Wydział Elektroniki
i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Programowanie obiektowe
Dziedziczenie i polimorfizm

Krzysztof Gracki
kgr@ii.pw.edu.pl

pok. 312

Politechnika
Warszawska

Temat

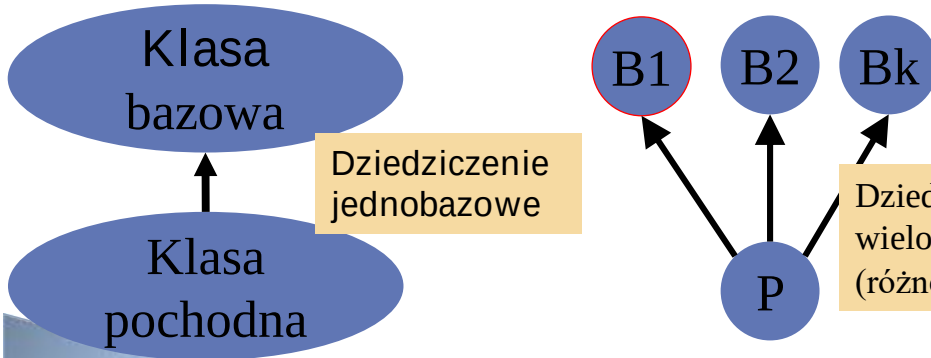
- Dziedziczenie: zapis, terminologia, zasięgi
- Dziedziczenie wirtualne
- Dostęp do składowych
- Dziedziczenie i zawieranie
- Referencje i wskazania
- Funkcje wirtualne i polimorfizm
- Funkcje wirtualne czyste i klasy abstrakcyjne
- Realizacja funkcji wirtualnych
- Polimorfizm i kowariancja typów
- Kolejność konstrukcji / destrukcji obiektów
- Funkcje wirtualne i ochrona dostępu
- Funkcje wirtualne w interfejsie niewirtualnym

Mechanizm dziedziczenia

- Klasy autonomiczne => abstrakcyjne typy danych (rozszerzanie języka)
- Dziedziczenie: relacja pomiędzy klasami i obiektami wyrażająca przejęcie cech; dziedziczenie w C++ jest totalne
- Obiekt dziedziczący można traktować jako szczególny przypadek obiektu z którego dziedziczy - posiada wszystkie cechy tego obiektu, być może uzupełnione nowymi cechami
- Polimorfizm: ściśle związany z dziedziczeniem - obiekty należące na pewnym poziomie abstrakcji do wspólnej klasy mogą mieć odmienne cechy na poziomie szczegółowym
- Dziedziczenie jest podstawą tworzenia klasyfikacji hierarchicznej obiektów; dobrze zdefiniowane hierarchie pozwalają uprościć strukturę złożonych systemów

Dziedziczenie - zapis


```
wyróżnik KlasaPochodna: lista_klas_bazowych
   |    { lista_składowych } deklaratory_opt ;
   |
   ├── class
   └── struct ( ~~union~~ )
```



Dziedziczenie - terminologia

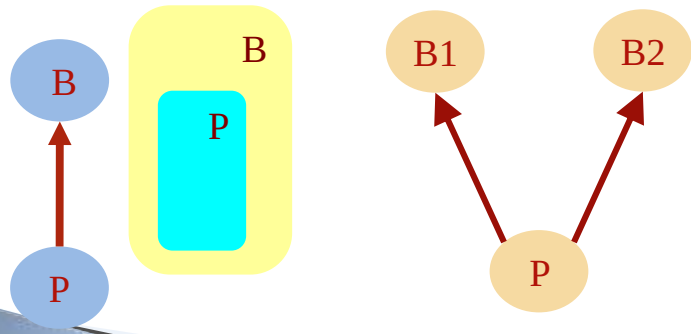

```
class EDokument: public Plik
{ string autor;
  // ...
public:
  // Składowe interfejsu publicznego
};
```

Kwalifikator dziedziczenia

- EDokument jest **klasą pochodną** względem Pliku
- EDokument jest szczególnym przypadkiem Pliku
- EDokument jest specjalizacją Pliku
- EDokument jest podklasą Pliku
- Plik jest **klasą bazową** EDokumentu
- Plik jest nadklasą EDokumentu

Dziedziczenie i zasięgi

- Klasa bazowa B stanowi zasięg otaczający dla klasy pochodnej P. Dostęp do nazwy przysłoniętej jest możliwy po zastosowaniu operatora '`::`' (`B::nazwa_przyslonieta`)
- Dla dziedziczenia wielobazowego powyższą regułę stosuje się odpowiednio do każdej bazy




```
struct B1
{ int i;
  B1() { i=1; }
};
struct B2
{ int i;
  B2() { i=2; }
};
struct P: B1, B2
{ int i;
  P(){ i=B1::i+B2::i;}
};
```

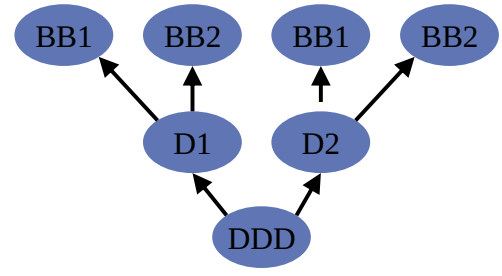
Przesłanianie metod


```
struct B {
   int fun(int i) { return 0; };
};

struct D : public B {
   int gun(string s) { return 1; };
   int fun(string s) {  // nazwa jak
      return 2; };      // w klasie bazowej
   }

int main()  {
  D d;
  cout << d.fun("OK");
  cout << d.fun(10);  // ERROR
  cout << d.B::fun(10) << d.D::fun("s");
}
```

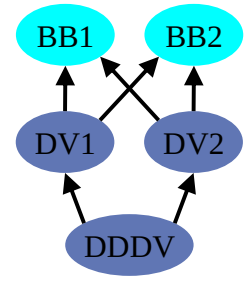
Przesłanianie nazw
(nie sygnatur funkcji)

Dziedziczenie wirtualne

```
struct BB1 { double i1; };
struct BB2 { double i2; };

struct D1: BB1, BB2 { int i3; };
struct D2: BB1, BB2 { int i4; };
struct DDD: D1, D2  { int i5; };
```




```
struct DV1: virtual BB1, virtual BB2
{ int i3; };
struct DV2: virtual BB1, virtual BB2
{ int i4; };
struct DDDV: DV1, DV2
{ int i5; };
```



Obiekty i podobiekty

- Składowe przejęte z klasy bazowej do klasy pochodnej tworzą (anonimowy) podobiekt;
- W dziedziczeniu wielobazowym może pojawić się wiele podobiektów pochodzących od tej samej klasy bazowej (p. klasy B1, B2 w poprzednim przykładzie)
- Dziedziczenie z bazami wirtualnymi gwarantuje obecność jednego podobiektu
- Rozmieszczenie podobiektów w obiekcie pełnym należy do kompetencji kompilatora (można programowo rozpoznać to rozmieszczenie - rzadko jest to potrzebne)

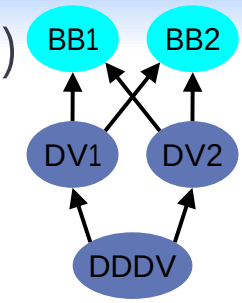
Obiekty i podobiekty (cd1)


```
struct BB1 { double i1; };
struct BB2 { double i2; };
struct D1: BB1, BB2 { int i3; };
struct D2: BB1, BB2 { int i4; };
struct DDD: D1, D2  { int i5;
friend
  ostream& operator<<(ostream& os,const
DDD& x)
  { char *p   = (char*)&x;
    char *p11 = (char*)(&x.D1::i1);
    char *p12 = (char*)(&x.D1::i2);
    char *p21 = (char*)(&x.D2::i1);
    char *p22 = (char*)(&x.D2::i2);
    char *p3  = (char*)(&x.i3);
    char *p4  = (char*)(&x.i4);
    char *p5  = (char*)(&x.i5);
```

Przykład (w środowisku Visual Studio) dla poprzedniej hierarchii klas pokazuje typowe konwencje kompilatorów.

```
os <<"DDD layout\n"
   <<"   start: "<<(void*)(p)<<'\n'
   <<"  D1::i1: "<<(p11-p) <<'\n'
   <<"  D1::i2: "<<(p12-p) <<'\n'
   <<"      i3: "<<(p3-p)  <<'\n'
   <<"  D2::i1: "<<(p21-p) <<'\n'
   <<"  D2::i2: "<<(p22-p) <<'\n'
   <<"      i4: "<<(p4-p)  <<'\n'
   <<"      i5: "<<(p5-p)  <<'\n'
   << endl;
 return os;
 }
```

Obiekty i podobiekty (cd2)

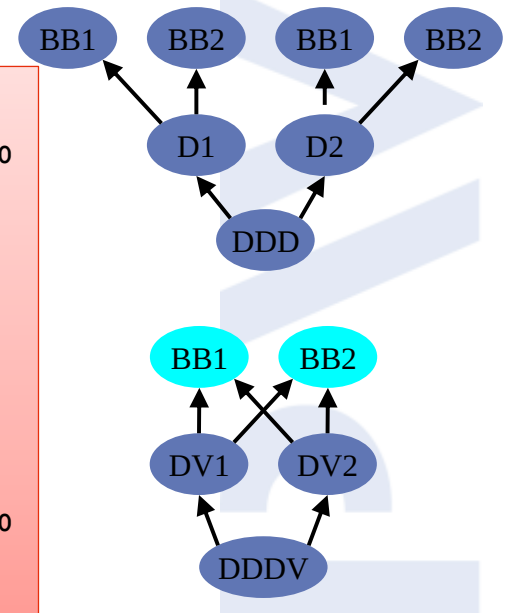



```
struct DV1: virtual BB1, virtual BB2
    { int i3; };
struct DV2: virtual BB1, virtual BB2
    { int i4; };
struct DDDV: DV1, DV2
    { int i5;
friend
  ostream& operator<<(ostream& os,const DDDV& x)
  {

    char *p  = (char*) &x;
    char *p1 = (char*)(&x.i1);
    char *p2 = (char*)(&x.i2);
    char *p3 = (char*)(&x.i3);
    char *p4 = (char*)(&x.i4);
    char *p5 = (char*)(&x.i5);

                                os <<"DDDV layout\n"
                                   <<"  start: "<<(void*)(p) <<'\n,
                                   <<"     i1: "<<(p1-p) <<'\n'
                                   <<"      i2: "<<(p2-p) <<'\n'
                                   <<"      i3: "<<(p3-p)  <<'\n'
                                   <<"      i4: "<<(p4-p)  <<'\n'
                                   <<"      i5: "<<(p5-p)  <<'\n'
                                   << endl;
                                return os;
                               }};
```


Obiekty i podobiekty (cd3)


```
#define PRN(x) cout<<#x" = "<<(x)<<endl

int main()
{
  DDD  ddd;
  DDDV dddv;
  PRN(sizeof(ddd));
  cout<<ddd;
  PRN(sizeof(dddv));
  cout<<dddv;

return 0;
}
```

```
sizeof(ddd) = 56
DDD layout
   start: 003FFAA0
  D1::i1: 0
  D1::i2: 8
      i3: 16
  D2::i1: 24
  D2::i2: 32
      i4: 40
      i5: 48

sizeof(dddv) = 40
DDDV layout
   start: 003FFA70
      i1: 24
      i2: 32
      i3: 4
      i4: 12
      i5: 16
```



Dostęp do składowych

- Zapis odwołań do składowych odziedziczonych: jak do zwykłych składowych
- Uprawnienia dostępu: wynik złożenia kwalifikatorów dostępu w dziedziczeniu i w klasie bazowej
- Można regulować dostępność indywidualnie dla składowych przy pomocy deklaracji **`using`** (ale bez naruszenia uprawnień z klasy bazowej)

```
#include <iostream>
class B {
  protected : int m;
  // B::m jest protected
};

class D : B {
  public:
  using B::m;
  // Skł. D::m jest public
};
```

Dostępność składowej z klasy bazowej w klasie pochodnej		Kwalifikator dziedziczenia		
		public	protected	private
Kwalifikator dostępu w klasie bazowej	public	public	protected	private
	protected	protected	protected	private
	private	niedostępna	niedostępna	niedostępna

Dziedziczenie i zawieranie

- Klasa zawiera obiekt innej klasy, jeżeli ma zadeklarowaną składową tej klasy
- Klasa dziedziczy podobiekt innej klasy, jeżeli jest względem niej pochodną
- Możliwe jest odziedziczenie podobiektu w którym są zawarte pewne obiekty jako składowe
- Wybór dziedziczenia albo zawierania wynika z roli pełnionej przez klasę względem innej klasy:
  - jeżeli uprawnione jest traktowanie obiektów pochodnych jako szczególnych przypadków obiektów bazowych → **dziedziczenie**
  - jeżeli powyższe jest nieuprawnione, albo obiekt musi posiadać kilka instancji innego → **zawieranie**

Tylko dziedziczenie może być podstawą zachowania polimorficznego klasy

Dziedziczenie i zawieranie (cd1)


```
class Punkt
{ double x, y;
public:
  Punkt(double xx=0, double yy=0): x(xx), y(yy) {}
  friend ostream& operator<<(ostream& os,const Punkt& p)
  { return os << '(' << p.x<< ',' << p.y << ')'; }
  // ...
};
class Odcinek {
protected:
  Punkt p1, p2;
public:
  Odcinek(const Punkt& a=0,const Punkt& b=0)
     : p1(a), p2(b) {}
  double miara() const;
  friend ostream& operator<<(ostream& os,const Odcinek& l)
     { return os << '[' << l.p1<< ',' << l.p2 << ']'; }
  // ...
};
```

Odcinek zawiera dwa Punkty
(ma punkt poczatkowy i końcowy)

Dziedziczenie i zawieranie (cd2)


```
class Wektor: public Odcinek
{
public:
  Wektor(){}  // Równoważne: Wektor(): Odcinek(){}
  Wektor(const Punkt& dop): Odcinek(0, dop) {}
  Wektor(const Punkt& a, const Punkt& b): Odcinek(a,b){}
  Wektor(const Odcinek& d): Odcinek(d) {}
  // ...
};

int main()
{ Punkt A(1,2), B(3,4);
  Odcinek s(A, B);
  Wektor v(B);
  Wektor w(s);
  cout<<s<<endl; cout<<v<<endl;
  cout<<w<<endl; cout<<Wektor(B,0)<<endl;
  return 0;
}
```

Wektor jest odmianą Odcinka

```
[(1,2),(3,4)]
[(0,0),(3,4)]
[(1,2),(3,4)]
[(3,4),(0,0)]
```

Referencje i wskazania

- Wskazanie na obiekt klasy pochodnej może zawsze być przekształcone na wskazanie na **publiczną** klasę bazową

```
class B { /* ... */ };
class D: public B { /* ... */ };
D *dptr = new D();
B *bptr = dptr;  // Konwersja standardowa
B &bref = *dptr; // bref jest pseudonimem *dptr
                 // (podobiektu B w obiekcie D)
```

- Konwersja w drugą stronę wymaga zastosowania operatora konwersji `static_cast` albo `dynamic_cast` dla typów polimorficznych;
  poprawność konwersji `static_cast` - na odpowiedzialność programisty

```
D d;
B *bp = &d;    // Konwersja standardowa
D *dp = static_cast<D*>(bp);// Potrzebny operator
D &dr = static_cast<D&>(*bp);
```

Funkcje wirtualne i polimorfizm

- Funkcje składowe niestatyczne poprzedzone kwalifikatorem `virtual` są nazywane funkcjami wirtualnymi (polimorficznymi)
- Mechanizm funkcji wirtualnych:
  - Funkcja zadeklarowana jako `virtual` w klasie bazowej pozostaje wirtualną w klasie pochodnej pod warunkiem zachowania tożsamego prototypu (ta sama: nazwa, typ wyniku, lista parametrów); kwalifikator `virtual` w klasie pochodnej nie jest niezbędny, ale można go powtórzyć
  - Wywołanie funkcji wirtualnej poprzez wskazanie albo referencję powoduje aktywację wersji zdefiniowanej w klasie bazowej albo w klasie pochodnej zgodnie z typem pełnego obiektu wskazywanego
  - Rozstrzygnięcie, którą wersję aktywować może nastąpić dopiero w czasie wykonania programu (polimorfizm dynamiczny)
- Przeciążanie funkcji można uważać za polimorfizm statyczny

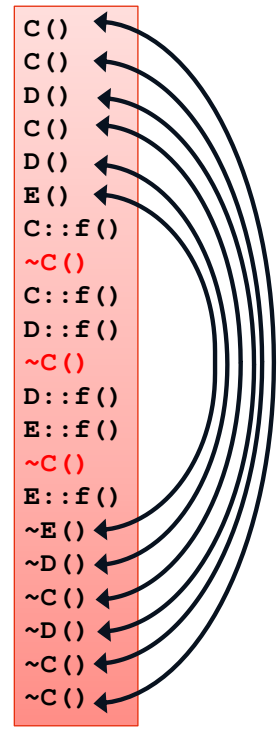
Politechnika Warszawska

Funkcje wirtualne i polimorfizm (cd1)


```
#define P(x) x(){ cout<<#x"()\n";}
struct C
{ P(C); P(~C);
  virtual void f()
  { cout << "C::f()\n"; }
};
struct D : C
{ P(D); P(~D);
  void f()
  { cout << "D::f()\n"; }
};
struct E : D
{ P(E); P(~E);
  void f()
  { cout << "E::f()\n"; }
};
C  fun(C *p) { p->f(); return *p; }
C& gun(C &r) { r.f();  return r;  }
```

Funkcje wirtualne i polimorfizm (cd2)


```
int main()
{ C c, *cp=&c;
  D d, *dp=&d;
  E e, *ep=&e;
  fun(cp); gun(*cp);
  fun(dp); gun(*dp);
  fun(ep); gun(*ep);
}
```



Wirtualny destruktor - problem


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

struct Buf
{ int n;
  char *p;
  Buf(int nn=1024)
    :n(nn), p(new char[n])
  { cout<<"Alokowano "<<n<<"B\n"; }

 // virtual   <=
 ~Buf()  {
    delete [] p;
    cout<<"Zwolniono "<<n<<"B\n";
  }
};
```

```
struct BufBuf: Buf
{ int m;
  char *q;
  BufBuf(int mm=2048)
         :m(mm), q(new char[m]) {
    cout<<"Alokowano "<<m<< "B\n"; }
 ~BufBuf()  {
    delete [] q;
    cout<<"Zwolniono "<<m<<"B\n";
  }
};

int main()
{
   Buf *bptr = new BufBuf;
   delete bptr;
 }
```

Bez virtual

```
Alokowano 1024B
Alokowano 2048B
Zwolniono 1024B
```

Z virtual

```
Alokowano 1024B
Alokowano 2048B
Zwolniono 2048B
Zwolniono 1024B
```

Politechnika Warszawska

Wirtualny destruktor (cd)

- "Wirtualzacja" destruktorów jest niezbędna, gdy możliwa jest destrukcja obiektu klasy pochodnej poprzez wskazanie albo referencję na klasę bazową (destrukcja polimorficzna).
- Mechanizm rozpoznawania funkcji wirtualnych wymaga uzupełnienia konwencji – destruktory w klasach pochodnych mają inne nazwy niż destruktor klasy bazowej, zatem tożsamość prototypów nie wchodzi w grę.
- Przyjmuje się jako zasadę, że klasa bazowa posiadająca funkcję wirtualną powinna także posiadać destruktor wirtualny; gwarantuje to polimorfizm destruktorów w głąb hierarchii.
- Nie istnieje symetryczne pojęcie konstruktora wirtualnego (polimorfizm destruktora wynika z typu obiektu niszczonego; co miałoby być podstawą polimorfizmu konstruktora?).
- Można jednak zdefiniować polimorficzne funkcje klonowania obiektów lub tworzenia wariantów istniejących obiektów.

Politechnika Warszawska

Funkcje wirtualne czyste i klasy abstrakcyjne

- Funkcja wirtualna czysta: funkcja niestatyczna z deklaratorem postaci:

  ```
  virtual wynik fun( parametry ) = 0;
  ```

- Klasa abstrakcyjna: klasa z funkcją wirtualną czystą
- Klasa abstrakcyjna specyfikuje (poprzez swoje funkcje wirtualne) **interfejs publiczny** całej hierarchii klas
- Wg klasy abstrakcyjnej nie można tworzyć obiektów; można natomiast wyprowadzać z niej klasy konkretne
- **Uwaga**:
  Każda klasa wykorzystywana jako baza hierarchii polimorficznej powinna posiadać **destruktor wirtualny**; w szczególności destruktor wirtualny czysty powinien być zdefiniowany

Funkcje wirtualne czyste i klasy abstrakcyjne


```
struct Figura
{ virtual void show() = 0;
  // Funkcja wirtualna czysta
  virtual ~Figura() = 0 {}
  // Destruktor wirtualny
};
struct Polygon: public Figura
{ void show() override
  { cout<< "Polygon::show()\n"; }
};
struct Histogram: public Figura
{ void show() override
  { cout<< "Histogram::show()\n"; }
};
struct Bezier: public Figura
{ void show() override
  { cout<< "Bezier::show()\n"; }
};
```

```
// Do kompilacji 'show'
// potrzebna tylko klasa Figura

void show(Figura *fp[])
{ int i;
  for(i=0; fp[i] !=0; ++i)
    fp[i] -> show();
  cout<<"Liczba figur: "<<i<<endl;
}

int main()
{ Histogram h;
  Polygon p;
  Bezier b;

  Figura* fp[]={ &p,&b,&h,&p,0 };

  show(fp);
  return 0;
}
```

```
Polygon::show()
Bezier::show()
Histogram::show()
Polygon::show()
Liczba figur: 4
```

Ostrzeżenie!

Obowiązujące w C++ identyczne traktowanie wskazań i tablic w parametryzacji funkcji może być źródłem niespodzianek:

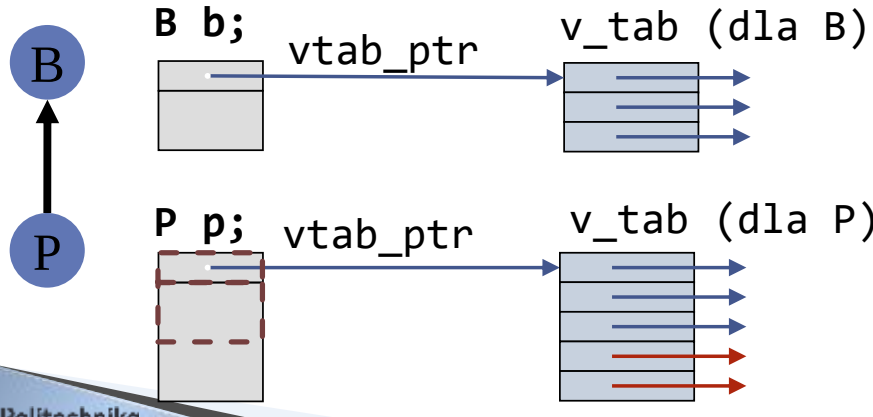
```
struct Figura// To nie jest klasa abstrakcyjna
{ virtual void show() { cout<< "Figura::show()\n"; }
};

// ... Pozostałe klasy dziedziczące

void show(Figura *f)  // Tablica Figur?
{ f->show();
}
int main2()
{ Histogram h;
  Polygon p;
  Bezier b;
  Figura f[] = { h, p, b };
  show(f);
  return 0;
}
```

Gdzie podział się polimorfizm?

`Figura::show()`

Realizacja funkcji wirtualnych

Do realizacji mechanizmu funkcji wirtualnych wystarcza tabela wskazań na funkcje dla każdej klasy w hierarchii polimorficznej i wskazanie na tę tabelę w każdym obiekcie polimorficznym



Dziedziczenie i interfejsy

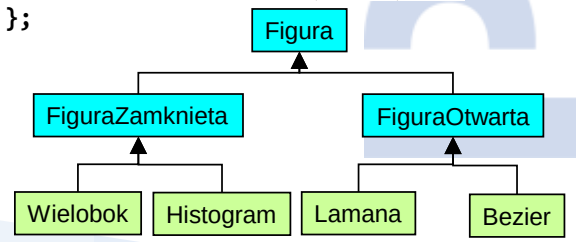
- Wszystkie akcesoria z kwalifikacją dostępu **public** stanowią podstawowy interfejs publiczny klasy – dostępny dla wszystkich obiektów i funkcji (jeśli spełnione są warunki widzialności zasięgu); do interfejsu publicznego w szerszym sensie dołącza się także funkcje zaprzyjaźnione i inne funkcje zewnętrzne korzystające a podstawowego interfejsu publicznego.
- Akcesoria z kwalifikacją dostępu **protected** stanowią interfejs chroniony klasy – dostępny dla składowych i zaprzyjaźnień klasy oraz dla klas pochodnych. Interfejs chroniony jest wykorzystywany do propagacji szczegółów implementacyjnych w głąb hierarchii klas.
- Akcesoria z kwalifikacją dostępu **private** stanowią ukryty dla otoczenia zestaw konwencji i usług implementacyjnych klasy.

Istotne kryterium projektowe: traktować interfejs publiczny jako **stabilny** kontrakt użytkowy.


```
struct Figura // Klasa abstrakcyjna
{ virtual void rysuj()=0;
  virtual Figura* klonuj()=0;
  virtual ~Figura()=0 {}
};
// Pokaz kolekcji figur
void rysuj(Figura *f[])
{ for(int i=0; f[i]; ++i)
    f[i]->rysuj();
}

struct FiguraZamknieta: Figura { };

struct Wielobok: FiguraZamknieta
{ void rysuj();

  Figura* klonuj()
  { return new Wielobok(*this); }
};

struct Histogram: FiguraZamknieta
{ void rysuj();

  FiguraZamknieta* klonuj()
  { return new Histogram(*this); }
};
```

Polimorfizm i kowariancja typów


```
struct FiguraOtwarta: Figura { };

struct Lamana: FiguraOtwarta
{ void rysuj();

  Lamana* klonuj()
  { return new Lamana(*this); }
};

struct Bezier: FiguraOtwarta
{ void rysuj();

  Figura* klonuj()
  { return new Bezier(*this); }
};
```



Polimorfizm i kowariancja typów (cd1)


```
int main()
{ Figura* f1 = new Histogram;
  Figura* f2 = new Bezier;
  Figura* f3 = f1->klonuj();
  Figura* f4 = f2->klonuj();
  Figura*fp[]= {f1, f2, f3, f4, 0};
rysuj(fp); }
```

```
Histogram::rysuj()
Bezier::rysuj()
Histogram::rysuj()
Bezier::rysuj()
```

Relacja kowariantności / kontrawariantności między typami

- W C++ odnosi się do **wskazań albo referencji** w strukturach dziedziczenia
- Konwersja kowariantna (w górę hierarchii): wykorzystywana w specyfikacji typów zwracanych przez funkcje wirtualne w klasach pochodnych (np. konwersja **`Bezier*`** ⇒ **`Figura*`** w funkcji wirtualnej **`Bezier::klonuj()`**).
- Konwersja kontrawariantna (w głąb hierarchii): stosowana w parametryzacji funkcji z efektami polimorficznymi (np. konwersja **`Figura*`** ⇒ **`Bezier*`** w funkcji rysuj(**`Figura*f[]`**)).

Ogólnie: funkcja wirtualna **`P::vf()`** w klasie pochodnej **P** zwraca typ kowariantny **T*** lub **T&** względem tejże funkcji w klasie bazowej **B**, jeżeli spełnione są warunki:

- **`B::vf()`** zwraca **BT*** lub **BT&**, gdzie **BT** jest bezpośrednią lub pośrednią klasą bazową **T**.
- Kwalifikacje **`const`** i **`volatile`** w typie zwracanym przez **`P::vf()`** są takie same lub słabsze niż w typie zwracanym przez **`B::vf`**().

Kolejność konstrukcji / destrukcji obiektów

Czynniki wpływające na porządek konstruowania obiektów:

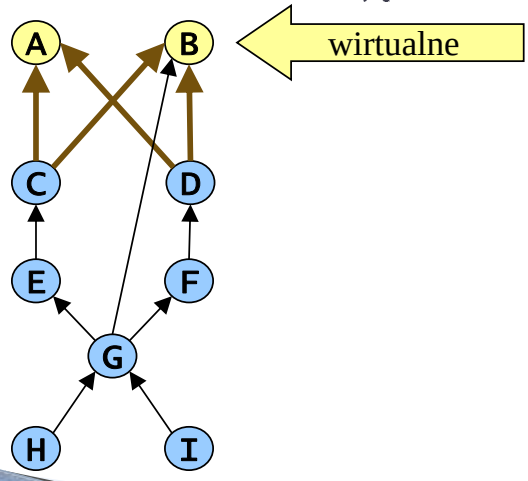
- Kategoria żywotności obiektów
  - **statyczne**: konstruowane przed aktywacją funkcji main() wg porządku definiowania
  - **lokalne**: konstruowane w kolejności definiowania w swoim zasięgu
  - **tymczasowe**: wg potrzeb w kolejności ewaluacji wyrażeń
  - **dynamiczne**: wg akcji alokacji / dealokacji podejmowanych w programie
- Osadzenie obiektu jako podobiektu
  - podobiekt implikowany relacją dziedziczenia: konstruowany wg porządku dziedziczenia, ale z pierwszeństwem dla dziedziczenia wirtualnego
  - podobiekt zawarty jako składowa: po skonstruowaniu podobiektów wg dziedziczenia konstruowane są składowe w kolejności zdefiniowania

Destrukcja obiektów w porządku odwrotnym do konstrukcji (z wyłączeniem destrukcji obiektów dynamicznych)

W odniesieniu do konstrukcji pojedynczego obiektu o znanej strukturze dziedziczenia obowiązuje reguła priorytetowa: klasy wirtualne ⇒ w głąb grafu dziedziczenia ⇒ w ramach poziomu od lewej do prawej ⇒ składowe klasy.

Kolejność konstrukcji / destrukcji obiektów (cd1)

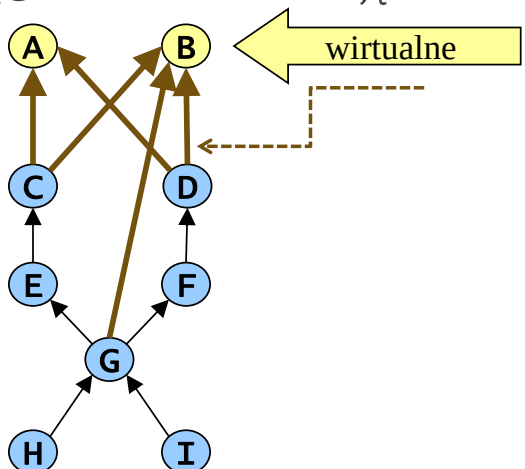
Kolejność konstrukcji dla deklaracji `H h;`
(@ oznacza konstrukcję składowych klasy)




```
A; @
B; @
C; @
E; @
B; @
D; @
F; @
G; @
H; @
```


Kolejność konstrukcji / destrukcji obiektów (cd1)

Kolejność konstrukcji dla deklaracji `H h;`
(@ oznacza konstrukcję składowych klasy)




```
A; @
B; @
C; @
E; @
B; @
D; @
F; @
G; @
H; @
```

Funkcje wirtualne i ochrona dostępu

- Podmiana funkcji wirtualnej w klasie pochodnej jako prywatnej nie blokuje dostępu do polimorfizmu poprzez wskazanie lub referencję na klasę bazową.

```
// ...
struct Bezier: FiguraOtwarta
{ void rysuj() { cout<< "Bezier::rysuj()\n"; }
  private:
  Figura* klonuj() { return new Bezier(*this); }
};

int main()
{ Bezier& bref = *new Bezier;
  Figura& fr   = bref;
  Figura& fref = *bref.klonuj(); // BŁĄD: private
  Figura& fref1= *fr.klonuj();   // OK
}
```

- Przyjąć zasadę: interfejs publiczny powinien zachować dostępność w klasach pochodnych

Funkcje wirtualne w interfejsie niewirtualnym

- Klasa bazowa może ustanowić uniwersalną usługę dla wszystkich klas pochodnych dopuszczającą lokalne modyfikacje polimorficzne.

```
struct Figura
{ // ...
   void rysuj(){ cout<<nazwa()<<"::rysuj()\n"; }// niewirtualna
   virtual const char* nazwa()=0; // Polimorficzna modyfikacja
  // ...
};
struct Bezier: FiguraOtwarta
{ // ...
   const char* nazwa(){ return "BEZIER"; }
  // ...
};
// Podobnie w pozostałych klasach pochodnych
int main() {
  Figura* f1 = new Histogram;
  Figura* f2 = new Bezier;
  Figura* f3 = f1->klonuj();
  Figura* f4 = f2->klonuj();
  Figura*fp[]= {f1, f2, f3, f4, 0};

  rysuj(fp);
}
```

```
HISTOGRAM::rysuj()
BEZIER::rysuj()
HISTOGRAM ::rysuj()
BEZIER::rysuj()
```