Klasy 2

- Klasy są głównym elementem większości programów. Podstawowym zadaniem programisty jest ta zwana abstrakcja danych, czyli zdefiniowanie klas odpowiednich do zadania.
- Tworząc nowe klasy możemy używać już istniejących klas na wiele sposobów.
- Może to być kompozycja - obiekt jednej klasy zawarty jest jako składnik w innej klasie. Obiekt będący składnikiem klasy nie korzysta z żadnych elementów klasy nadrzędnej, a jego istnienie - tworzenie i niszczenie jest zarządzane w całości przez nadrzędną klasę. W przypadku kompozycji mówimy, że klasa nadrzędna „zawiera” jakiś obiekt.
- Przykładem może być klasa nadrzędna (osoba) która zawiera jako składnik inną klasę (historia zatrudnienia).

Klasy 2

- Innym sposobem wykorzystania istniejących klas jest tak zwana agregacja. Jest to sytuacja, gdzie obiekt pewnej klasy jest składnikiem innej klasy, ale istnieje niezależnie. Tworzenie i niszczenie nie należy do klasy nadrzędnej. W typowej sytuacji nadrzędna klasa zawiera wskaźnik albo referencję do innego obiektu, a nie obiekt jako taki. W odróżnieniu od kompozycji obiekt będący składnikiem może należeć też do innych klas. W przypadku agregacji mówimy, że klasa nadrzędna „posiada" jakiś obiekt.
- Tutaj przykładem może być klasa nadrzędna (pracodawca), i jej składnik, pracownik (osoba). Osoba jest tworzona niezależnie od pracodawcy, i może być składnikiem innych klas, np. klasy „związek zawodowy".

Klasy 2

- Jeszcze inny związek pomiędzy klasami to tak zwana asocjacja. Klasa może zawierać składniki innej klasy, przy czym taka relacja może być wzajemna. Typowo klasa zawiera tylko niektóre składniki innej klasy. W przypadku asocjacji mówimy, że jedna klasa „używa” obiektu innej klasy.
- Różnica pomiędzy asocjacją a agregacją jest teka, ze asocjacja może być wzajemna. Przykładem może być klasa „lekarz” i klasa „pacjent”. Każda z nich może zawierać obiekty drugiej.

Dziedziczenie

- Najważniejszą relacją pomiędzy klasami jest dziedziczenie, które jest bardzo starannie wyspecyfikowanym elementem języka C++.
- Idea dziedziczenia jest prosta. Zamiast tworzyć klasę od zera, zaczynamy od istniejącej klasy, po czym dodajemy bądź modyfikujemy składniki.
- Tworzenie klasy `A` jako klasy pochodnej (potomka) klasy pierwotnej (przodka) klasy `B` wygląda następująco:

```
class B
{
        ...
}
        ...
class A : B
{
        ...
}
```

Dziedziczenie

- Prawidłowy proces konstrukcji klas przez dziedziczenie powinien zacząć się od napisania najbardziej ogólnej klasy. Kolejni potomkowie powinni być dopasowani do bardziej szczegółowych typów danych. Powstaje w ten sposób hierarchia klas w której nie powtarza się ten sam kod.
- Przykładem może być `iostream`. Zdefiniowana jest klasa `ios`. Potomkami tej klasy są `istream`, `ostream` i `streambuf`
- Klasa `ios` zawiera wszystkie podstawowe metody wejścia/wyjścia, w tym sam bufor danych, metody do manipulacji nim, kontroli rozmiaru, alokacji pamięci, przeładowane operatory << i >>. Klasa `istream` dodaje wszystko co jest nam potrzebne do konkretnie wczytywania, np. `getline()`.

Dziedziczenie

- Proces tworzenia obiektów pochodnych następuje kolejno od obiektów najbardziej zagnieżdżonych (najstarszych). W przykładzie powyżej, gdzie klasa `A` dziedziczy po klasie `B` w momencie tworzenia obiektu klasy `A` najpierw stworzony zostaje obiekt klasy `B`. a następnie obiekt klasy `A`. Wyboru konstruktora obiektu `B` można dokonać wywołując go jawnie na liście inicjalizacyjnej konstruktora obiektu klasy `A`.
- Dziedziczenie oczywiście można iterować. Jeżeli chcemy wywołać konkretnego konstruktora dla głębiej zagnieżdżonych obiektów, musimy przejść przez listy inicjalizacyjne wszystkich pośrednich potomków. Nie ma bezpośredniego dostępu do konstruktorów głębiej zagnieżdżonych.
- Przykłady: `Prog1.cpp`, `Prog1a.cpp`

Dziedziczenie

- Przypomnijmy, kiedy tworzony jest obiekt klasy pochodnej, wywoływane są kolejno konstruktory klas pierwotnych, w kolejności od najbardziej pierwotnej
- Kiedy obiekt klasy dziedziczonej jest niszczony, wywoływane są kolejno destruktory w kolejności odwrotnej.
- Przypomnijmy, że składniki i metody klasy mogą być typu `private`, `public` oraz `protected`. Domyślnym typem jest `private`. `protected` działa tak jak `private`, ale daje dostęp do składnika metodom dowolnej klasy dziedziczącej.
- Dziedziczenie też może mieć powyższe 3 typy. składnia jest następująca:

```
class B
{
        ...
}
        ...
class A : public/private/protected B
{
        ...
}
```

Dziedziczenie

- Domyślnym typem dziedziczenia jest `private`.
- Niezależnie od typu dziedziczenia składniki `private` klasy bazowej nie są dostępne w klasie pochodnej. Dostęp do nich musi wykorzystywać metody klasy bazowej.
- Jeżeli dziedziczenie jest typu `public`, to składniki `public` i `protected` klasy bazowej zachowują swój typ w klasie pochodnej.
- Jeżeli dziedziczenie jest `private`, to składniki `public` i `protected` stają się `private`.
- Jeżeli dziedziczenie jest `protected`, to składniki klasy bazowej `public` i `protected` stają się w klasie pochodnej `protected`.

Dziedziczenie

- Klasa pochodna może mieć dodatkowe składniki, zmienne i metody, lub może modyfikować funkcje dziedziczone. Jeżeli w klasie pochodnej zdefiniowana jest taka sama funkcja jak w klasie bazowej, to dla obiektu klasy pochodnej definicja z klasy bazowej jest przesłonięta. Funkcja jest uważana za taką samą, jeżeli ma taką samą nazwę, takie same parametry formalne, i zwraca taki sam typ.
- Wszystkie składniki nieprywatne klasy bazowej są dziedziczone przez klasę pochodną. Klasa pochodna może z nich korzystać tak, jakby były zdefiniowane bezpośrednio w niej. Wyjątkiem są konstruktory, destruktory i operator przypisania `operator=`. Wyjątki te mają oczywiste wyjaśnienie.

Dziedziczenie

- Klasa pochodna to wszystkie składniki klasy bazowej plus dodatkowe rzeczy. Konstruktor i destruktor klasy bazowej nie wie o tych wszystkich dodatkach. Nie może ich więc zainicjalizować. Podobnie z destruktorem i operatorem przypisania.
- Funkcję z klasy bazowejj (która jest przesłonięta w klasie potomnej) wciąż z obiektu klasy pochodnej można wywołać używając operatora `::`
- Przykład `Prog2.cpp`

Dziedziczenie

- Podsumujmy pierwsze idee dziedziczenia. Wyobraźmy sobie sytuację, że mamy gotowy kod, ale potrzebujemy pewnych modyfikacji, pewnej rozbudowy. Pierwszym odruchem może być po prostu modyfikacja istniejącego kodu. Ale często kod, który mamy jest wytworzony w innym procesie, jest częścią większego programu. Co, jeżeli pojawi się nowa wersja? W tej sytuacji należy użyć dziedziczenia. Piszemy własną klasę pochodną, dziedziczącą po wyjściowej.
- Można też wyobrazić sobie sytuację, gdzie kodu wyjściowego nie możemy modyfikować. Na przykład, chcielibyśmy zmodyfikować którąś z bibliotecznych klas C++.

Funkcje wirtualne

- C++ pozwala wskaźnik do obiektu klasy pochodnej zadeklarować jako wskaźnik klasy bazowej. Może się to początkowo wydawać dziwne, ale jest to ważna zaleta. Intuicyjnie można myśleć, że klasa dziedziczy nie tylko składniki, ale też typ.
- Jeżeli klasa `B` jest klasą pochodną klasy `A`, to możliwa jest więc następująca sytuacja

```
B *pochodny_ptr;
A *bazowy_ptr;
bazowy_ptr = pochodny_ptr;
```

Funkcje wirtualne

- Należy myśleć, że typ `A` jest *ogólniejszy* niż typ `B`. Obiekty typu `B` są także obiektami typu `A`. Logiczna zależność w przypadku dziedziczenia jest typu: `B` jest rodzajem `A`. Można też myśleć o tym, jako o rzutowaniu typów. `pochodny_ptr` sygnalizuje kompilatorowi, że pod danym adresem mamy obiekt pochodny. `bazowy_ptr` sygnalizuje kompilatorowi, że pod danym adresem mamy obiekt bazowy. Zauważmy, że obie sytuacje są prawidłowe. Adres jest ten sam, a obiekt pochodny w pamięci zaczyna się od obiektu bazowego

Funkcje wirtualne

- Przypomnijmy, że kiedy tworzony jest obiekt klasy pochodnej, to najpierw wywoływany jest konstruktor klasy bazowej, i tworzony jest obiekt bazowy. Następnie wywoływany jest konstruktor klasy pochodnej i tworzony dodatkowy obiekt, który zawiera wszystkie dodatkowe elementy. Jego miejsce w pamięci jest zaraz po obiekcie bazowym.
- Jeżeli więc chodzi o *wartość* wskaźnika, to nie ma różnicy, czy wskazujemy na istniejący obiekt bazowy, czy pochodny. Adres jest taki sam. Różnica to typ wskaźnika.
- Osobie rozpoczynającej programowanie może się to wydawać sprawą marginalną, ale jest to jedną z podstawowych korzyści dziedziczenia.

Funkcje wirtualne

- Funkcje napisane dla argumentów będących obiektami klasy bazowej mogą przyjmować obiekty klas pochodnych. Używane klasy pochodne mogą w ogóle nie istnieć w momencie tworzenia funkcji. Jeżeli funkcja nie korzysta ze specyfiki klasy pochodnej, można jej przekazać takie obiekty - oczywiście tylko przez wskaźnik (albo referencję). Zauważmy różnicę w funkcjonalności w porównaniu z przeładowaniem funkcji.
- Podobnie z wartościami zwracanymi przez funkcję. Jeżeli program spodziewa się wartości zwracanej będącej obiektem klasy bazowej można mu zwrócić obiekt klasy pochodnej.

Funkcje wirtualne

- Korzystamy też z tej funkcjonalności w przypadku inicjalizacji: możemy zainicjalizować obiekt klasy bazowej obiektem klasy pochodnej:

```
A obiekt_pochodny;
B obiekt_bazowy = obiekt_pochodny;
```

  Zauważmy przy okazji, że w powyższym przykładzie = nie jest wywołaniem operatora przypisania. Jest to wywołanie konstruktora kopiującego klasy bazowej B:

```
B::B( B &obiekt );
```

  Jest więc to w rzeczywistości szczególny przypadek opisanego powyżej przekazywania do funkcji argumentu pochodnego zamiast bazowego.

- Jeszcze inną sytuacją w której często przydatny jest opisywany mechanizm jest używanie przeładowanych operatorów.

Funkcje wirtualne

- Pojawia się jednak problem, jeżeli klasy pochodne zawierają modyfikacje funkcji z klas bazowych. Przy wywołaniu takich funkcji przez obiekty klasy pochodnej powinny być wywoływane funkcji zmodyfikowane, natomiast przy wywoływaniu przez obiekty klas bazowych powinny być wywoływane funkcji zdefiniowane w klasie bazowej.
- Jeżeli obiekty klasy pochodnej są też obiektami klasy bazowej (tylko w sensie wskaźników i referencji), to którą wersję funkcji wywołać?
- Przypomnijmy, dotyczy to tylko sytuacji, gdy funkcje są przesłaniane. To znaczy klasa pochodna musi definiować funkcję taką samą jaka występowała w klasie bazowej. Taka sama nazwa, argumenty, typ zwracany, `const` czy nie-`const`.

Funkcje wirtualne

- Wyobraźmy sobie, że obie klasy, bazowa `B` i pochodna `A` mają zdefiniowaną funkcję

```
void B::wypisz(   ) {cout << "Funkcja z klasy bazowej";
void A::wypisz(   ) {cout << "Funkcja z klasy pochodnej";
```

  Następnie korzystamy z kompatybilności wskaźników:

```
A obiekt_pochodny;
A *wsk_pochodny = &obiekt_pochodny;
B *wsk_bazwy = wsk_pochochodny;
wsk_bazowy -> wypisz();
```

- Zgodnie z typem wskaźnika kompilator powinien wywołać funkcję z klasy `B`. Ale wygląda, że nie o to chodziło, gdyż w rzeczywistości `wsk_pierw` wskazuje na adres, pod którym istnieje obiekt klasy `A`.

Funkcje wirtualne

- Rozwiązaniem tego problemu jest deklaracja funkcji w klasie bazowej jako `virtual`. Jeśli funkcja jest wirtualna, to jej dziedziczenie jest inteligentne.
- W przypadku funkcji wirtualnej kompilator nie tylko sprawdza jakiego typu jest wskaźnik, ale też na jaki typ obiekt w rzeczywistości wskazuje - na obiekt klasy bazowej, czy jakiejś dziedziczonej (przypomnijmy, że dziedziczenie może być wielokrotne).
- Jeżeli funkcja jest wirtualna, to kompilator znajduje najgłębiej dziedziczoną (nie głębiej niż obiekt wywołujący) wersję funkcji.
- Słowo `virtual` powinno występować tylko przy definicji funkcji w klasie bazowej. Można je jednak stosować też przy definicjach funkcji klas pochodnych - dla przypomnienia.
- Słowo `virtual` pojawia się tylko przy deklaracji. Jeżeli definicja jest odłożona, nie wymaga już słowa `virtual`
- Przykład `Prog3.cpp`

Funkcje wirtualne

- Czasem mówi się, że funkcje wirtualne cechuje *polimorfizm*. Decyzja, który wariant zostanie użyty odłożona jest do momentu wywołania.
- Kompilator realizuje polimorfizm w ten sposób, że tworzy tabelę modyfikacji (przez klasy pochodne) funkcji wirtualnej. Właściwą funkcję wybiera z tabeli po sprawdzeniu typu obiektu wywołującego.
- Przykład `Prog4.cpp`
- Przykład `Prog5.cpp`

Funkcje wirtualne

- Pamiętajmy o podstawowych zasadach użycia funkcji wirtualnych:
- Funkcji wirtualnych nie można wywoływać z konstruktorów ani destruktorów. Jest to oczywiste. Pamiętajmy, że konstruktor klasy bazowej wywoływany jest jako pierwszy. Obiekt pochodny nie jest jeszcze stworzony. Podobnie z destruktorem.
- Jeżeli piszemy własne destruktory, zawsze powinny być wirtualne. Powód jest oczywisty.
- Nie należy nadużywać wirtualności. Wirtualizowanie funkcji wiąże się z dodatkowo alokowaną pamięcią, oraz wolniejszym wywołaniem.
- Dostęp do funkcji wirtualnej regulowany jest przez typ wskaźnika. Jeżeli w klasie bazowej funkcja jest `public`, to wskaźniki typu bazowego mają do niej dostęp, nawet jeżeli w istocie wskazują na obiekt pochodny, w którym funkcja jest `private`.

Funkcje wirtualne

- Funkcja wirtualna może nie mieć żadnej treści w klasie bazowej. Do tej pory omawialiśmy sytuację, gdy klasa pochodna modyfikowała funkcję istniejącą w klasie bazowej. Ale ta funkcja w klasie bazowej może nie zawierać żadnej treści. Taką funkcję nazywa się *czysto wirtualną*, i jej definicja jest szczególna. Zamiast bloku z treścią przypisujemy jej wartość 0:

```
virtual jakas_funkcja() = 0;
```

- Jeżeli klasa zawiera chociaż jedną taką funkcję, to staje się tak zwaną klasą abstrakcyjną. Klasa abstrakcyjna nie może służyć do tworzenia obiektów. Staje się pełnoprawną klasą dopiero w postaci odziedziczonej. Co więcej, klasa dziedzicząca musi zdefiniować każdą czysto wirtualną funkcję z klasy bazowej. W przeciwnym wypadku również będzie klasą abstrakcyjna.