PREŤAŽOVANIE (OVERLOADING)

Preťaženie

- Viacero funkcií (metód) s tým istým menom
- Rozlíšenie na základe počtu parametrov, typu parametra plus mínus konverzie
- C++ umožňuje preťažiť štandardné operátory ako napríklad + - * / ...
- C++ umožňuje definovať vlasntné konverzie typov

Preťaženie: Výber metódy na základe typu

```
class A {} ;  enum E {C1, C2 };

void fun(char c)              {  printf("fun(char) ");}
void fun(int i)               {  printf("fun(int) ");}
void fun(long int i)          {  printf("fun(long int) ");}
void fun(E i)                 {  printf("fun(E) ");}
void fun(A a)                 {  printf("fun(A) ");}
void fun(A *pa)               {  printf("fun(A*) ");}
void fun(int i, int j)        {  printf("fun(int, int) ");}

int main() {
  A a;
  fun('a');   fun(1);   fun(1L);   fun(C1);
  fun(a);     fun(&a);
  fun(2, 3);
}
```

Preťaženie operátorov

```
enum E {
     C0, C1, C2, C3
};

E operator +(E x, E y) {
     printf("moje enum plus");
     return(C3);
}

int main() {
       enum E x,y,z;
       x = y + z;
       printf("x == %d \n", x);
}
```

Typ "referencia" (potrebný na pochopenie preťažených operátorov)

```
int main() {
    int a, b, c;
    int &r = a;
    a = 5;
    r = 6;
    printf("%d ", a);
}
```

Dá sa povedať, že referencia je pointer, ktorý nikdy nemôže byť NULL a syntakticky sa správa ako originálny typ.

Vypíše: 6

Typ "referencia"

```
struct A {
   int u;
   ...
} ;

int fcia(A &x, B &y) {
    x.u = 0;
    ...
}

int main() {
    A  a;
    A b;
    fcia(a, b);
}
```

Syntakticky sa zdá, že posielam celé objekty, ale v skutočnosti sa pošle iba smerník.

Typ "referencia" (zrada)

Čo vypíše ?

```
int incrementAndMultiply(int &x, int &y) {
    x ++;
    return(x * y);
}

int main() {
    int b, c;
    b = 2;
    c = incrementAndMultiply(b, b);
    printf("%d ", c);
}
```

Preťažené operátory a referencie

```
class  FDCN {
    double a, b, c, d, e;
};

FDCN &operator +(FDCN &x, FDCN &y) {
    printf("FDCN plus");
    FDCN *res = new FDCN();
    ...
    return(*res);
}

int main() {
      FDCN x,y,z;
      x = y + z;
}
```

Five Dimensional Complex Number ...

Preťažené operátory zvyčajne pracujú s referenciami. Ich parameter je referencia a vracajú referenciu na výsledok. Vlastne, preťažovanie operátorov bol hlavný dôvod, prečo referencie boli do C++ zavedené.

Preťažovanie operátorov v triedach

```
class A {
  public:
  A &operator +(A   &y) {
     printf("moje plus");
     return(*this);
  }
};

int main() {
  A a,b,c;
  a = b + c;
}
```

Operátory v triedach majú aritu o jedna menšiu ako by sme čakali. Prvým parametrom je implicitný parameter (daný objekt).

Aké operátory možno preťažiť?

- Temer všetky
- T.j. všetky okrem . .* :: ?:
- Operátory = (priradenie), [ ] (indexácia), -> (prístup k políčku) a n-árne (, ,) (vyvolanie funkcie) môžu byť preťažené len vo vnútri triedy.

Preťaženie príklad 2

```
class A {
  public:
  A& operator *(A &x) {
    printf("moje krat ");
    return(*this);
  }
  A& operator =(A &x) {
    printf("moje rovna sa ");
    return(*this);
  }
};

int main() {
  A a,b,c;
  a = b * c;
}
```

Preťaženie operátorov ++ a -- (problém)

```
class A {
public:
  A &operator ++() { ... }
};

int main() {
   A a;
   a ++;  // ???
   ++ a;  // ???
}
```

Preťaženie operátorov ++ a --

```
class A {
public:
 A &operator ++() { ... }          // prefix
 A &operator ++(int x) { ... }     // postfix
};

int main() {
    A a;
    a ++;
    ++ a;
}
```

Copy konštruktor

```
class A {
public:
  A() {
    printf("default constructor\n");
  }
  A(A &a) {
    printf("copy constructor\n");
  }
  A& operator=(A& a) {
    printf("assignement\n");
  }
};

int main() {
  A a;
  A b = a;
  b = a;
}
```

Užívateľom definované konverzie typov

```
class A {
public:
    int i;
    operator int() {
        printf("hi ");
        return(i);
    }
};

int main() {
    int x;
    A   a;
    x = a;
}
```

Vypíše:
hi

Smart pointers

- Trieda, ktorá implementuje užívateľom definované smerníky (pointre).
- Cieľom bolo dať programátorom nástroj na implementáciu takejto triedy. Niečo podobné ako bolo preťaženie aritmetických operátorov, ktoré dalo možnosť na implementáciu užívateľom definovanej aritmetiky.
- Umožnilo by to úplnú a transparentnú kontrolu nad daným dátovým typom, včítane správy pamäte napríklad počítadlom referencií, atď.

Smart pointers

- Implementácia takejto triedy je možná
- Smerníková aritmetika sa predefinuje preťažením aritmetických operátorov.
- Increment, decrement (++, --) obdobne
- Indexovanie ([]) sa tiež dá ľahko predefinovať ako operator [] (int i) {...}.
- Ale ako predefinovať operátor -> ? Nie je to obyčajný operátor ako ostatné. Druhá časť výrazu x->toto je proste len meno políčka, alebo metódy, nie je to žiaden objekt existujúci počas behu programu, ktorého hodnota by sa dala poslať metóde.

Smart pointers: riešenie C++

- Operátor -> sa chápe ako unárny operátor, ktorý transformuje objekt na pointer iného typu. Na výsledný pointer s potom aplikuje pôvodný operátor ->.

```
class A {
public:
    int toto;
};

class B {
   ...
   A *operator ->() {
   ...
   }
};
```

```
int main() {
   B p;
   p->toto;
}
```

```
int main() {
   B p;
   (p.operator->)()->toto;
}
```

Výber preťaženej metódy na základe typu

```
void fun(short int i) {  printf("fun(short int) ");}
void fun(long int i) {  printf("fun(long int) ");}


int main() {
  fun('a');          // ???
}
```

Hierarchia číselných konverzií C++ (zjednodušene)

- Presná korešpondencia (exact match). Zahŕňa konverziu pole <-> pointer, funkcia<->pointer, typy líšiace sa len modifikátormi `const` a `volatile`.
- Povýšenia na int (integral promotions). Napríklad char na int, short na int, unsigned short na int, ...
- Povýšenie float na double.
- Ostatné built-in konverzie. Napr. signed <-> unsigned, long -> int, short->long, long->short, double->int, ...
- Užívateľom definované konverzie. Napríklad A -> int. Nie sú prípustné konverzie vzniknuté postupnosťou užívateľom definovaných konverzií.

Hierarchia konverzií smerníkov

- Konverzia 0 na X *
- Konverzia X * na void *
- Konverzia ExtendedClass * na BaseClass *.

Výber preťaženej metódy na základe typu

- Nájde sa "najbližší" scope obsahujúci danú funkciu (funkciu s daným menom).
- Spomedzi funkcií v danom scope (bez ohľadu na viditeľnosť public, private, ...) sa vyberú funkcie s rovnakým možným počtom parametrov. T.j. doplnia sa defaultné parametre (int i=0).
- Z nich sa vyberie funkcia ktorej parametre korešpondujú najlepšie vzhľadom k predchádzajúcim hierarchiam konverzií
- Otestuje sa viditeľnosť

Výber preťaženej metódy na základe typu

- V prípade viacerých parametrov sa funkcia f1 považuje za lepšieho kandidáta ako f2 ak pre každý parameter je konverzia pre f1 lepšia alebo rovnako dobrá ako pre f2 a navyše aspoň pre jeden z parametrov je ostro lepšia.
- V prípade rôzneho počtu parametrov sa korešpondencia k ... (ellipsis, variabilný počet parametrov) považuje za najhoršiu možnú konverziu v hierarchii

Príklad

```
void fun(short int i)    {  printf("fun(short int) ");}
void fun(int i)          {  printf("fun(int) ");}
void fun(long int i)     {  printf("fun(long int) ");}

int main() {
  fun('c');
}
```

Vypíše:

Príklad

```
void fun(short int i)    {  printf("fun(short int) ");}
void fun(int i)          {  printf("fun(int) ");}
void fun(long int i)     {  printf("fun(long int) ");}

int main() {
  fun('c');
}
```

Vypíše:

```
fun(int)
```

Príklad

```
void fun(short int i) {  printf("fun(short int) ");}
void fun(int i) {  printf("fun(int) ");}
void fun(long int i) {  printf("fun(long int) ");}

int main() {
  fun(1U);     // fun( (unsigned)1 );
}
```

Vypíše

Príklad

```
void fun(short int i) {  printf("fun(short int) ");}
void fun(int i) {  printf("fun(int) ");}
void fun(long int i) {  printf("fun(long int) ");}

int main() {
  fun(1U);     // fun( (unsigned)1 );
}
```

Vypíše chybu !!

```
class Complex {...};
void f(int, double) {...}
void f(double, int) {...}
void f(Complex, int) {...}
void f(int ...) {...}
void f(Complex ...) {...}

int main() {
  Complex z;
  f(1,  2.0);
  f(1.0, 2);
  f(z, 1.2);
  f(z, 1, 2);
  f(1.0, z);
  f(1, 1);
}
```

```
class Complex {...};
void f(int, double) {...}
void f(double, int) {...}
void f(Complex, int) {...}
void f(int ...) {...}
void f(Complex ...) {...}

int main() {
  Complex z;
  f(1,  2.0);    // f(int, double)
  f(1.0, 2);     // f(double, int)
  f(z, 1.2);     // f(complex, int)
  f(z, 1, 2);    // f(complex ...)
  f(1.0, z);     // f(int ...)
  f(1, 1);       // Chyba!
}
```

Pre porovnanie, jazyk Java

- Nevyberá scope, množina kandidátov sa vyberá z celej hierarchie tried.
- Public / private sa berie do úvahy pri výbere množiny kandidátov
- Pri konvertovaní typov sú možné len konverzie z nižšieho typu na vyžší.

Kopírovanie štruktúr / tried

```
struct X {
    int i, j;
};

struct Y {
    struct X  x;
    double    d;
};

int main() {
  Y  a, b;
  a = b;
}
```

V C kopíruje blok pamäte a v C++?

Kopírovanie štruktúr / tried

```
struct X {
    int i, j;
    X &operator =(X &x) { printf("Uff\n"); }
};

struct Y {
    struct X  x;
    double    d;
};

int main() {
  Y  a, b;
  a = b;
}
```