

# (Średnio)zaawansowane programowanie w C++

## Wykład 2 - polimorfizm, funkcje wirtualne, zarządzanie pamięcią

Robert Nowak

25L

# Plan wykładu

- ▶ Funkcje wirtualne
- ▶ Dziedziczenie wielobazowe
- ▶ Dynamiczna informacja o typie
- ▶ Zarządzanie pamięcią
  - ▶ operator `new` i `delete`,
  - ▶ przydzielanie pamięci dla dużej liczby małych obiektów,
  - ▶ alokator w bibliotece `stl`.

# Deklaracja i definicja

Deklaracja - sygnalizacja że pewna nazwa odnosi się do „czegoś”

```
extern int x; //deklaracja obiektu
long silnia(long n); //deklaracja funkcji
class Logger; //deklaracja klasy
template<typename T> class Node; //deklaracja szablonu
```

Definicja - podanie szczegółów;

```
int x = 4; //definicja i inicjacja zmiennej x
long silnia(long n) { //definicja funkcji
    if(n < 2) return 1;
    return n * silnia(n-1);
};
```

Inicjacja - nadanie początkowej wartości

**Definicja obiektu: wtedy gdy można zainicjować**

# Obiekt. Czas życia obiektu.

Obiekt: ma typ oraz unikalny identyfikator.

1-wartość: odnosi się do „czegoś” co istnieje w pamięci.

- ▶ automatyczne
  - ▶ tworzone w chwili napotkania definicji
  - ▶ są niszczone w momencie wyjścia z zasięgu (np. z bloku)
- ▶ statyczne (globalne i zadeklarowane jako `static`)
  - ▶ tworzone tylko raz
  - ▶ są niszczone po zakończeniu programu
- ▶ dynamiczne (tworzone na stercie)
  - ▶ operatory `new` i `delete` (uwaga! oddzielna wersja dla tablic),
  - ▶ programista steruje czasem życia obiektów (tworzy je i usuwa),
  - ▶ Nie używamy funkcji z `<stdlib.h>`, czyli `malloc`, `free` itp.
- ▶ tymczasowe

# Ukrywanie nazw

Nazwy „lokalne” przykrywają globalne

```
int x;  
//x jest obiektem typu int  
void f() {  
    double x;  
    //x ma typ double  
}
```

Podobnie przy dziedziczeniu

```
class Base {  
    int x;  
public:  
    virtual void f() {  
        //x jest obiektem typu int  
    }  
};
```

```
class Derived : public Base {  
public:  
    void x(){}  
    void f() override {  
        //x jest nazwą metody  
    }  
};
```

# Przeciążanie nazw, dyrektywa using

```
class Base {  
    public:  
    virtual void f();  
    virtual void f(int);  
}  
  
Derived d;  
d.f(2); //Błąd, nie ma takiej metody  
//metoda 'f' przykrywa Base::f(), Base::f(int)!
```

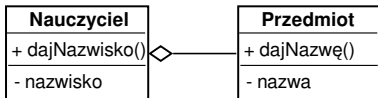
```
class Base {  
    public:  
    virtual void f();  
    virtual void f(int);  
}  
  
Derived d;  
d.f(); //OK, wołana Derived::f()  
d.f(2); //OK, wołana Base::f(int)
```

# Programowanie przyrostowe. Tworzenie nowych klas

Budowa klas na podstawie już istniejących, ponowne wykorzystanie kodu:

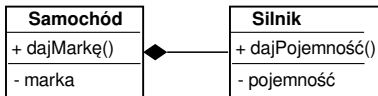
- ▶ agregacja,
- ▶ dziedziczenie.

Agregacja („posiada”) - gdy budowa z mniejszych kawałków większej całości.



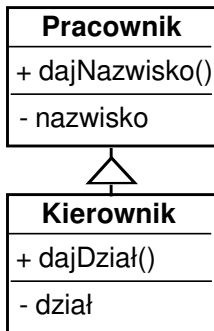
```
class Przedmiot { /* ... */ };
class Nauczyciel {
private:
    std::vector<Przedmiot*> przedm_;
};
```

Kompozycja - agregacja, obiekt składowy nie istnieje bez obiektu głównego



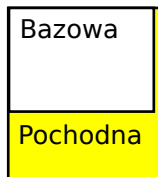
```
class Silnik { /* ... */ };
class Samochod {
private:
    Silnik silnik_;
};
```

# Dziedziczenie - relacja „może być traktowany jako”



```
//klasa bazowa
class Pracownik {
    //...
};

//klasy pochodna
class Kierownik : public Pracownik {
    //...
};
```



Dziedziczenie wprowadza powiązanie pomiędzy typami.



# Wycinanie

Działanie konstruktora kopiującego lub operatora przypisania:

```
class Pracownik { ... };  
class Kierownik : public Pracownik { ... };
```

```
Kierownik k(...);  
Pracownik p = k;
```

- ▶ kopiuje tylko część klasy,
- ▶ źródło niespodzianek i błędów,
- ▶ rozwiązanie: przekazywanie wskaźników lub referencji do obiektów.

# Problem typu dla obiektów

```
class Osoba {  
public:  
    void drukuj(ostream& os) const { os << "osoba"; }  
};  
class Pracownik : public Osoba {  
public://Przedefiniowanie metody (redefining)  
    void drukuj(ostream& os) const { os << "pracownik"; }  
};  
void drukujOsobe(const Osoba& f) {  
    f.drukuj(cout);  
}  
Osoba o;  
Pracownik p;  
p.drukuj( cout );//woła metodę Pracownik::drukuj  
drukujOsobe(o);//wołana metoda Osoba::drukuj  
drukujOsobe(p);//błąd! wołana metoda Osoba::drukuj
```

# Polimorfizm za pomocą pola typu (bardzo złe rozwiązanie)

```
class Osoba {  
public:  
    enum Typ { OSOBA, PRACOWNIK, KLIENT };  
    const Typ typ_; // Pole pamięta typ obiektu  
    Osoba(Typ t = OSOBA) : typ_(t) {}  
};  
class Pracownik : public Osoba {  
public:  
    Pracownik() : Osoba(PRACOWNIK) {}  
};  
void drukuj(const Osoba& o) {  
    switch(o.typ)  
    //...  
}
```

- ▶ kompilator nie potrafi sprawdzić poprawności
- ▶ kod staje się nieczytelny i trudno modyfikowalny

# Polimorfizm z wykorzystaniem funkcji wirtualnych

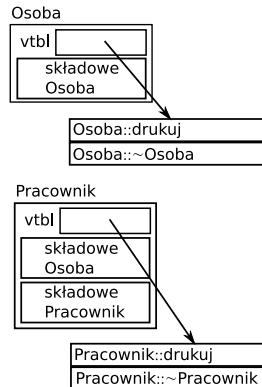
wspierane przez kompilator: późne wiązanie

```
class Osoba {  
    virtual void drukuj(ostream& os) const  
    { os << "osoba"; };  
    virtual ~Osoba() { /* ... */ }  
};  
class Pracownik : public Osoba {  
public://Nadpisywanie metody (overriding)  
    void drukuj(ostream& os) const override  
    { os << "pracownik"; }  
    ~Pracownik() override { /* ... */ }  
};
```

narzuty pamięciowe:

- ▶ wskaźnik w obiekcie (vtbl),
- ▶ tablica wskaźników dla klas dostarczających funkcji wirtualnych

narzuty czasowe: dodatkowe adresowanie pośrednie



# Typy klas

Klasa wartość (klasa autonomiczna):

- ▶ brak metod wirtualnych
- ▶ konstruktor, konstruktor kopiujący, operator przypisania, destruktor
- ▶ najczęściej obiekt automatyczny lub składowa klasy
- ▶ może być przekazywana przez wartość

Klasa bazowa dla hierarchii klas:

- ▶ używa metod wirtualnych, powinna mieć wirtualny destruktor
- ▶ najlepiej gdy abstrakcyjna albo prywatny konstruktor kopiujący i prywatny operator przypisania (zapobiega wycinaniu)
- ▶ najczęściej obiekt na stacku
- ▶ przekazywana przez wskaźnik lub referencję

# Metody - możliwości modyfikacji w klasach pochodnych

```
class Base {  
public:  
    //metoda  
    void a() { /* ... */ }  
    //metoda wirtualna  
    virtual void b() { /* ... */ }  
    //metoda czysto wirtualna  
    virtual void c() = 0;  
};
```

- ▶ metoda - kod nie może być zmieniany w klasach pochodnych
- ▶ metoda wirtualna - kod może (ale nie musi) być dostarczony w klasach pochodnych, domyślna implementacja w klasie bazowej
- ▶ metoda czysto wirtualna - kod musi być dostarczony w klasach pochodnych



# Alternatywa dla funkcji wirtualnych - wskaźnik do funkcji

//Przykładowa funkcja

```
int defaultCalcFun(const Calc& c){ /* ... */ }
```

```
class Calc {
```

```
public:
```

```
    using CalcFun = int (*)(const Calc&); //Sygnatura (typ) funkcji
```

```
    Calc(CalcFun cf = defaultCalcFun ) : calcFun_(cf) {}
```

```
    int calculate() const { return calcFun_(*this); } //Funkcja polimorficzna
```

```
private:
```

```
    CalcFun calcFun_; //Wskaźnik do funkcji
```

```
};
```

Polimorfizm za pomocą uchwytu (wskaźnika):

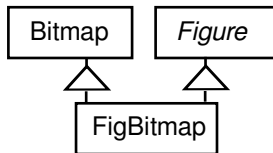
- ▶ zachowanie może być różne nawet dla obiektów tej samej klasy,
- ▶ zachowanie może się zmienić w czasie działania,
- ▶ funkcja nie jest metodą, ma dostęp tylko do interfejsu klasy.



# Dziedziczenie wielobazowe (multiple inheritance, MI)

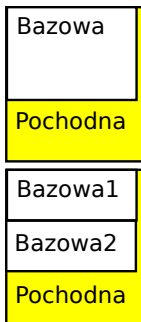
Wprowadzone, ponieważ może istnieć wiele niezależnych hierarchii klas.

*W C++ nie ma klasy bazowej (o nazwie np. `Object`) dla wszystkich innych klas.*



```
class FigBitmap
: public Figure, public Bitmap
{ /* ... */ };
```

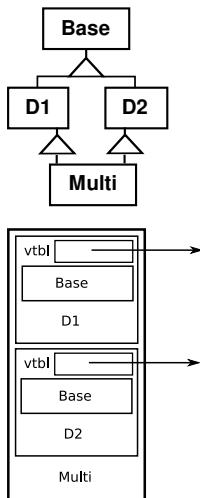
# Obiekty przy wielo-dziedziczeniu



```
class Bazowa { /* */ };
class Pochodna : public Bazowa { /* */ };
class B1 {
    public: void f();
};
class B2 {
    public: void f();
};
class P2 : public B1, public B2 { /* */ };
```

```
P2 d; // Rzutowanie w górę może zmienić adres
B1* pb1 = &d; //pb1 to ten sam adres co &d
B2* pb2 = &d; //pb2 jest innym adresem niż pb1
//Wołanie metody może być niejednoznaczne
d.f(); //Błąd kompilacji
static_cast<B1*>(d).f(); //OK, woła B1::f()
```

# Wielokrotne wystąpienie klasy podstawowej



```
class Base { /* ... */ };
```

```
class D1 : public Base { /* ... */ };
```

```
class D2 : public Base { /* ... */ };
```

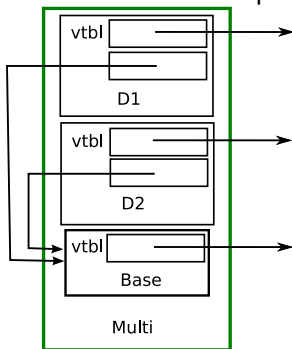
```
class Multi : public D1, public D2
{ /* ... */ };
```

Obiekty Klasy Multi zawierają dwa obiekty klasy Base

- ▶ niejednoznaczność
- ▶ nie zawsze jest to pożądane (zasoby)

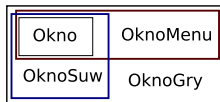
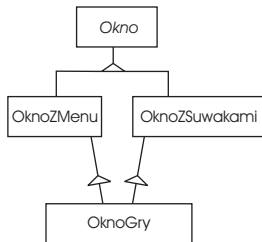
# Wirtualne klasy bazowe

Zapewnia, że tylko jedna instancja obiektu klasy bazowej będzie umieszczana w klasie pochodnej



```
class Base { /* ... */ };  
  
class D1 : virtual public Base  
{ /* ... */ };  
class D2 : virtual public Base  
{ /* ... */ };  
class Multi : public D1, public D2  
{ /* ... */ };
```

# Dziedziczenie wirtualne (przykład)



```
class Okno { /* ... */ };
```

```
class OknoZMenu : virtual public Okno  
{ /* ... */ };
```

```
class OknoZSuw : virtual public Okno  
{ /* ... */ };
```

```
class OknoGry  
: public OknoZMenu, public OknoZSuw  
{ /* ... */ };
```

# Metody wirtualne a konstruktory

Mechanizm funkcji wirtualnych nie działa w konstruktorach i destruktorach.

```
class Base {  
public:  
    Base(){ f(); }//Wołana Base::f()  
    virtual void f(){ /* ... */ }  
};  
class Derived : public Base {  
public:  
    Derived() : Base() { }  
    void f() override { /* ... */ }  
};  
Derived d; //Wołana Base::f() w konstruktorze  
Base* b = new Derived; //Wołana Base::f() w konstruktorze  
b->f(); //Wołana Derived::f()
```

# Dziedziczenie prywatne i chronione

```
class D: public B { ... }; //Dziedziczenie publiczne
class D: protected B { ... }; //Dziedziczenie chronione
class D: private B { ... }; //Dziedziczenie prywatne
```

Ochrona dla różnych rodzajów dziedziczenia:

rodzaj	Dostęp do		
	konwersji $D^* \rightarrow B^*$	składowych publicznych B	składowych chronionych B
publiczne	wszystkie funkcje		metody D oraz klas pochodnych po D
chronione	metody D oraz klas pochodnych po D		
prywatne	metody D		

# dynamiczna informacja o typie (RTTI)

typeid - operator, zwraca informację o typie

```
const type_info& typeid(nazwa_typu) throw();  
const type_info& typeid(wyrażenie) throw(bad_typeid);
```

Rzutowanie klasy bazowej na pochodną (w dół, downcasting)

```
class B1 { /* ... */ };  
class B2 { /* ... */ };  
class P : public B1, public B2 { /* ... */ };
```

```
B1* b = new P;  
//wskaźniki: dla odp. typu zwraca wskaźnik, inaczej nullptr  
P* p = dynamic_cast<P*>(b);  
//referencje: dla błędnego typu wyjątek 'bad_cast'  
P& r = dynamic_cast<P&>(b);  
//Rzutowanie skrócone (do klasy siostrzanej, crosscasting)  
B2* pb2 = dynamic_cast<B2*>(b);
```



# RTTI - koszty

- ▶ wielkość kodu
  - ▶ struktura `type_info` dla każdej klasy
  - ▶ wskaźnik do `type_info` w tablicy funkcji wirtualnych
  - ▶ lista referencji do struktur `type_info` do klas bazowych (dla klas, które dziedziczą)
- ▶ czas wykonania
  - ▶ wołanie `typeid` - uzyskanie wskaźnika z tablicy funkcji wirtualnych
  - ▶ wołanie `dynamic_cast` rekurencyjne przeszukanie referencji do `type_info` do klas bazowych (przechowywane w strukturze `type_info`)
  - ▶ badanie typu w `catch` - jak `dynamic_cast`

# Operatory new i delete

- ▶ operator new
- ▶ operator new[]
- ▶ operator delete
- ▶ operator delete[]

//Przykłady

```
Foo* f = new Foo; //Przydziela pamięć, woła konstruktor  
//Generuje 'bad_alloc' jeżeli brak pamięci
```

```
delete f; //woła destruktor, zwalnia pamięć
```

```
Foo* f = new Foo[N]; //Przydziela pamięć dla N elementów  
//woła konstruktory  
delete [] f; //woła destruktory, zwalnia pamięć
```

//Stary sposób zgłaszania błędów przydziału (przed ISO93)

```
Foo* f = new (std::nothrow) Foo; //Zwraca 0 jeżeli brak pamięci  
//Uwaga! konstruktor może rzucać wyjątek!
```

# nullptr - C++11 nowe słowo kluczowe

Sposoby oznaczania pustych wskaźników

- ▶ `#define NULL (void*)0`
- ▶ `0L`
- ▶ `nullptr`

Problemy z rzutowaniem i błędy:

```
void f(char*);  
void f(int);
```

```
f(0L); //niejednoznaczność  
f(nullptr); //będzie wołane f(char*)
```

# Obsługa błędów przydziału ::new

zanim ::new wygeneruje bad\_alloc woła funkcję użytkownika

```
#include <new> //Zawiera funkcję set_new_handler

void MyNewHandler() { /* np. zwolnienie pamięci */ }

//Ustawienie własnej funkcji obsługi niepowodzenia przydziału
set_new_handler(MyNewHandler);
//Usunięcie własnej funkcji obsługi
set_new_handler(nullptr);
```

Sensowna funkcja powinna:

- ▶ próbować „odzyskać” pamięci. Jeżeli pamięć zostanie zwolniona, to funkcja powinna zakończyć się (return).
- ▶ generować wyjątek bad\_alloc
- ▶ zainstalować lub odinstalować funkcję obsługi

# Definiowanie operatora new i delete dla klasy

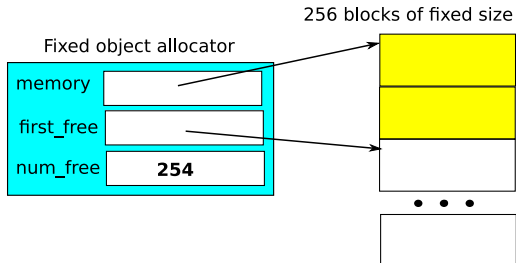
- ▶ wykrywanie błędów
- ▶ poprawa efektywności (::new jest ogólnego użytku)
- ▶ statystyki na temat wykorzystania pamięci

```
class X {  
    public:  
        //Własna wersja operatora new  
        static void* operator new(std::size_t size) throw(std::bad_alloc){  
            //np. rejestracja  
            return ::operator new(size); //woła funkcję globalną  
        }  
        static void //Własna wersja operatora delete  
        operator delete(void* raw_memory, std::size_t size) throw ();  
};
```

**Uwaga!** własny przydział pamięci bardzo trudny do implementacji

# Przykład wykorzystania: small object allocator

- ▶ częste przydziały i zwalnianie małych obiektów o tej samej wielkości
- ▶ fragmentacja pamięci
- ▶ znaczący narzut danych pomocniczych



- ▶ `std::pmr::unsynchronized_pool_resource` (C++17)
- ▶ Boost.Pool
- ▶ [loki-lib.sourceforge.net](http://loki-lib.sourceforge.net)

# Biblioteka STL (standardowa w C++ od 1997)

## Skład:

- ▶ kontenery(vector, list,...)
- ▶ iteratory
- ▶ algorytmy
- ▶ funktory
- ▶ adaptery
- ▶ alokatory

## Cechy:

- ▶ wykorzystuje mechanizm szablonów
  - ▶ kod wielokrotnego użycia
  - ▶ można stosować do typów wbudowanych
- ▶ wykorzystuje wzorzec iteratora (zmniejsza liczbę algorytmów)
- ▶ bezpieczna w aplikacjach wielowątkowych
- ▶ bardzo efektywna

# Biblioteka STL - alokatory

Kontenery umożliwiają dostarczenie własnej obsługi pamięci:

```
template<class T,  
        class Allocator = std::allocator<T> > class vector;
```

Biblioteka standardowa dostarcza kilka alokatorów

```
#include <memory_resource> //od C++17
```

```
std::pmr::unsynchronized_pool_resource //alokator dla małych obiektów
```

```
std::pmr::synchronized_pool_resource //wersja współbieżna
```

```
//zarządza dostarczonym buforem
```

```
//przy żądaniu dodatkowej pamięci zgłasza bad_alloc
```

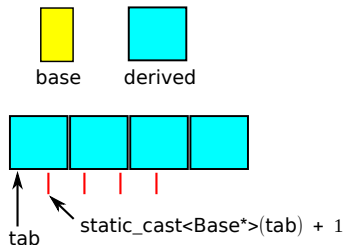
```
std::pmr::null_memory_resource
```



# Podwójna rola wskaźników

- ▶ identyfikator obiektu
- ▶ iterator dla tablicy (arytmetyka wskaźników)

```
class Base { /* ... */ };  
class Derived : public Base  
{ /* ... */ };  
void f(base* tab, int size) {  
    tab[1];  
};  
void g(base* b);  
//tablica obiektów  
derived tab[N];  
  
f(tab, N); //Niebezpieczne!  
g(&tab[1]); //OK
```



*Dziękuję*