uboczny polega na wypełnieniu ekranu monitora (standardowe wyjście) znakami spacji i umieszczeniu kursora w lewym górnym narożniku ekranu	`conio.h`

Nazwa	Typ	Parametry	Wynik	Plik nagłówkowy
Obsługa plików dyskowych				
`fopen`	`FILE*`	**1.** `char*` **2.** `char*`	wskaźnik pomyślnie otwartego pliku (**1**) o trybie dostępu ustalonym przez (**2**), lub `NULL`	`cstdio.h`
`fclose`	`int`	**1.** `FILE*`	`0`, w wypadku pomyślnego zamknięcia pliku wskazanego przez (**1**); `EOF`, w przeciwnym wypadku	`cstdio.h`
`fseek`	`int`	**1.** `FILE*` **2.** `long` **3.** `int`	`0`, w wypadku poprawnego ustalenia aktualnej pozycji pliku (**1**) na bajcie o numerze (**2**), ewent. w pozycji określonej przez stałą (**3**); wartość nieokreślona, w przeciwnym wypadku	`cstdio.h`
`feof`	`int`	**1.** `FILE*`	wartość niezerowa, gdy podczas ostatniej operacji odczytu napotkano koniec pliku (**1**); `0`, w przeciwnym wypadku	`cstdio.h`
`fgetc`	`int`	**1.** `FILE*`	liczba całkowita reprezentująca bieżący znak wczytany z pliku (**1**)	`cstdio.h`
`fputc`	`int`	**1.** `int` **2.** `FILE*`	kod ASCII znaku (**1**), w wypadku jego poprawnego zapisu do pliku (**2**); `EOF`, w przeciwnym wypadku	`cstdio.h`

Nazwa	Typ	Parametry	Wynik	Plik nagłówkowy
Obsługa plików dyskowych - ciąg dalszy				
`fscanf`	`int`	**1.** `FILE*` **2.** `char*` **3.** *typ_wsk1* **4.** *typ_wsk2* …..	liczba ciągów znaków, które zostały poprawnie odczytane z pliku (**1**) zgodnie z formatem (**2**), a następnie przetworzone do postaci binarnej i zapamiętane w obszarach pamięci wskazywanych przez wskaźniki (**3**), (**4**), …; `EOF`, w wypadku napotkania końca pliku	`stdio.h`
`fprintf`	`int`	**1.** `FILE*` **2.** `char*` **3.** *typ1* **4.** *typ2* …..	liczba bajtów zajmowanych przez zapisane do pliku (1), zgodnie z formatem (2), ciągi znaków będace wartościami wyrażeń (3), (4), …; `EOF`, w wypadku nieudanego zapisu	`stdio.h`
`fread`	`int`	**1.** *typ_wsk1* **2.** `int` **3.** `int` **4.** `FILE*`	liczba (**3**) struktur danych o rozmiarze (**2**) odczytanych z pliku (**4**) i zapamiętanych w obszarze pamięci wskazywanym przez (**1**); `0`, w wypadku napotkania końca pliku	`stdio.h`
`fwrite`	`int`	**1.** *typ_wsk1* **2.** `int` **3.** `int` **4.** `FILE*`	liczba (**3**) struktur danych o rozmiarze (**2**) pobranych z obszaru pamięci wskazywanego przez (**1**) i zapisanych do pliku (**4**); `0`, w wypadku wystapienia błędu	`stdio.h`

3.4. Biblioteki strumieni wejściowo-wyjściowych języka C++ zawierają definicje klas i obiektów umożliwiających obiektową implementację operacji wejścia-wyjścia.

Przede wszystkim, zawarto w nich definicje klas do tworzenia standardowych strumieni danych (`istream` i `ostream`) oraz definicje samych strumieni danych, reprezentujących:
- standardowe wejście `stdin` - obiekt `cin` (typu `istream`),
- standardowe wyjście `stdout` - obiekt `cout` (typu `ostream`),
- standardowy zbiór z informacjami o błędach `stderr` - obiekt `cerr` (typu `ostream`),

a także klas do tworzenia obiektów reprezentujących:
- dyskowe pliki wejściowe – klasa `ifstream`,
- dyskowe pliki wyjściowe – klasa `ofstream`.

W klasie `ostream` zamieszczono funkcje i operatory umożliwiające wysyłanie danych na standardowe wyjście, w klasie `istream` – funkcje i operatory umożliwiające pobieranie danych ze standardowego wejścia, a w klasach `ifstream`, `ofstream` oraz `fstream` – funkcje i operatory do działań na plikach dyskowych.

3.5. W celu skorzystania z bibliotek strumieni wejściowo-wyjściowych, należy w pliku źródłowym programu zamieścić dyrektywy ich przyłączania:

- `#include <iostream>` lub
- `#include <fstream>` .

3.6. Zestaw najczęściej używanych obiektowych operatorów i funkcji wejścia-wyjścia obejmuje:

Nazwa	Typ	Parametry	Wynik	Plik nagłówkowy
Obsługa standardowego wyjścia				
`operator<<` (dodatkowa możliwosc przeciążenia operatora dla wysyłania składowych klasy)	`ostream&`	**1.** `ostream&` **2.** *typ liczbowy*, `char*` lub `void*`	referencja do obiektu (**1**) reprezentującego standardowy strumień wyjściowy, do którego wysłano ciąg znaków uzyskany w efekcie konwersji binarno-znakowej wartości (**2**)	`iostream`
`flush`	`ostream&`	`void`	referencja do obiektu repre-zentującego standardowy strumień wyjściowy, do którego skierowano znaki zgromadzone w buforze	`iostream`
`put`	`ostream&`	**1.** `char`	referencja do obiektu repre-zentującego standardowy strumień wyjściowy, do którego skierowano znak (**1**)	`iostream`
`write`	`ostream&`	**1.** `char*` **2.** `int`	referencja do obiektu repre-zentującego standardowy strumień wyjściowy, do którego skierowano znaki (**1**) w ogólnej liczbie (**2**)	`iostream`

Nazwa	Typ	Parametry	Wynik	Plik nagłówkowy
Obsługa standardowego wejścia				
`operator>>` (dodatkowa możliwosc przeciążenia operatora dla wprowadzania składowych klasy)	`istream&`	**1.** `istream&` **2.** *typ liczbowy* lub `char*`	referencja do obiektu (**1**) reprezentującego standardowy strumień wejściowy, z którego pobrano ciąg znaków, który nastepnie – po przeprowadzeniu konwersji znakowo-binarnej – przypisano zmiennej (**2**)	`iostream`
`get`	`istream&`	`void`	referencja do obiektu repre-zentującego standardowy strumień wejściowy, z którego pobrano jeden znak	`iostream`
`read`	`istream&`	**1.** `char*` **2.** `int`	referencja do obiektu repre-zentującego standardowy strumień wejściowy, z którego pobrano – wprowadzając równocześnie do bufora (**1**) - ciąg znaków w ogólnej licz-bie (**2**)	`iostream`

Nazwa	Typ	Parametry	Wynik	Plik nagłówkowy
Obsługa plików dyskowych				
`open`	`ifstream&` lub `ofstream&`	**1.** `char*` **2.** *flaga* (opcja)	referencja do obiektu reprezentującego strumień wejściowy/wyjściowy po uprzednim związaniu z nim i otwarciu pliku o ścieżce dostępu (**1**), z trybem dostępu określonym flagą (**2**)	`fstream`
`close`	`ifstream&` lub `ofstream&`	`void`	referencja do obiektu reprezentującego strumień wejściowy/wyjściowy po uprzednim zamknięciu związanego z nim pliku	`fstream`
`operator<<`	`ofstream&`	**1.** `ofstream&` **2.** *typ liczbowy* lub `char*`	referencja do obiektu (**1**) reprezentującego strumień wyjściowy, do którego wysłano ciąg znaków uzyskany w efekcie konwersji binarno-znakowej wartości (**2**)	`fstream`
`operator>>`	`ifstream&`	**1.** `ifstream&` **2.** *typ liczbowy* lub `char*`	referencja do obiektu (**1**) reprezentującego strumień wejściowy, z którego pobrano ciąg znaków, kóry następnie przypisano – po przeprowadzeniu konwersji znakowo-binarnej – zmiennej (**2**)	`fstream`

3.7. W celu ustalenia formatu danych pobieranych ze strumieni wejściowych (lub wysyłanych do strumieni wyjściowych) można się posłużyć flagami, zdefiniowanymi w klasie `ios`. Każda z flag, związana z określoną pozycją binarną pewnej składowej typu `long int` z klasy `ios`, określa format wprowadzania/ wyprowadzania danych. Do ustawiania flag

służą specjalne makroinstrukcje, zwane manipulatorami (np. `ws`, `endl`, `setw(`*szer*`)`, `hex`, `dec`, `oct`). Wynikiem działania dowolnego manipulatora jest referencja do bieżącego strumienia. Aby móc korzystać z manipulatorów, należy przyłączyć do programu bibliotekę `<iomanip>`.