Tomáš Oberhuber

Faculty of Nuclear Sciences and Physical Engineering
Czech Technical University in Prague

Video na Youtube

Úvod do OOP

Chceme napsat program pro manipulaci se dvěma funkcemi typu

- $g(x) = ax^2 + bx + c$,
  - a, b, c jsou koeficienty polynomu druhého stupně
- $h(x) = A\sin(fx + p)$,
  - A je amplituda, f je frekvence a p je fáze

Chceme napsat program v jazyce C pro vykreslení těchto funkcí.

Úvod do OOP

```
1  enum Function { quadratic, sinus };
2
3  void drawFunction( Function function,
4                     double left, double right, double step,
5                     double a, double b, double c,
6                     double A, double f, double p )
7  {
8      for( double x = left; x <= right; x += step )
9      {
10         double fx = 0;
11         switch( function)
12         {
13             case quadratic:
14                 fx = ( a * x + b ) * x + c;
15                 break;
16             case sinus:
17                 fx = A * sin( f * x + p );
18                 break;
19             default:
20                 printf( "Unknown function. \n" );
21                 abort();
22         }
23         printf( " %f %f \n", x, fx );
24     }
25 }
```

Úvod do OOP

Volání této funkce by pak vypadalo takto:

```
1  int main( int argc, char* argv[] )
2  {
3          /****
4           * Evaluate f(x) = 2 x^2 + 3 x + 1 on <0,1>
5           */
6           drawFunction( quadratic,     // we work with quadratic function
7                         0.0, 1.0,      // on interval <0, 1>
8                         0.1,           // we draw it with resolution 0.1
9                                        //    -> 11 points on the interval
10                        2.0, 3.0, 1.0,// polynom coefficients
11                        0.0, 0.0, 0.0 // sinus coefficienst - do not make
12                         );
13
14         /****
15          * Evaluate f(x) = 2 sin( 5 *x + 1 ) on <0,1>
16          */
17          drawFunction( sinus,         // we work with sine function
18                        0.0, 1.0,      // on interval <0, 1>
19                        0.1,           // we draw it with resolution 0.1
20                                       //    -> 11 points on the interval
21                        0.0, 0.0, 0.0,// polynom coefficients - do not mak
22                        2.0, 5.0, 1.0 // sinus coefficients
23                         );
24
25 }
```

Úvod do OOP

Tento přístup má několik nevýhod:

- i když vykresluji vždy jen jednu funkci, musím předávat parametry pro obě
  - jedna sada parametrů je vždy zbytečná
  - mohl bych sice použít jednu sadu parametrů pro obě funkce, ale pak bych přišel o přirozené pojmenování parametrů tak, jak jsme zvyklí z teorie
- funkce `drawFunction` tak má příliš mnoho parametrů a při jejich předávání snadno dojde k chybě
- s každou další funkcí bych musel měnit parametry funkce `drawFunction`
- proměnná `function` může mít omylem špatnou hodnotu

Úvod do OOP

Vidíme, že máme na jedné straně funkční předpis (instrukce), např.:

```
1 fx = ( a * x + b ) * x + c;
```

a na druhé straně parametry (data), např.:

```
1 double a, double b, double c
```

Řešením našeho problému je obalení instrukcí i dat do jednoho balíčku, tj. **objektu**.

Úvod do OOP

Nejprve obalíme data:

```
1  enum Function { quadratic, sinus };
2
3  struct Quadratic
4  {
5     double a, b, c;
6  }
7
8  struct Sinus
9  {
10    double A, f, p;
11 }
```

Úvod do OOP

```
1  void drawFunction( Function function,
2                     double left, double right, double step,
3                     void* data )
4  {
5      for( double x = left; x <= right; x += step )
6      {
7          double fx = 0;
8          switch( function)
9          {
10             case quadratic:
11             {
12                 Quadratic* f = ( Qudratic* ) data;
13                 fx = ( f->a * x + f->b ) * x + f->c;
14                 break;
15             }
16             case sinus:
17             {
18                 Sinus* f = ( Sinus* ) data;
19                 fx = f->A * sin( f->f * x + f->p );
20                 break;
21             }
22             default:
23                 printf( "Unknown function. \n" );
24                 abort();
25         }
26         printf( " %f %f \n", x, fx );
27     }
28 }
```

Úvod do OOP

Volání této funkce by pak vypadalo takto:

```
1  int main( int argc, char* argv[] )
2  {
3          /****
4           * Evaluate f(x) = 2 x^2 + 3 x + 1 on <0,1>
5           */
6          Quadratic f1;
7          f1.a = 2.0;
8          f1.b = 3.0;
9          f1.c = 1.0;
10         drawFunction( quadratic, // we work with quadratic function
11                       0.0, 1.0,  // on interval <0, 1>
12                       0.1,       // we draw it with resolution 0.1
13                                  //    -> 11 points on the interval
14                       &f1        // pointer to function data
15                        );
16
17         /****
18          * Evaluate f(x) = 2 sin( 5 *x + 1 ) on <0,1>
19          */
20         Sinus f2;
21         f2.A = 2.0;
22         f2.f = 5.0;
23         f2.p = 1.0;
24         drawFunction( sinus,     // we work with sine function
25                       0.0, 1.0, // on interval <0, 1>
26                       0.1,       // we draw it with resolution 0.1
27                                  //    -> 11 points on the interval
28                       &f2        // pointer to function data
29                        );
30
31 }
```

Úvod do OOP

- tím jsme se zbavili problému s předáváním parametrů
- výpočet hodnoty je však stále prováděn nešikovně
- pří volání funkce `drawFunction` musíme dávat dobrý pozor, aby spolu správně seděl typ předávaných dat `data` a hodnota proměnné `function`
- proto nejprve kód pro výpočet hodnoty funkce přesuneme k samotným parametrům

Úvod do OOP

```
1  enum Function { quadratic, sinus };
2
3  struct Quadratic
4  {
5     double a, b, c;
6
7     double getFx( double x )
8     {
9        return ( a * x + b ) * x + c;
10    }
11 }
12
13 struct Sinus
14 {
15    double A, f, p;
16
17    double getFx( double x )
18    {
19       fx = A * sin( f * x + p );
20    }
21 }
```

Úvod do OOP

```
1  void drawFunction( Function function,
2                     double left, double right, double step,
3                     void* data )
4  {
5      for( double x = left; x <= right; x += step )
6      {
7          double fx = 0;
8          switch( function)
9          {
10             case quadratic:
11             {
12                 Quadratic* f = ( Qudratic* ) data;
13                 fx = f->getFx( x );
14                 break;
15             }
16             case sinus:
17             {
18                 Sinus* f = ( Sinus* ) data;
19                 fx = f->getFx( x );
20                 break;
21             }
22             default:
23                 printf( "Unknown function. \n" );
24                 abort();
25         }
26         printf( " %f %f \n", x, fx );
27     }
28 }
```

Volání této funkce by se nezměnilo.

Úvod do OOP

Nyní k parametrů přidáme i typ funkce:

```
1  enum Function { quadratic, sinus };
2
3  struct Quadratic
4  {
5     double a, b, c;
6
7     double getFx( double x )
8     {
9        return ( a * x + b ) * x + c;
10    }
11
12    Function getType()
13    {
14       return quadratic;
15    }
16 }
17
18 struct Sinus
19 {
20    double A, f, p;
21
22    double getFx( double x )
23    {
24       fx = A * sin( f * x + p );
25    }
26
27    Function getType()
28    {
29       return sinus;
30    }
31 }
```

Úvod do OOP

Volání funkce `drawFunction` se pak změní takto:

```
1  int main( int argc, char* argv[] )
2  {
3          /****
4           * Evaluate f(x) = 2 x^2 + 3 x + 1 on <0,1>
5           */
6          Quadratic f1;
7          f1.a = 2.0;
8          f1.b = 3.0;
9          f1.c = 1.0;
10         drawFunction( f1.getType(), // WE DO NOT NEED TO CARE ABOUT FUNCTION TYPE
11                       0.0, 1.0,     // on interval <0, 1>
12                       0.1,          // we draw it with resolution 0.1
13                                     //    -> 11 points on the interval
14                       &f1           // pointer to function data
15                        );
16
17         /****
18          * Evaluate f(x) = 2 sin( 5 *x + 1 ) on <0,1>
19          */
20         Sinus f2;
21         f2.A = 2.0;
22         f2.f = 5.0;
23         f2.p = 1.0;
24         drawFunction( f2.getType(), // WE DO NOT NEED TO CARE ABOUT FUNCTION TYPE
25                       0.0, 1.0, // on interval <0, 1>
26                       0.1,      // we draw it with resolution 0.1
27                                 //    -> 11 points on the interval
28                       &f2       // pointer to function data
29                        );
30
31 }
```

Úvod do OOP

- takto je to výrazně jednodušší, stále ale ne úplně elegantní
- vidíme, že s oběma funkcemi navenek manipulujeme stejně, uvnitř mají pouze různou implementaci
- taková situace nastává při psaní kódu velice často
- v OOP ji řeší tzv. dědičnost
- nejprve si navrhneme objekt popisující obecnou funkci

Úvod do OOP

```
1 struct Function
2 {
3    virtual double getFx( double x );
4 }
```

- říkáme, že všechny naše funkce budou umět vrátit hodnotu v bodě *x* pomocí **metody** `getFx`
- slovo `virtual` znamená, že implementace této metody se bude pro různé funkce lišit

Úvod do OOP

Implementace funkcí pak může vypadat takto:

```
1  struct Quadratic : public Function  // !!!!
2  {
3     double a, b, c;
4
5     double getFx( double x )
6     {
7        return ( a * x + b ) * x + c;
8     }
9  }
10
11 struct Sinus : public Function   // !!!!
12 {
13    double A, f, p;
14
15    double getFx( double x )
16    {
17       fx = A * sin( f * x + p );
18    }
19 }
```

Všimněme si, že jsme zrušili řádek

```
1  enum Function { quadratic, sinus };
```

a dále jsme odstranili metody `getType()` z obou funkcí.

Úvod do OOP

Další důležitá změna je v deklaraci funkcí např.

```
1 struct Quadratic : public Function
```

- tím říkáme, že objekt `Quadratic` je tzv. **odvozen** od objektu `Function`
- znamená to, že `Quadratic` dědí stejné vlastnosti jako `Function`
- **nyní mohu s objety typu `Quadratic` a `Sinus` pracovat jako s objekty typu `Function` a překladač zařídí to, že budou správně rozpoznány**

Úvod do OOP

Funkce `drawFunction` nyní bude vypadat takto:

```
1  void drawFunction( Function* function,
2                     double left, double right, double step )
3  {
4      for( double x = left; x <= right; x += step )
5      {
6          printf( " %f %f \n", x, function->getFx( x ) );
7      }
8  }
```

- celou konstrukci s příkazy `switch .... case`, za nás vytvoří překladač jazyka C++
- tím nám šetří práci a zaručí, že nedojde k chybě programátora

Úvod do OOP

Volání funkce `drawFunction` se pak změní takto:

```
1  int main( int argc, char* argv[] )
2  {
3          /****
4           * Evaluate f(x) = 2 x^2 + 3 x + 1 on <0,1>
5           */
6           Quadratic f1;
7           f1.a = 2.0;
8           f1.b = 3.0;
9           f1.c = 1.0;
10          drawFunction( &f1, // WE DO NOT NEED TO CARE ABOUT FUNCTION TYPE
11                        0.0, 1.0,     // on interval <0, 1>
12                        0.1,          // we draw it with resolution 0.1
13                                      //    -> 11 points on the interval
14                         );
15
16          /****
17           * Evaluate f(x) = 2 sin( 5 *x + 1 ) on <0,1>
18           */
19          Sinus f2;
20          f2.A = 2.0;
21          f2.f = 5.0;
22          f2.p = 1.0;
23          drawFunction( &f2, // WE DO NOT NEED TO CARE ABOUT FUNCTION TYPE
24                        0.0, 1.0, // on interval <0, 1>
25                        0.1,      // we draw it with resolution 0.1
26                                  //    -> 11 points on the interval
27                         );
28
29 }
```

Úvod do OOP

Další velikou výhodou je to, že pokud nyní budu chtít do svého programu přidat další funkci, stačí pouze napsat její implementaci, ale kód funkce `drawFunction` již měnit nemusím!!!

Např.

```
1  struct Gauss : public Function
2  {
3     double sigma, mu;
4
5     double getFx( double x )
6     {
7        double aux = ( x - mu ) / sigma;
8        return 1.0 / ( sigma * sqrt( 2.0 * M_PI) ) *
9               exp( -0.5 * aux * aux );
10    }
11 }
```