Dzisiejszy wykład

- Dziedziczenie wielobazowe
- Hierarchie klas

Dziedziczenie wielobazowe

- Klasa może posiadać więcej niż jedną bezpośrednią klasę bazową
- Rozważmy symulator, w którym jednoczesne zdarzenia są reprezentowane przez klasę *Task*, a zbieranie danych i wyświetlanie przez klasę *Displayed*. Zdefiniujmy klasę symulowanych obiektów, *Satellite*:

```
class Satellite : public Task, public Displayed {
 // ...
};
```

- Użycie więcej niż jednej bezpośredniej klasy bazowej nosi nazwę dziedziczenia wielobazowego

Dziedziczenie wielobazowe

- Poza operacjami zdefiniowanymi w klasie *Satellite*, na rzecz obiektu tego typu można wywoływać operacje zdefiniowane w klasach *Task* i *Displayed*

```
void f(Satellite& s)
{
 s.draw() ; // Displayed::draw()
 s.delay(10) ; // Task::delay()
 s.transmit() ; // Satellite::transmit()
}
```

- Obiekt klasy *Satellite* może być przekazywany do funkcji, przyjmującej *Task* lub *Displayed*

```
void highlight(Displayed*) ;
void suspend(Task*) ;
void g(Satellite* p)
{
 highlight(p) ; // pass a pointer to the Displayed part of the Satellite
 suspend(p) ; // pass a pointer to the Task part of the Satellite
}
```

Dziedziczenie wielobazowe

- Przy dziedziczeniu wielobazowym działają również funkcje wirtualne

```
class Task {
 // ...
 virtual void pending() = 0;
};
class Displayed {
 // ...
 virtual void draw() = 0;
};
class Satellite : public Task, public Displayed
{
 // ...
 void pending() ; // override Task::pending()
 void draw() ; // override Displayed::draw()
};
```

- Zapewnia to, że *Satellite::draw()* i *Satellite::pending()* będą wywołane dla obiektu Satellite traktowanego odpowiednio jako *Displayed* i *Task*

Dziedziczenie wielobazowe i niejednoznaczności

- Dwie klasy podstawowe mogą mieć składowe o takich samych nazwach

```
class Task {
 // ...
 virtual debug_info* get_debug() ;
};
class Displayed {
 // ...
 virtual debug_info* get_debug() ;
};
```

- Kiedy używamy obiektu *Satellite*, należy rozstrzygnąć niejednoznaczność

```
void f(Satellite* sp)
{
 debug_info* dip = sp->get_debug() ; // error: ambiguous
 dip = sp->Task::get_debug() ; // ok
 dip = sp->Displayed::get_debug() ; // ok
}
```

Dziedziczenie wielobazowe i niejednoznaczności

- Jawne usuwanie niejednoznaczności jest niewygodne, zwykle lepiej zdefiniować nową funkcję w klasie pochodnej

```
class Satellite : public Task, public Displayed {
 // ...
 debug_info* get_debug() // override Task::get_debug()
                         // and Displayed::get_debug()
 {
   debug_info* dip1 = Task::get_debug() ;
   debug_info* dip2 = Displayed::get_debug() ;
   return dip1->merge(dip2) ;
 }
};
```

- Taki zapis powoduje, że informacje o klasach podstawowych *Satellite* jest zlokalizowana w jednym miejscu. *Satellite::get_debug()* zasłania składowe *get_debug()* z klas podstawowych.

Dziedziczenie wielobazowe i niejednoznaczności

- Nazwa z kwalifikatorem zasięgu *Telstar::draw* może odnosić się do *draw* zadeklarowanego w *Telstar* lub jednej z klas podstawowych

```
class Telstar : public Satellite {
 // ...
 void draw()
 {
   draw() ; // oops!: recursive call
   Satellite::draw() ; // finds Displayed::draw
   Displayed::draw() ;
   Satellite::Displayed::draw() ; // redundant double qualification
 }
};
```

- Jeżeli *Satellite::draw* nie zostanie znalezione w klasie *Satellite*, kompilator rekursywnie poszukuje nazwy w klasach podstawowych, tj. poszukuje *Task::draw* lub *Displayed::draw*. Jeżeli znajdzie dokładnie jedną z tych funkcji, zostanie ona użyta. W przeciwnym przypadku program jest błędny.

Dziedziczenie i deklaracje *using*

- Nie stosuje się reguł rozstrzygania przeciążenia do różnych zasięgów klas, a zwłaszcza nie rozstrzyga się niejednoznaczności między funkcjami z różnych klas podstawowych na podstawie typów argumentów.
- Łączenie się klas, które zasadniczo nie są spokrewnione, jak *Task* i *Displayed* w przykładzie z satelitą, powoduje, że podobieństwo nazw zwykle nie wskazuje na wspólny cel. Takie konflikty nazw zwykle stanowią dużą niespodziankę dla programisty.

```
class Task {
 // ...
 void debug(double p) ; // print info only if priority is lower than p
};
class Displayed {
 // ...
 void debug(int v) ; // the higher the 'v,'
                     // the more debug information is printed
};
class Satellite : public Task, public Displayed {
 // ...
};
void g(Satellite* p){
 p->debug(1) ; // error: ambiguous.
               // Displayed::debug(int) or Task::debug(double) ?
 p->Task::debug(1) ; // ok
 p->Displayed::debug(1) ; // ok
}
```

Dziedziczenie i deklaracje *using*

- Jeżeli użycie tej samej nazwy w różnych klasach podstawowych było wynikiem świadomej decyzji projektowej, a użytkownik chciał dokonać wyboru na podstawie typów argumentów, można skorzystać z deklaracji użycia, która wprowadza funkcje do wspólnego zasięgu

```
class A {
 public:
   int f(int) ;
   char f(char) ;
   // ...
};
class B {
 public:
   double f(double) ;
   // ...
};
class AB: public A, public B {
 public:
   using A::f;
   using B::f;
   char f(char) ; // hides A::f(char)
   AB f(AB) ;
};
void g(AB& ab)
{
 ab.f(1) ;        // A::f(int)
 ab.f('a') ;      // AB::f(char)
 ab.f(2.0) ;      // B::f(double)
 ab.f(ab) ;       // AB::f(AB)
}
```

Dziedziczenie i deklaracje *using*

- Deklaracje użycia umożliwiają programiście tworzenie zbioru funkcji przeciążonych z klas podstawowych i klasy pochodnej. Funkcje zadeklarowane w klasie pochodnej zasłaniają funkcje, które w przeciwnym razie byłyby dostępne z klasy podstawowej. Funkcje wirtualne z klas podstawowych mogą być zastępowane na zwykłych zasadach.
- Deklaracja użycia w definicji klasy musi odnosić się do składowych klasy podstawowej; nie można jej użyć w odniesieniu do składowej klasy spoza tej klasy, jej klas pochodnych i ich metod. Dyrektywa użycia nie może się pojawić w definicji klasy i nie można jej użyć w odniesieniu do klasy.

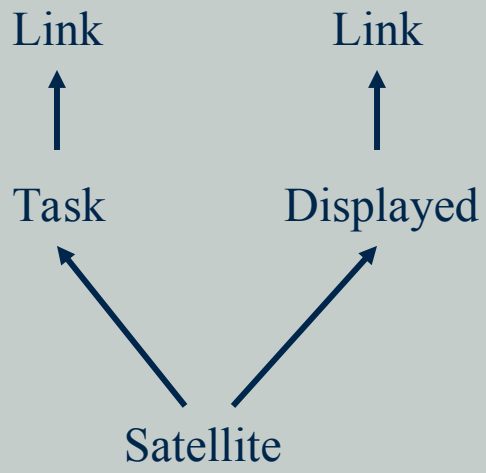
Wielokrotne wystąpienia klasy podstawowej

- Wraz z możliwością specyfikowania więcej niż jednej klasy podstawowej pojawia się możliwość dwukrotnego posiadania jakiejś klasy jako podstawowej Gdyby zarówno klasa *Task* jak i klasa *Displayed* były wyprowadzone z klasy *Link*, to klasa *Satellite* miałaby dwa dowiązania

```
struct Link {
 Link* next;
};
class Task : public Link {
 // the Link is used to maintain a list of all Tasks
 // (the scheduler list)
 // ...
};
class Displayed : public Link {
 // the Link is used to maintain a list
 // of all Displayed objects (the display list)
 // ...
};
```

Wielokrotne wystąpienia klasy podstawowej

- Nie powoduje to żadnych problemów. Do reprezentowania dowiązań służą dwa osobne obiekty klasy *Link* i te dwie listy wzajemnie sobie nie przeszkadzają.
- Nie można odwoływać się do składowych klasy *Link*, nie ryzykując niejednoznaczności



Wielokrotne wystąpienia klasy podstawowej

- Przykłady, w których wspólna klasa podstawowa nie powinna być reprezentowana przez dwa odrębne obiekty, można realizować z użyciem wirtualnej klasy podstawowej.
- Zwykle klasa podstawowa zwielokrotniona tak, jak tutaj Link, jest szczegółem implementacyjnym i nie powinno się jej używać spoza jej bezpośredniej klasy pochodnej. Jeśli trzeba się odwoływać do takiej klasy podstawowej z miejsca, gdzie jest widoczna więcej niż jedna kopia, to trzeba jawnie kwalifikować odwołanie, by uniknąć niejednoznaczności

```
void mess_with_links(Satellite* p)
{
 p->next = 0; // error: ambiguous (which Link?)
 p->Link::next = 0; // error: ambiguous (which Link?)
 p->Task::next = 0; // ok
 p->Displayed::next =0; // ok
 // ...
}
```

Zastępowanie funkcji wirtualnych

- Funkcję wirtualną powielonej klasy podstawowej można zastąpić (pojedynczą) funkcją w klasie pochodnej. Tak na przykład można reprezentować zdolność obiektu do przeczytania siebie z pliku i zapisania siebie z powrotem do pliku

```
class Storable {
 public:
   virtual const char* get_file() = 0;
   virtual void read() = 0;
   virtual void write() = 0;
   virtual ~Storable() {write() ; } // to be called
                                    // from overriding destructors
};
```

- Można skorzystać z tego do projektowania klas, które można stosować niezależnie lub w połączeniu do budowania bardziej złożonych klas

Zastępowanie funkcji wirtualnych

- Sposobem na zatrzymanie i ponowne rozpoczęcie symulacji jest przechowanie składników symulacji i odtworzenie ich w późniejszym terminie. Oto implementacja tej idei:

```
class Transmitter : public Storable {
 public:
   void write() ;
   // ...
};
class Receiver : public Storable {
 public:
   void write() ;
   // ...
};
class Radio : public Transmitter, public Receiver {
 public:
   const char* get_file() ;
   void read() ;
   void write() ;
   // ...
};
```

- Zazwyczaj funkcja zastępująca wywołuje swoje wersje z klas podstawowych, a następnie wykonuje pracę charakterystyczną dla klasy pochodnej

```
void Radio::write()
{
 Transmitter::write() ;
 Receiver::write() ;
 // write radio-specific information
}
```

Wirtualne klasy podstawowe

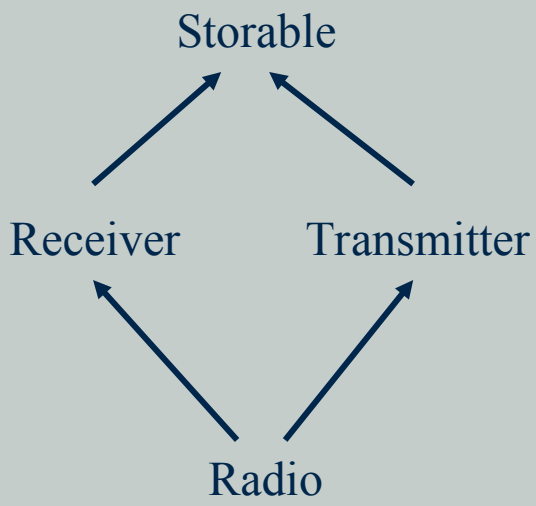
- Przykład z radiem działa, gdyż klasę *Storable* można bezpiecznie, wygodnie i efektywnie powielać. Klasy, które są dobrymi klockami do budowania innych klas, często nie mają tej cechy. Klasę *Storable* można np. zdefiniować w ten sposób, że będzie zawierać nazwę pliku używanego do przechowywania obiektu:

```
class Storable {
 public:
   Storable(const char* s) ;
   virtual void read() = 0;
   virtual void write() = 0;
   virtual ~Storable() ;
 private:
   const char* store;
   Storable(const Storable&) ;
   Storable& operator=(const Storable&) ;
};
```

Wirtualne klasy podstawowe

- Po wykonaniu tej pozornie drobnej zmiany, trzeba zmienić projekt klasy *Radio*. Wszystkie części obiektu muszą współdzielić pojedynczą kopię obiektu klasy *Storable*, gdyż w przeciwnym razie zbyt dużo trudu będzie kosztowało uniknięcie przechowywania wielu kopii obiektu. Mechanizmem do specyfikowania takiego współdzielenia jest wirtualna klasa podstawowa. Każda wirtualna klasa podstawowa klasy pochodnej jest reprezentowana przez ten sam (współdzielony) obiekt, np.

```
class Transmitter : public virtual Storable {
 public:
   void write() ;
   // ...
};
class Receiver : public virtual Storable {
 public:
   void write() ;
   // ...
};
class Radio : public Transmitter, public Receiver
{
 public:
   void write() ;
   // ...
};
```



Programowanie wirtualnych klas podstawowych

- Definiując funkcje klasy z wirtualną klasą podstawową, programista zwykle nie wie, czy klasa podstawowa będzie współdzielona z innymi klasami pochodnymi. Stanowi to problem w sytuacji, gdy implementuje się usługę która wymaga jednokrotnego wywołania funkcji z klasy podstawowej.
- Język zapewnia, że konstruktor wirtualnej klasy podstawowej jest wywoływany dokładnie raz: jest on wywoływany (niejawnie lub jawnie) z konstruktora pełnego obiektu (konstruktora dla klasy najbardziej pochodnej)

```
class A { // no constructor
 // ...
};
class B {
 public:
   B(); // default constructor
   // ...
};
class C {
 public:
   C(int); // no default constructor
};
class D : virtual public A, virtual public B, virtual public C
{
 public:
  D() { /* ... */ }; // error: no default constructor for C
  D(int i) :C(i) { /* ... */ }; // ok
  // ...
};
class E: public D
{
 public:
  E() { /* ... */ }; // error: no default constructor for C
  E(int i) :C(i) { /* ... */ }; // ok
  // ...
};
```

- Konstruktor wirtualnej klasy podstawowej jest wywoływany przed konstruktorami jej klas pochodnych

Programowanie wirtualnych klas podstawowych

- Programista może, w miarę potrzeby, zasymulować ten schemat, wywołując funkcję należącą do wirtualnej klasy podstawowej tylko z najbardziej pochodnej klasy. Załóżmy na przykład, że mamy klasę podstawową Window zaprogramowaną tak, że może narysować swoją zawartość:

```
class Window {
 // basic stuff
 virtual void draw() ;
};
```

- Ponadto mamy różne sposoby dekorowania okna i dodawania udogodnień:

```
class Window_with_border : public virtual Window {
 // border stuff
 void own_draw() ; // display the border
 void draw() ;
};
class Window_with_menu : public virtual Window {
 // menu stuff
 void own_draw() ; // display the menu
 void draw() ;
};
```

- Funkcje *own_draw()* nie muszą być wirtualne, gdyż będą wywoływane z funkcji wirtualnej *draw()*, która "zna" typ obiektu, dla którego go wywołano.

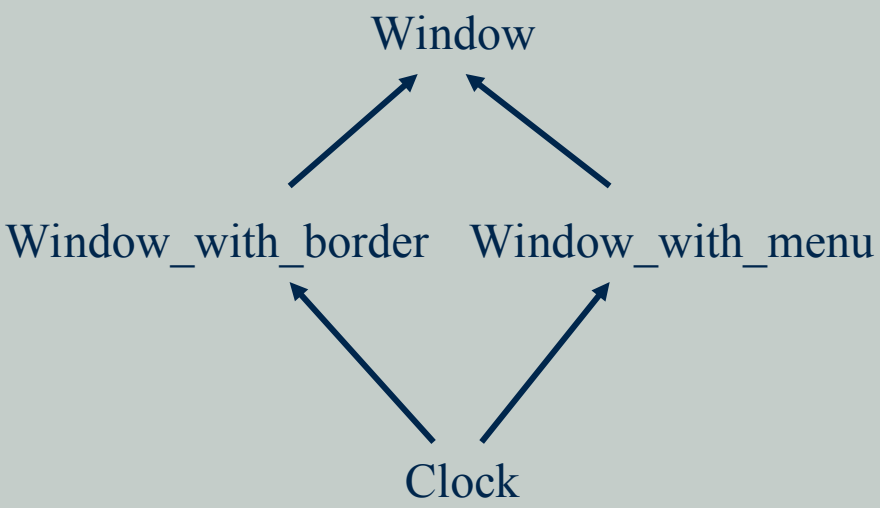
Programowanie wirtualnych klas podstawowych

- Z tych klas można zbudować sensowną klasę *Clock*:

```
class Clock : public Window_with_border, public Window_with_menu {
 // clock stuff
 void own_draw(); // display the clock face and hands
 void draw();
};
```

- Można teraz zapisać funkcje draw() za pomocą funkcji own_draw() w taki sposób, by wywołanie draw(), niezależnie od rodzaju okna, dla którego się to robi, powodowało jednokrotne wykonanie Window::draw()

```
void Window_with_border::draw()
{
 Window::draw() ;
 own_draw(); // display the border
}
void Window_with_menu::draw()
{
 Window::draw() ;
 own_draw(); // display the menu
}
void Clock::draw()
{
 Window::draw() ;
 Window_with_border::own_draw() ;
 Window_with_menu::own_draw() ;
 own_draw() ; // display the clock
              // face and hands
}
```

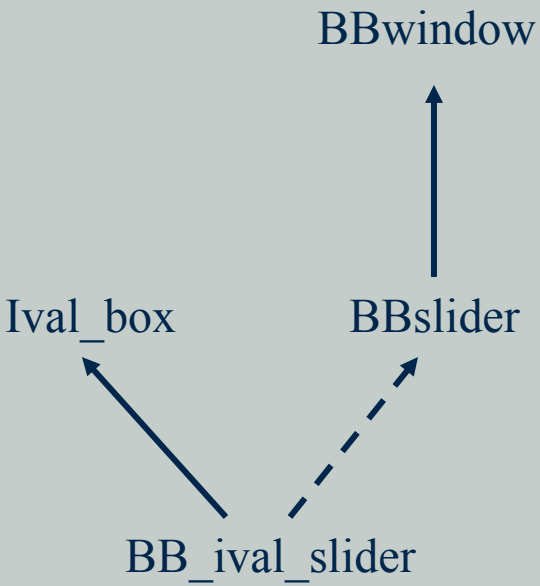


Używanie wielodziedziczenia

- Najprostszym i najbardziej narzucającym się sposobem użycia wielodziedziczenia jest "sklejenie" dwóch niezwiązanych klas, jako części implementacji trzeciej klasy. Przykładem jest klasa *Satellite* zbudowana z klas *Task* i *Displayed*.
- Bardziej podstawowe jest użycie wielodziedziczenia do implementacji klas abstrakcyjnych, gdyż wpływa ono na sposób projektowania programu.

Używanie wielodziedziczenia

```
class Ival_box {
 public:
   virtual int get_value() = 0;
   virtual void set_value(int i) = 0;
   virtual void reset_value(int i) = 0;
   virtual void prompt() = 0;
   virtual bool was_changed() const = 0;
   virtual ~Ival_box() { }
};
class BB_ival_slider
 : public Ival_box // interface
 ,protected BBslider // implementation
{
 // implementation of functions required
 // by 'Ival_slider' and 'BBslider'
 // using the facilities provided by 'BBslider'
};
```



- W tym przykładzie dwie klasy podstawowe pełnią logicznie różne role. Jedna jest publiczną klasą abstrakcyjną dostarczającą interfejs, a druga jest chronioną klasą konkretną dostarczającą szczegóły implementacyjne.
- Użycie wielodziedziczenia jest tutaj prawie konieczne, ponieważ klasa pochodna musi zastąpić funkcje wirtualne z implementacji i z interfejsu.

Zastępowanie funkcji z wirtualnych klas podstawowych

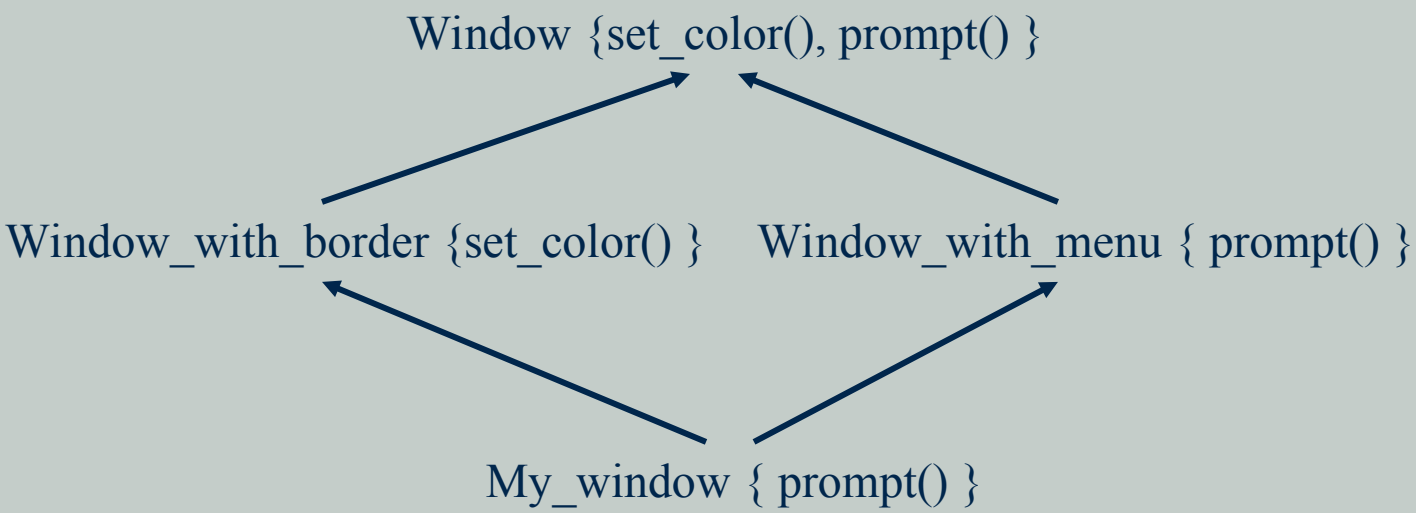
- Klasa pochodna może zastąpić funkcję wirtualną swojej bezpośredniej lub pośredniej wirtualnej klasy podstawowej
- Dwie różne klasy mogą zastąpić różne funkcje wirtualnej klasy podstawowej
- W ten sposób kilka klas pochodnych może złożyć się na implementację interfejsu reprezentowanego przez wirtualną klasę podstawową

```
class Window {
 // ...
 virtual set_color(Color) = 0; // set background color
 virtual void prompt() = 0;
};
class Window_with_border : public virtual Window {
 // ...
 set_color(Color) ; // control background color
};
class Window_with_menu : public virtual Window {
 // ...
 void prompt() ; // control user interactions
};
class My_window : public Window_with_menu, public Window_with_border {
 // ...
};
```

Zastępowanie funkcji z wirtualnych klas podstawowych

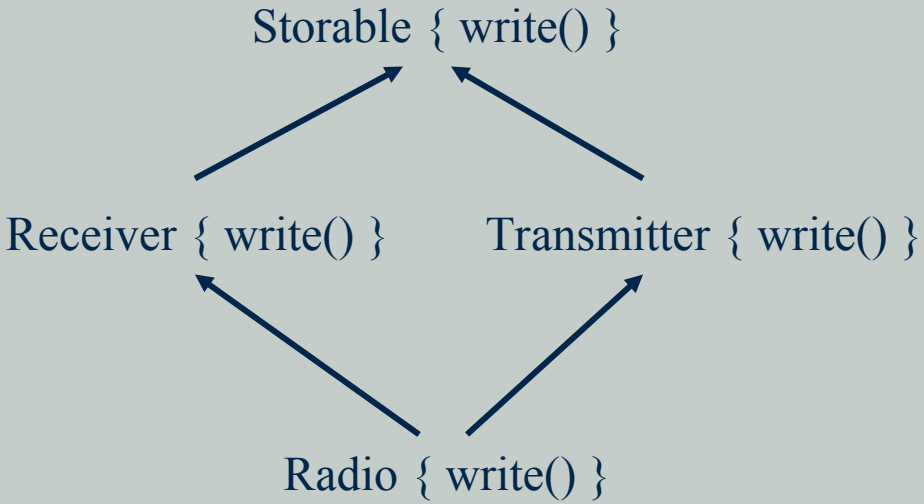
- Jeżeli różne klasy pochodne zastąpią tą samą funkcję, jest to dozwolone tylko wtedy, gdy klasa zastępująca pochodzi od wszystkich innych klas, które zastępują tę funkcję, tzn. jedna funkcja musi zastępować wszystkie pozostałe.
- *My_window* może zastąpić funkcję *prompt()*

```
class My_window : public Window_with_menu, public Window_with_border {
 // ...
 void prompt() ; // don't leave user interactions to base
};
```



Zastępowanie funkcji z wirtualnych klas podstawowych

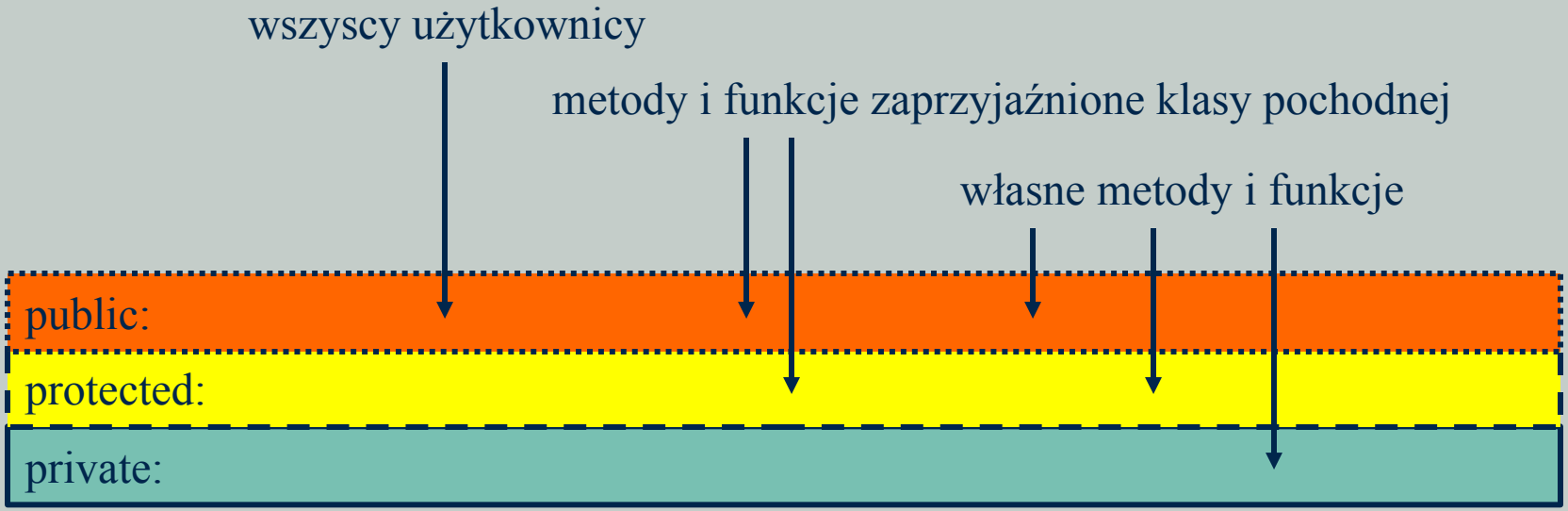
- Jeśli dwie klasy zastępują funkcję z klasy podstawowej, lecz żadna nie zastępuje drugiej, to hierarchia jest błędna
- Nie można skonstruować żadnej tablicy funkcji wirtualnych, ponieważ wywołanie tej funkcji dla pełnego obiektu byłoby niejednoznaczne.
- Gdyby klasa Radio nie deklarowała funkcji *write()*, to deklaracje *write()* w *Receiver* i *Transmitter* spowodowałyby błąd podczas definiowania Radio.
- Taki konflikt rozwiązuje się przez dodanie funkcji zastępującej do klasy najbardziej pochodnej



- Klasę, która dostarcza część - ale nie całość - implementacji wirtualnej klasy podstawowej, nazywa się często domieszką (ang. *mixin*)

Kontrola dostępu

- Składowa klasy może być prywatna, chroniona lub publiczna
  - Jeśli jest prywatna, to jej nazwy mogą używać jedynie metody i funkcje zaprzyjaźnione klasy, w której jest zadeklarowana
  - Jeśli jest chroniona, to jej nazwy mogą używać jedynie metody i funkcje zaprzyjaźnione klasy, w której jest zadeklarowana, oraz metody i funkcje zaprzyjaźnione klas pochodnych tej klasy
  - Jeśli jest publiczna, to jej nazwy mogą używać wszystkie funkcje



Składowe chronione

- Rozważmy ponownie klasę Window
- Funkcje *own_draw()* celowo niekompletnie realizowały swoją usługę
- Zaprojektowano je jako bloki budulcowe do użytku (jedynie) w klasach pochodnych, a nie jako bezpieczne i wygodne narzędzia ogólnego przeznaczenia
- Operacje *draw()* były zaprojektowane do ogólnego użytku
- Można to zapisać jawnie:

```
class Window_with_border {
 public:
   virtual void draw() ;
   // ...
 protected:
   void own_draw() ;
   // other toolbuilding stuff
 private:
   // representation, etc.
};
```

Składowe chronione

- Klasa pochodna może sięgać do chronionych składowych klasy podstawowej jedynie dla obiektów swojego typu:

```
class Buffer {
 protected:
   char a[128] ;
   // ...
};
class Linked_buffer : public Buffer{ /* ... */ };
class Cyclic_buffer : public Buffer {
// ...
 void f(Linked_  buffer* p) {
   a[0] = 0; // ok: access to cyclic_buffer's own protected member
   p->a[0] = 0; // error: access to protected member of different type
 }
};
```

- Zapobiega to subtelnym błędom, które mogłyby się pojawić, gdyby jedna klasa pochodna niszczyła dane należące do innej klasy pochodnej

Użycie składowych chronionych

- Prosty prywatno-publiczny model ukrywania danych nadaje się dobrze do typów konkretnych
- Gdy używa się klas pochodnych, ma się do czynienia z dwoma rodzajami użytkowników klasy: klasami pochodnymi i pozostałymi użytkownikami
- Funkcje składowe i zaprzyjaźnione, które implementują operacje klasy, działają na obiektach w imieniu tych użytkowników
- Model prywatno-publiczny pozwala programistom inaczej traktować implementatorów, a inaczej ogół użytkowników, lecz nie dostarcza sposobu zaspokajania specyficznych potrzeb klas pochodnych
- Deklarowanie pól jako chronionych jest przeważnie błędem projektowym
  - Umieszczanie znacznych ilości danych we wspólnej klasie, do użytku przez wszystkie klasy pochodne, naraża te dane na zniszczenie
  - Nie można łatwo zmienić struktury danych chronionych i publicznych

Dostęp do klas podstawowych

- Klasę podstawową można zadeklarować jako prywatną, chronioną lub publiczną

```
class X : public B{ /* ... */ };
class Y : protected B{ /* ... */ };
class Z : private B{ /* ... */ };
```

- Publiczne wyprowadzenie powoduje, że klasa pochodna staje się podtypem swojej klasy podstawowej
- Wyprowadzenie chronione i prywatne służy do reprezentowania szczegółów implementacyjnych
  - Chronione klasy podstawowe są potrzebne w hierarchii klas, w których normą jest dalsze wyprowadzanie
  - Klasy podstawowe prywatne są najbardziej potrzebne podczas definiowania klasy przez ograniczenie interfejsu do klasy podstawowej, by można było dostarczyć lepsze gwarancje

Dostęp do klas podstawowych

- Można opuścić specyfikator dostępu dla klasy podstawowej. Wówczas domyślnie przyjmuje się specyfikator *private* dla klasy i *public* dla struktury

```
class XX :B{ /* ... */ }; // B is a private base
struct YY :B{ /* ... */ }; // B is a public base
```

- Specyfikator dostępu dla klasy podstawowej kontroluje dostęp do składowych tej klasy podstawowej oraz konwersję wskaźników i referencji z typu klasy pochodnej do typu klasy podstawowej. Rozważmy klasę *D* wyprowadzoną z klasy podstawowej *B*
  - Jeśli *B* jest prywatną klasą podstawową, to jej publicznych i chronionych składowych mogą używać jedynie metody i funkcje zaprzyjaźnione *D*; tylko one mogą dokonywać konwersji *D** do *B**
  - Jeśli *B* jest chronioną klasą podstawową, to jej publicznych i chronionych składowych mogą używać jedynie metody i funkcje zaprzyjaźnione *D* oraz metody i funkcje zaprzyjaźnione klas pochodnych *D*; tylko one mogą dokonywać konwersji *D** do *B**
  - Jeśli *B* jest publiczną klasą podstawową, to jej publicznych składowych mogą używać wszystkie funkcje, a jej chronionych składowych mogą używać metody i funkcje zaprzyjaźnione *D* oraz metody i funkcje zaprzyjaźnione klas pochodnych *D*. Każda funkcja może dokonywać konwersji *D** do *B**

Wielodziedziczenie i kontrola dostępu

- Jeśli do nazwy lub klasy podstawowej można sięgnąć przez różne ścieżki w kracie wielodziedziczenia, to jest ona dostępna wtedy, kiedy jest dostępna przez jakąkolwiek ścieżkę

```
struct B {
 int m;
 static int sm;
 // ...
};
class D1 : public virtual B{ /* ... */ } ;
class D2 : public virtual B{ /* ... */ } ;
class DD : public D1, private D2{ /* ... */ };
DD* pd = new DD;
B* pb = pd;  // ok: accessible through D1
int i1 = pd->m; // ok: accessible through D1
```

- Jeżeli pojedynczy element jest dostępny przez kilka ścieżek, to nadal można odwoływać się do niego jednoznacznie

```
class X1 : public B{ /* ... */ } ;
class X2 : public B{ /* ... */ } ;
class XX : public X1, public X2{ /* ... */ };
XX* pxx = new XX;
int i1 = pxx->m; // error, ambiguous: XX::X1::B::m or XX::X2::B::m
int i2 = pxx->sm; // ok: there is only one B::sm in an XX
```

Deklaracje użycia i kontrola dostępu

- Deklaracja użycia nie daje możliwości uzyskania dostępu do dodatkowej informacji. Jest to mechanizm, który umożliwia wygodniejsze używanie dostępnej informacji. Po uzyskaniu dostępu można przekazać go innym użytkownikom

```
class B {
 private:
   int a;
 protected:
   int b;
 public:
   int c;
};
class D : public B {
 public:
   using B::a; // error: B::a is private
   using B::b; // make B::b publically available through D
};
```

- Jeżeli połączy się deklarację użycia z prywatnym lub chronionym wyprowadzeniem, to można jej używać do specyfikowania interfejsów do pewnych, ale nie wszystkich, udogodnień zwykle oferowanych przez klasę

```
class BB : private B{ // give access to B::b and B::c, but not B::a
 using B::b;
 using B::c;
};
```