C --> C++

Class vs. struct

- Slovo "class" bude slúžiť na definíciu tried.
- Pôvodný pohľad bol, že triedy budú rozšírením štruktúr o objektovo orientované črty.
- V C++ však všetky rozšírenia používané v triedach boli povolené aj pre štruktúry.
- Nakoniec štruktúry v C++ sú také triedy, kde štandardná viditeľnosť je "public", zatiaľ čo v triedach "private"

Atribúty viditeľnosti

```
class A {
  public:
    int i;
  private:
    int x;
};

void A::fun1() {
        i = 0;
        x = 0;
}

int main() {
   A a;
   a.i = 0;
   a.x = 0;       // chyba, nema pravo pristupovat
}
```

Class vs. struct: príklad

```
class A {
    int x, y;
};

struct B {
   int u,v;
};

int main() {
   A a; B b;
   a.x = 0;
   b.u = 1;
}
```

defaultná viditeľnosť "private"

defaultná viditeľnosť "public"

chyba!

Trieda generuje typedef

```
struct S { int a; };

struct S xx;              // C
S x;                      // C++
```

Trieda generuje typedef: Problém

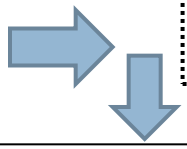
```
struct S { int a; };

int S;                          // ?

void f() {
      S = 1;                    // ?
}

void g(S x) {                   // ?
}
```

Trieda generuje typedef: Problém

```
struct S { int a; };

int S;                          // OK v C++

void f() {
      S = 1;                    // OK v C++
}

void g(S x) {                   // Chyba
}
```

Polymorfizmus

```
void fun(int a) {}
void fun(double a) {}

class A {
  void mfun(int a) {}
  void mfun(double a) {}
};

int main() {
   A a;
   fun(1);
   fun(1.0);
   a.mfun(5);
   a.mfun(5.0);
}
```



Problém pri kompilácii pre klasický linker sa rieši "manglovaním" mien:

Príklad: kompilácia do C

```
typedef struct A {} A;

void _Z3funi(int a) {}
void _Z3fund(double a){}
void _ZN1A4mfuni(A *cp, int a){}
void _ZN1A4mfund(A *cp, double a){}
int main() {
  A a;
 _Z3funi(1);
 _Z3fund(1.0);
 _ZN1A4mfuni(&a, 5);
 _ZN1A4mfund(&a, 5.0);
}
```

Skutočne sa mená manglujú?

```
#include <iostream>
using namespace std;

int fun(int i) {
  return(42);
}

extern "C" int _Z3funi();

int main() {
  cout << _Z3funi() << '\n';
}
```

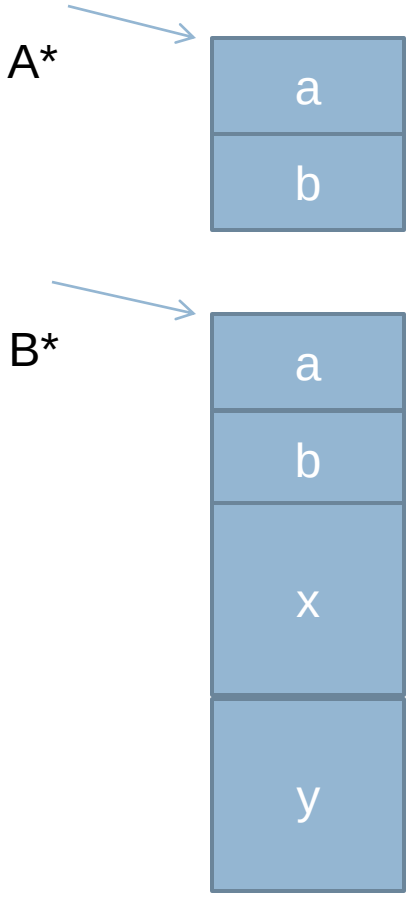
Odvodené triedy, dedenie, konverzie

kód:

```
class A {
      int a,b;
};

class B : public A {
      double x,y;
};
```

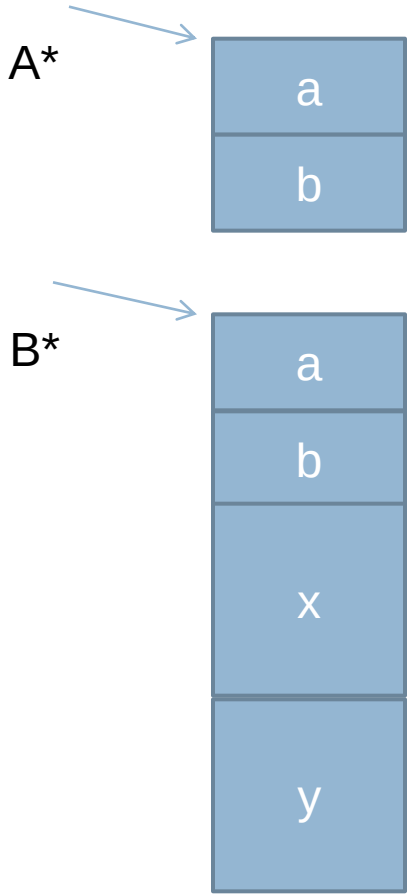
B rozširuje triedu A

pamäť:

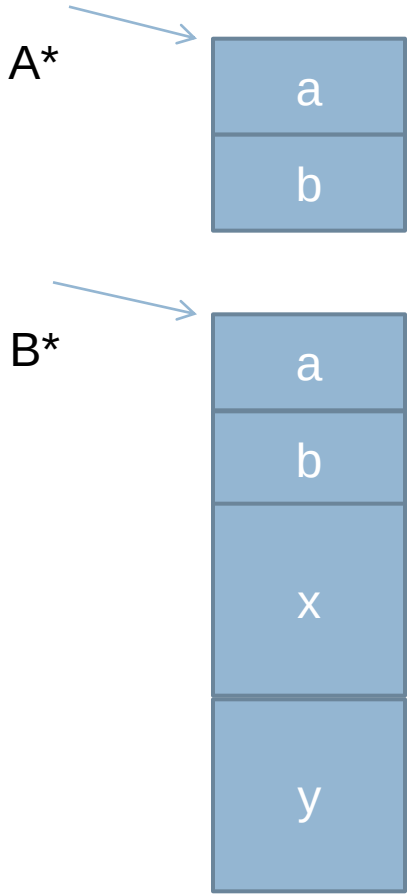


```
void funa(A *x) {...}
void funb(B *x) {...}

int main() {
  A *pa; B *pb;
  funa(pa);
  funb(pa);
  funa(pb);
  funb(pb);
  pa = pb;
  pb = pa;
}
```

Ktoré riadky sú v poriadku?



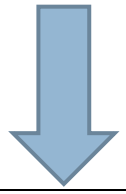
```
void funa(A *x) {...}
void funb(B *x) {...}

int main() {
  A *pa; B *pb;
  funa(pa);
  funb(pa);
  funa(pb);
  funb(pb);
  pa = pb;
  pb = pa;
}
```



Metódy, viditeľnosť členov a implicitný parameter

```
class Average {
  int sum;
  int n;
  void add(int x) {sum += x; n++; }
  int  getAverage() {return(sum / n);}
};

int main() {
  Average  a, *pa;
  pa = &a;
  a.add(1);
  a.getAverage();
  pa->add(2);;
  pa->getAverage();
}
```

```
class Average {
  int sum;
  int n;
  void add(int x) { sum += x; n++; }
  int getAverage() { return(sum / n); }
};
```



```
struct Average {
  int sum;
  int n;
};

void _ZN7Average3addi(struct Average *cp, int x) {
  cp->sum += x; cp->n ++;
}

int _ZN7Average10getAverage(struct Average *cp) {
  return(cp->sum / cp->n);
}
```

Metódy: Dedenie

```
struct A {
  void fun(double x) {
      printf("A::fun(double)");
  }
};

struct B : A {
};

int main() {
  B b;
  b.fun(3.14159);
}
```

Prekrývanie symbolov (metód)

- V prípade, že odvodená trieda definuje symbol s tým istým menom aký existuje v základnej triede, tento nebude ďalej viditeľný
- Platí to aj pre metódy a to tak, že všetky metódy zo základnej triedy prestanú byť viditeľné!

Prekrývanie metód: príklad

```
struct A {
  void fun(double x) {
      printf("A::fun(double)");
  }
};

struct B : A {
  void fun(int x) {
      printf("B::fun(int)");
  }
};

int main() {
  B b;
  b.fun(3.14159);
}
```

Prekrývanie metód: príklad

```
struct A {
  void fun(double x) {
      printf("A::fun(double)");
  }
};

struct B : A {
  void fun(int x) {
      printf("B::fun(int)");
  }
};

int main() {
  B b;
  b.fun(3.14159);
}
```

Vypíše: B::fun(int) aj ked lepšia korešpondencia typov je s A::fun(double).

Virtuálne metódy

```
class A {
  int a;
  fun() { printf("A"); }
};
class B : A {
  int b;
  fun() { printf("B"); }
};

int main() {
   A *pa;  B b;
   pa = &b;
   pa->fun();
}
```

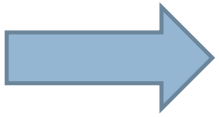
Vypíše:
A

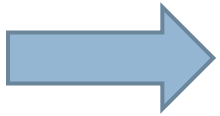
```
class A {
  int a;
  virtual fun() { printf("A"); }
};
class B : A {
  int b;
  virtual fun() { printf("B"); }
};

int main() {
   A *pa;  B b;
   pa = &b;
   pa->fun();
}
```

Vypíše:
B

Implementácia / kompilácia prvého programu

```
class A {
  int a;
  fun() { printf("A"); }
};
class B : A {
  int b;
  fun() { printf("B"); }
};

int main() {
   A *pa;  B b;
   pa = &b;
   pa->fun();
}
```

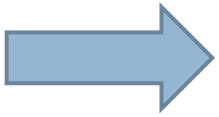


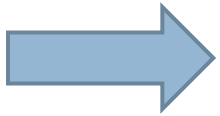


```
struct A {
  int a;
};
struct B {
  int a;
  int b;
};

_ZN1A3fun(struct A*p) {printf("A");}
_ZN1B3fun(struct B*p) {printf("B");}

int main() {
   struct A *pa;  struct B b;
   pa = (struct A*) &b;
   _ZN1A3fun(pa);
}
```

Implementácia / kompilácia druhého programu

```
class A {
public:
  int a;
  virtual fun() { printf("A"); }
};
class B : A {
public:
 int b;
 virtual fun() { printf("B"); }
};

int main() {
   A *pa;  B b;
   pa = &b;
   pa->fun();
}
```

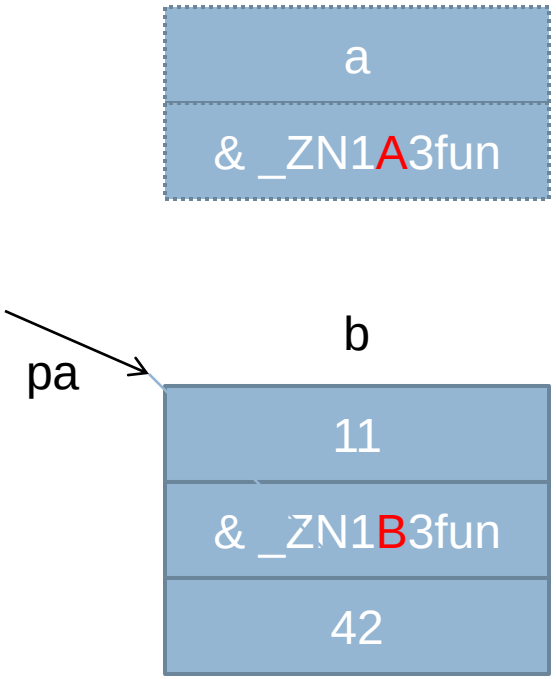




```
struct A {
  int a;
};
struct B {
  int a;
  int b;
};


_ZN1A3fun(struct A*p) {printf("A");}
_ZN1B3fun(struct B*p) {printf("B");}


int main() {
   struct A *pa;  struct B b;
   pa = (struct A*) &b;
   ?????(pa);
}
```

Virtuálne metódy: implementácia0

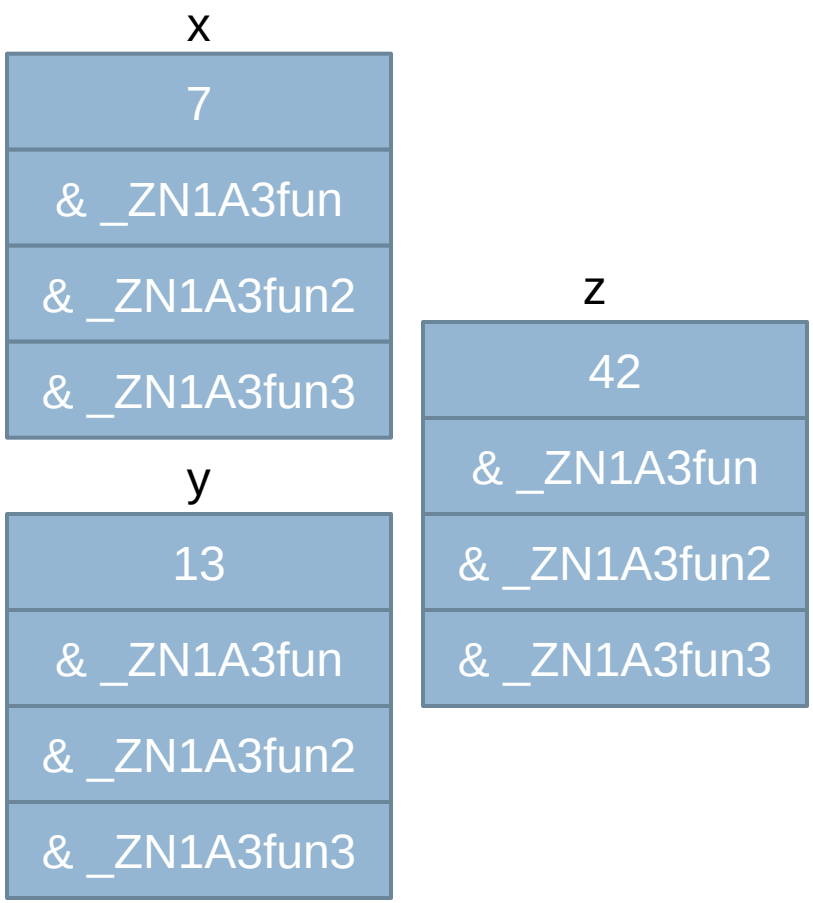
```
struct A {
  int a;
  void (*fun)();
};
struct B {
  int a;
  void (*fun)();
  int b;
};
_ZN1A3fun(struct A*p) {printf("A");}
_ZN1B3fun(struct B*p) {printf("B");}

int main() {
   struct A *pa;  struct B b;
   b.a = 11;  b.b = 42;
   pa = (struct A*) &b;
   (*pa->fun)(pa);
}
```



Pri konštrukcii objektu b sa do b.fun priradí **_ZN1B3fun**

Implementácia1 nevýhoda

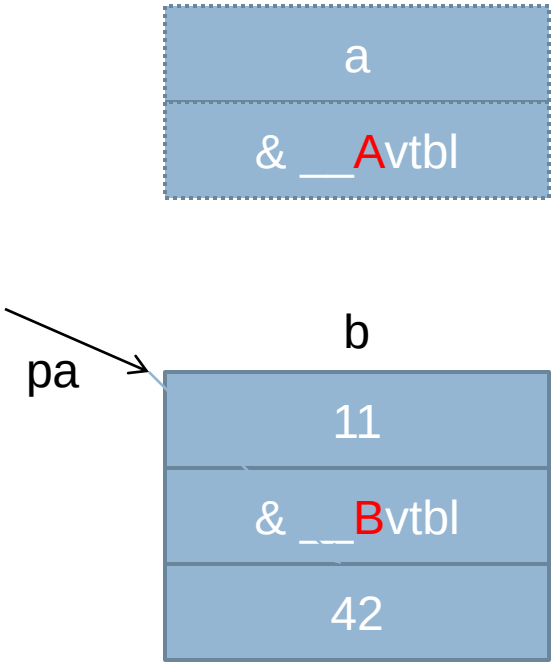
```
struct A {
  int a;
  virtual void fun() {...}
  virtual void fun2() {...}
  virtual void fun3() {...}
};

int main() {
   A x,y,z;
   x.a = 7; y.a = 13; z.a = 42;
}
```



V každom objekte by pribudli 4 byte-y pre každú pridanú virtuálnu metódu

Virtuálne metódy: implementácia

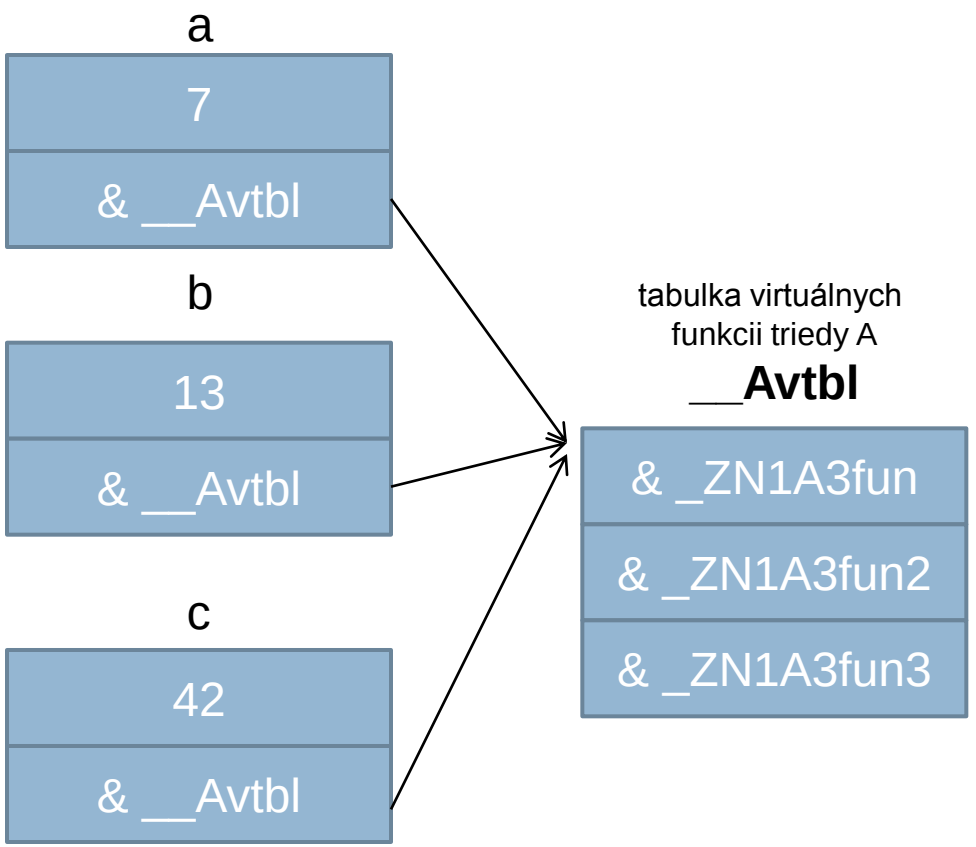
```
struct A {
  int a;
  void (**vtbl);
};
struct B {
  int a;
  void (**vtbl)();
  int b;
};
_ZN1A3fun(struct A*p) {printf("A");}
_ZN1B3fun(struct B*p) {printf("B");}
void *__Avtbl[] = {& _ZN1A3fun};
void *__Bvtbl[] = {& _ZN1B3fun};
int main() {
    struct A *pa;  struct B b;
    b.a = 11;  b.b = 42;
    pa = (struct A*) &b;
    (*pa->vtbl[0])(pa);
}
```



Pri konštrukcii objektu b sa do b.vtbl priradí **__Bvtbl** (tabulka virtuálnych funkcií)

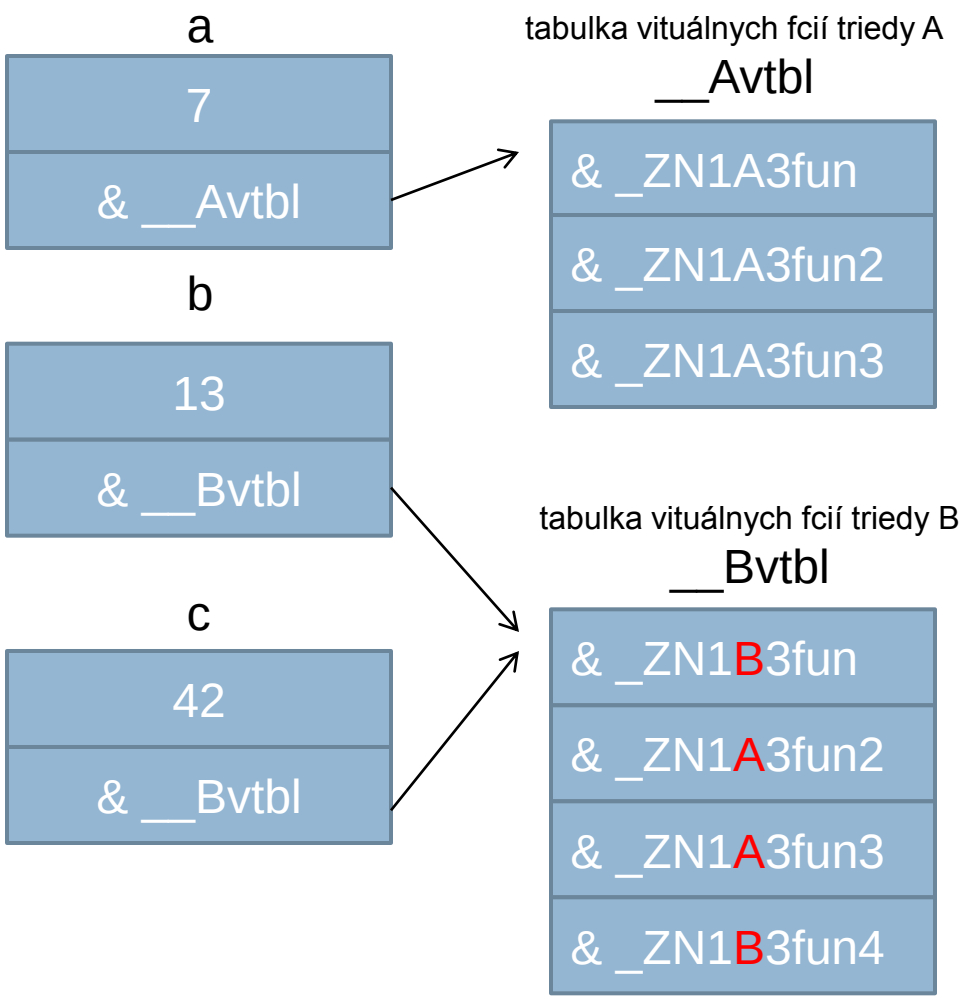
Tabulka virtuálnych funkcií: príklad1

```
class A {
public:
  int x;
  virtual void fun() {...}
  virtual void fun2() {...}
  virtual void fun3() {...}
};

int main() {
   A a,b,c;
   a.x = 7;
   b.x = 13;
   c.x = 42;
}
```



Tabulka virtuálnych funkcií: príklad2

```
class A {
public:
   int x;
   virtual void fun() {...}
   virtual void fun2() {...}
   virtual void fun3() {...}
};
class B : A {
public:
   virtual void fun() {...}
   virtual void fun4() {...}
}

int main() {
    A a; B b c;
    a.x = 7;
    b.x = 13; c.x = 42;
}
```



Virtuálne metódy: načo je to dobré?

- Dynamická náhrada za switch
- Umožňuje meniť beh už skompilovaných knižníc (callback)
- Generické programovanie

Náhrada za switch: Príklad

```
struct Komunikator {
  int typSpojenia;
  // ak typSpojenia == 0 --> file
  FILE *ff;
  // ak typSpojenia == 1 --> socket
  int socket;
  // ak typSpojenia == 2 --> shared memory
  char *p;
  ...
  void posliZnak(char c) {
       switch (kod) {
       case 0: fputc(c, ff); break;
       case 1: send(socket, &c, 1, 0); break;
       case 2: *p = c; p ++; break;
       }
  }
  ...
};
```

```
class Komunikator {
  ...
  virtual void posliZnak(char c);
};

class KomunikatorFile : Komunikator {
  FILE *ff;
  ...
  virtual void posliZnak(char c) {fputc(c, ff);}
};

class KomunikatorSocket : Komunikator {
  int socket;
  ...
  virtual void posliZnak(char c) {send(socket, &c, 1, 0);}
};

class KomunikatorSharedMemory : Komunikator {
  char *p;
  ...
  virtual void posliZnak(char c) {*p = c; p++; }
};
```

Callback z kompilovanej knižnice: príklad

```
class AssyncIO {                          // skompilovana kniznica
  ...
  virtual void onConnect()     {}
  virtual void onRead(char c)  {}
  virtual void onWrite(...)    {}
  virtual void onError()       {}
};

.....................................

class MojeAssyncIO : AssyncIO {           // moj kod
  ...
  virtual void onConnect()    {...}
  virtual void onRead(char c){...}
  virtual void onWrite()      {...}
  virtual void onError()      {...}
};
```

Knižnica volá svoje prázdne virtuálne procedúry pri každom výskyte udalosti

Ak odvodím svoju triedu od knižničnej, tak sa pri každej udalosti vyvolá moja metóda a ja môžem udalosť spracovať

Generické programovanie

```
class Ordered {
  virtual int lessThan(Ordered y);
};

void bubleSort(Ordered a[], int size) {
  int i, sorted;
  sorted = 0;
  while (! sorted) {
      sorted = 1;
      for(i=0; i<size-1; i++) {
        if (a[i+1].lessThan(a[i])) {
            t = a[i]; a[i] = a[i+1]; a[i+1] = t;
            sorted = 0;
        }
      }
  }
}
```

Prečo používať s mierou?

- Volanie virtuálnej metódy je pomalšie ako volanie nevirtuálnej metódy / funkcie.
- Sťažuje porozumenie existujúceho kódu, pretože nie je jasné ktorá metóda sa na danom mieste vlastne zavolá.

Scope (obzor, pôsobnosť, ...)

- Symboly definované v "scope" sú v jeho rámci viditeľné bez prefixu, ale mimo nie sú viditeľné vôbec.
- { ... } v kóde vytvárajú "scope".
- Trieda v C++ vytvára "scope" (na rozdiel od C).

Scope (obzor, pôsobnosť, ...)

Nasledovný kód je platný v C ale nie v C++ !

```
struct A {
  struct B {
     int b1, b2;
  } x;
  int a1, a2;
};

int main() {
  struct A a;
  struct B b;
  b = a.x;
  b.b1 = 0;
}
```

Post-definície obnovujú scope triedy

```
int i;

class A {
  int i, j;
  void fun();
};

int j;

void A::fun() {
  i = 0;       // A.i, nie globalne i
  j = 0;       // A.j, nie globalne j
}
```

Reference type

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

int main() {
   int a, b, c;
   int &r = a;
   a = 5;
   r = 6;
   printf("%d ", a);
   b = 2;
   c = mul(b,b);
   printf("%d ", c);
}
```

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

Vypíše: 6 9

Default parameter

```
int fun(int i, int j=10) {
  return(i * j);
}

int main() {
  int a, b;
  a = fun(2,3);
  b = fun(2);
}
```

a == 6

b == 20

Konštantné premenné, konštanty

```
const double pi = 3.14159;

const int velkost_tuto = 1000;

int tuto[velkost_tuto];

const char *p;
char *const q;


int main() {
   p = "toto";
   q = p;
   p = q;
}
```

V C++ možno konštantnú premennú použiť v konštantných výrazoch

Pointer na nemenný char

Nemenný pointer na char

Chyba