Programowanie obiektowe

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 25.10.2019

1. Wprowadzenie
Wskaźniki i referencje

Języki obiektowe

- Simula – pierwszy język obiektowy, 1967
- Smalltalk – 1980
- **C++ – 1985**
- Java – 1995
- C# – 2001
- Python – 1995
- Ruby
- Visual Basic .NET
- Objective-C (język programowania dla iOS)
- Curl
- Delphi
- Eiffel

Literatura

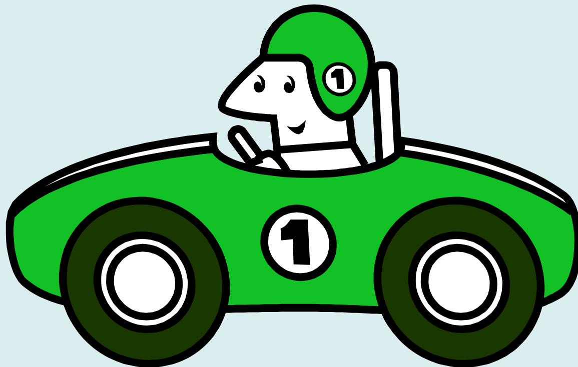
- Bjarne Strostrup, Język C++
- Bruce Eckel, Thinking in C++
  [także dostępne w sieci w jęz. angielskim]
- http://www.cplusplus.com/reference/
- J. Grębosz, Symfonia C++
- J. Grębosz, Pasja C++

Programowanie obiektowe 1

- Języki programowania pozwalają na modelowanie rzeczywistego problemu, który powinien być rozwiązany z użyciem oprogramowania.
- Charakter języka określa jakość i rodzaj modelu. Wiele języków stanowi w rzeczywistości wygodny interfejs do architektury komputera, na którym będzie uruchamiany program (asembler, FORTRAN, C, BASIC, Pascal)

Programowanie obiektowe 2

- Programowanie obiektowe dostarcza narzędzi do bezpośredniej reprezentacji elementów pojawiających się w rozwiązywanym problemie w postaci konstrukcji języka programowania.

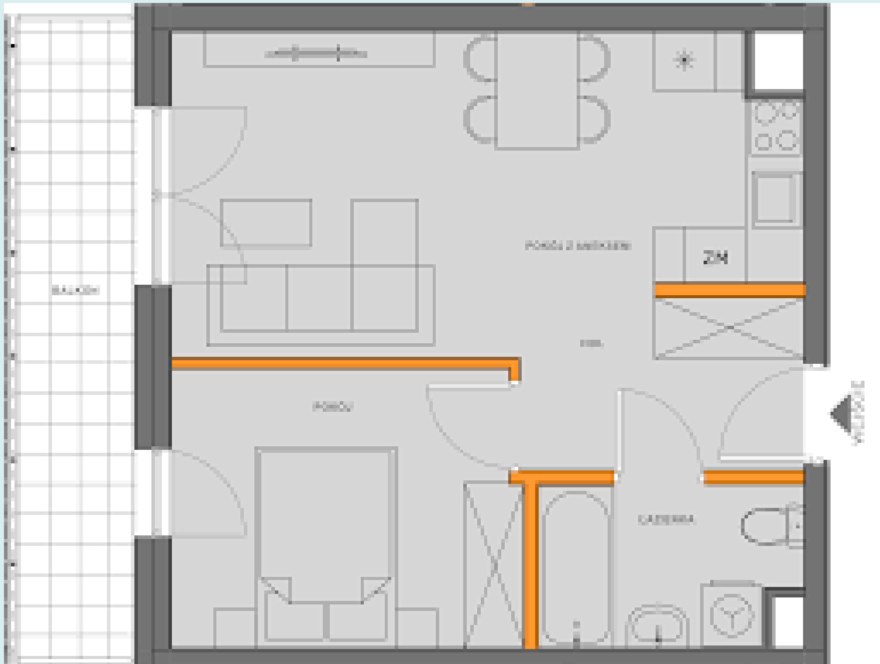



```
class Driver {

};
```

```
class Car{
public:
    void drive(Driver&d);

};
```

Programowanie obiektowe 3

- Inny przykład



```
class Room{
    double area;
public:

};
```

```
class Appartment{
    list<Room> rooms;
public:
    double calculateArea();

};
```

Trzy etapy konstrukcji programu

- **Analiza obiektowa** – identyfikuje obiekty występujące w świecie rzeczywistym, który ma być modelowany przez program (np.: osoby, instytucje, urządzenia, dokumenty, itd.)
- **Projektowanie obiektowe** odwzorowuje te obiekty w elementy platformy oprogramowania (klasy, obiekty, rekordy bazy danych) a także dodaje elementy specyficzne dla danej platformy (okna, okna dialogowe, elementy interfejsu użytkownika).
- **Programowanie obiektowe** obejmuje implementację rezultatów projektowania za pomocą wybranego obiektowego języka programowania.

W rezultacie elementy występujące w rzeczywistym problemie mogą zostać bezpośrednio odwzorowane na elementy występujące w języku programowania. Konstruując program można „myśleć” w kategorii dziedziny problemu i elastycznie dostosowywać go do dziedziny.

Założenia czystego podejścia obiektowego

Każdy element programu jest obiektem

- Dowolny element dziedziny może zostać zaimplementowany jako obiekt i włączony do programu.
- Obiekt może być traktowany jak zmienna przechowująca dane. W idealnej postaci, dane te są ukryte.
- Aby zmodyfikować obiekt lub odczytać jego dane wysyłamy do obiektu odpowiednie polecenie. W odpowiedzi obiekt wykonuje operacje na swojej zawartości.

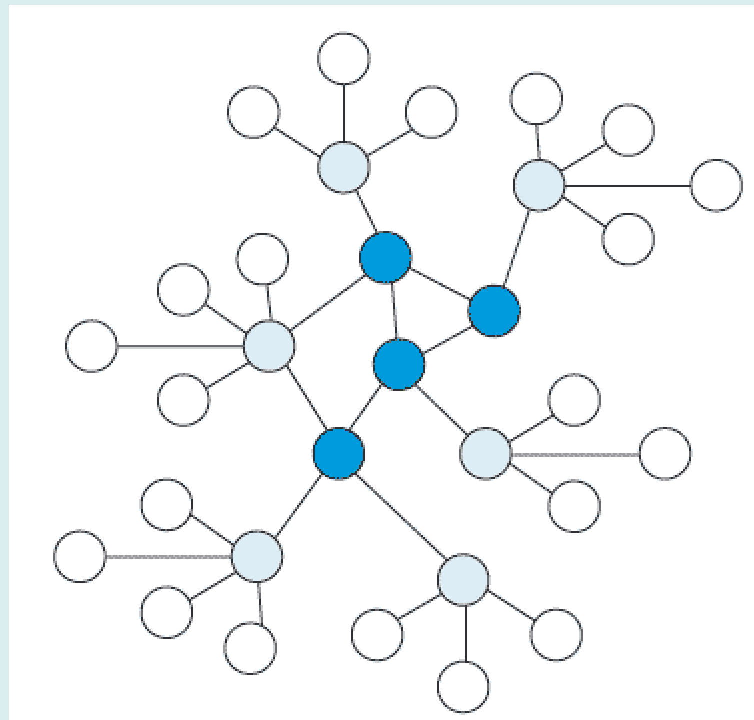
```
class Room{
    double area;
public:
    double getArea(){return area;}
};
```

```
class Appartment{
    list<Room> rooms;
public:
    double calculateArea();

};
```

Program jest siecią obiektów, które nawzajem przesyłają do siebie komunikaty

Aby skierować polecenie do obiektu wysyłamy do niego polecenie za pomocą komunikatu. W języku C++ wysłanie komunikatu jest utożsamiane z wywołaniem funkcji.

Implementacje obiektowe dostarczają także narzędzi do konstrukcji rozproszonych programów, gdzie poszczególne obiekty rezydują na różnych maszynach Wówczas komunikat jest rzeczywistym komunikatem sieciowym



Każdy obiekt ma własną pamięć, na którą składają się inne obiekty

Aby stworzyć nowy rodzaj obiektu możemy **zgrupować** obiekty niższego typu (także zmienne proste). Pozwala to ukryć złożoność modelowanego problemu.

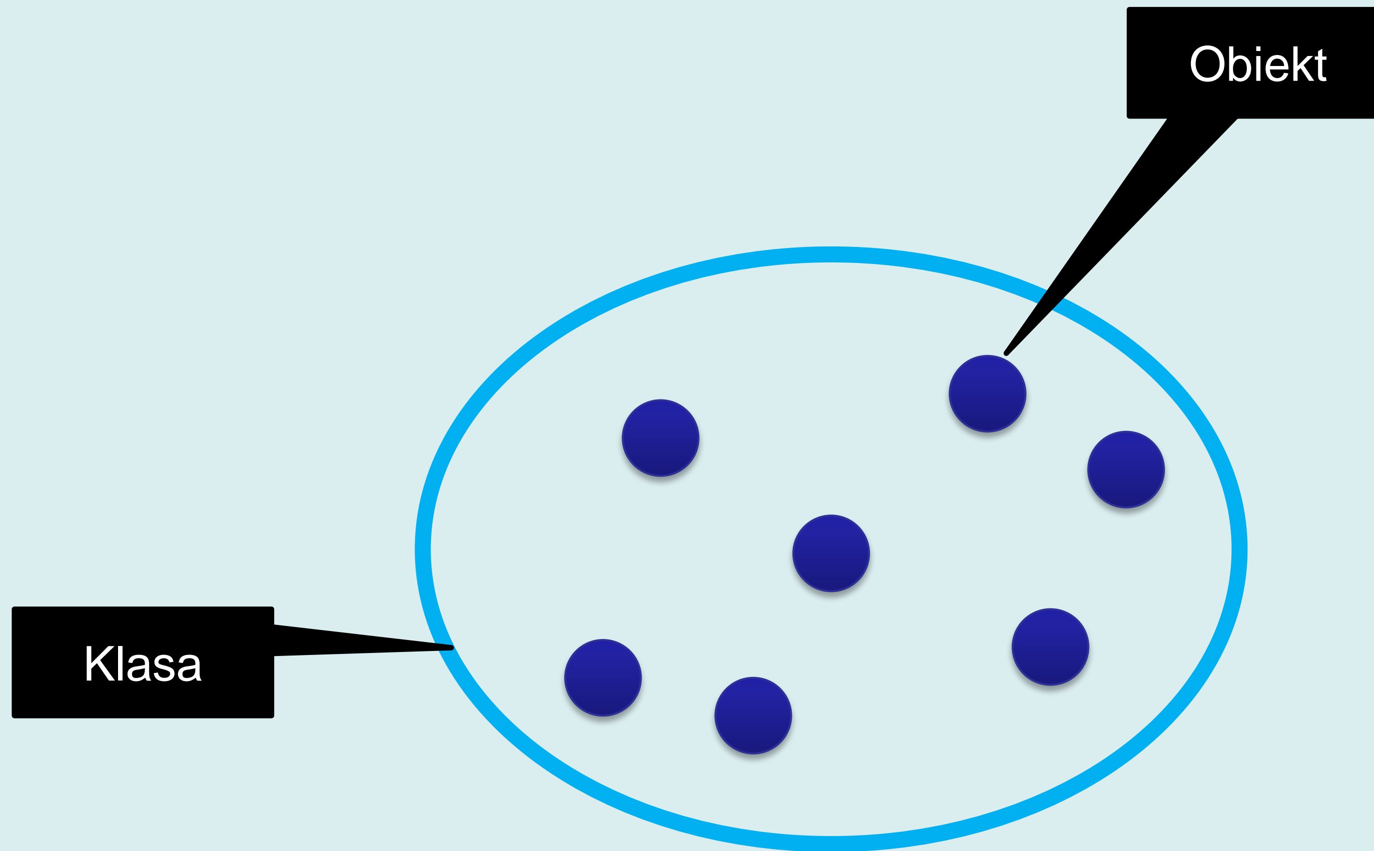
```
class Pracownik{
public:
    string imie;
    string nazwisko;
    string pesel;
    Adres adres;
    string stanowisko;
};
```

```
class Adres{
    string miescowosc;
    string ulica;
    string nr_domu;
    string nr_lokalu;
    int kod;
};
```

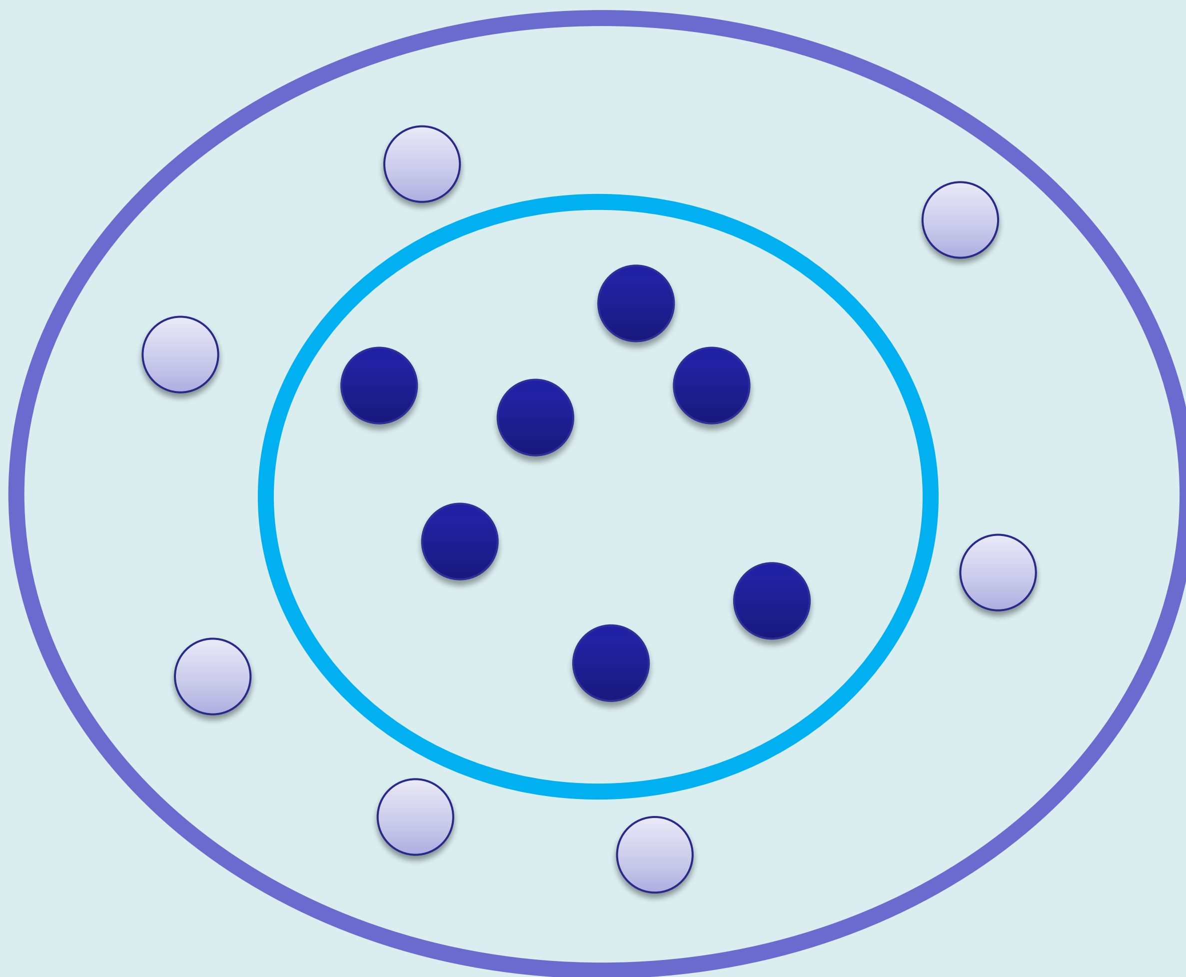
```
class Firma{
public:
    string nazwa;
    Adres adres;
    string nip;
    list<Pracownik> pracownicy;

};
```

Każdy obiekt ma typ

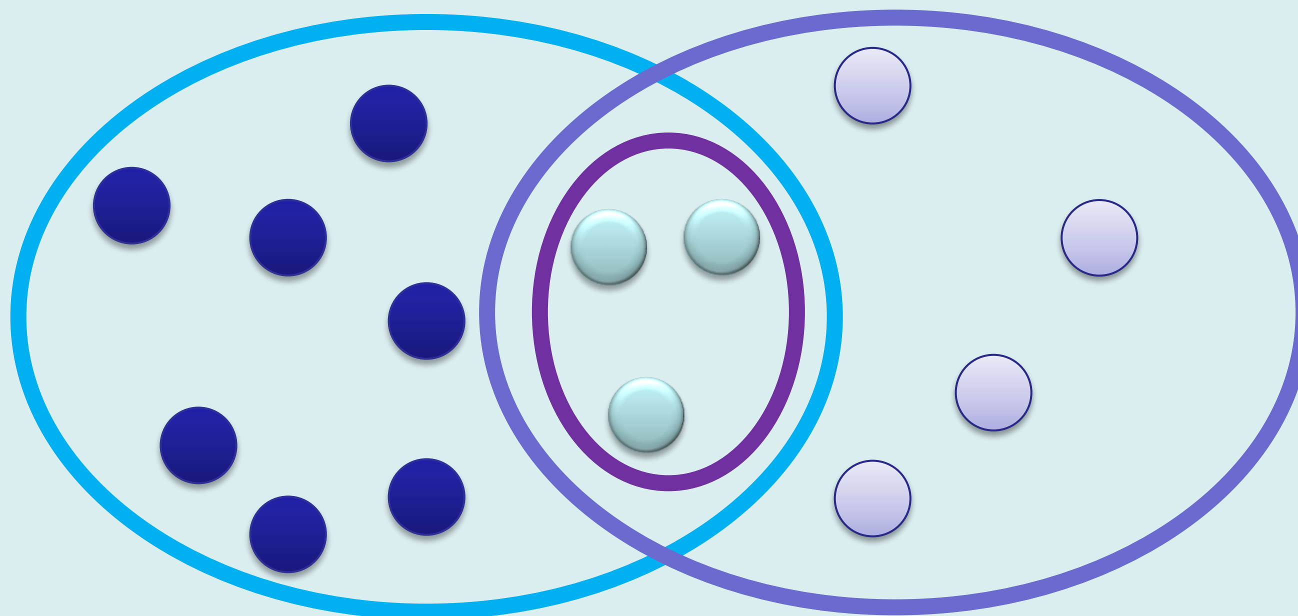
Każdy obiekt należy do pewnej *klasy*. Klasa jest tu synonimem typu.



Obiekty mogą równocześnie należeć do wielu klas 1

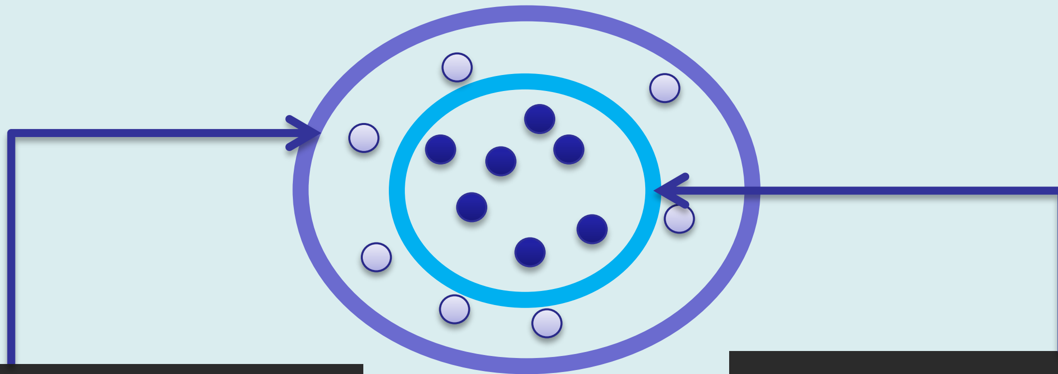


Obiekty mogą równocześnie należeć do wielu klas 2



Wszystkie obiekty danego typu mogą przyjmować te same komunikaty

Z danym typem można związać jego interfejs – zbiór komunikatów, które obiekt może przyjmować.



```cpp
class Shape{
    Color color;
public:
    virtual void draw();
    virtual void erase();
    Color getColor();
    void setColor(Color);
};
```

```cpp
class Circle : public Shape
{
public:
    double xc,yc;
    double radius;
    void draw();
    void erase();

};
```

Klasy

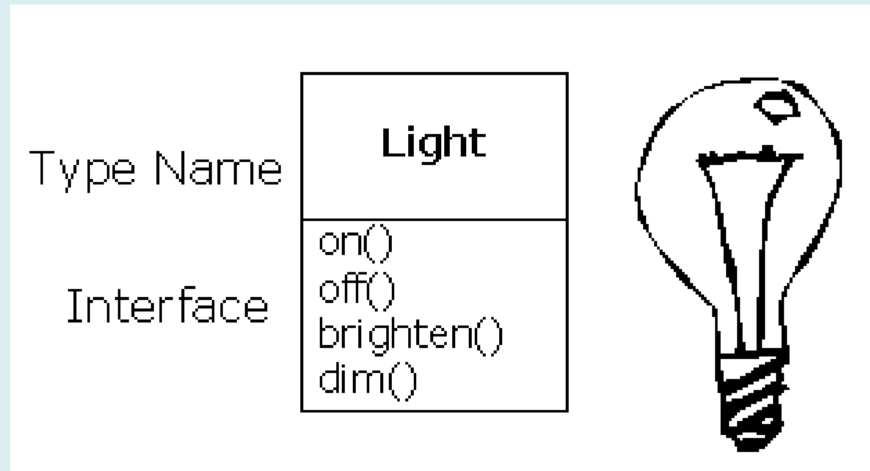
- Klasa określa wspólne własności i zachowanie należących do niej obiektów.
- Obiektowe języki programowania pozwalają na:
  - definiowanie klas
  - tworzenie obiektów danej klasy (instancji klasy)

Definiowanie klas 1

Definicja klasy obejmuje:

- *komponenty* klasy (elementy składowe)
- *interfejs* klasy (zbiór komunikatów, które obiekt może przyjmować) oraz
- *implementację* klasy, czyli opis, w jaki sposób obiekt danej klasy będzie reagował w odpowiedzi na wysyłane komunikaty.

Definiowanie klas 2




```
class Light
{
    double voltage;
public:
    void on(){voltage = 230;}
    void off(){voltage = 0;}
    void brighten(){if(voltage<=220) voltage+=10;}
    void dim(){if(voltage>=10) voltage-=10;}
};
```

Instancje klasy

- Obiekty należące do danej klasy tworzone są podobnie jak zmienne innych typów.
- Wysyłanie komunikatów realizowane jest poprzez wywołanie funkcji należącej do obiektu.

```
Light lt;
lt.on();
lt.dim();
lt.off();
```

Obiekt utworzony na stosie

Wywołanie metody

```
Light*plt = new Light();
plt->on();
plt->dim();
plt->brighten();
delete plt;
```

Obiekt utworzony na stercie

Kontrola dostępu 1

Specyfikacja klasy obejmuje:

- **atrybuty** (zmienne, pola składowe)
- **metody** (funkcje składowe)

Dwie role użytkowników klasy

- **Twórca klasy** projektuje interfejs klasy i go implementuje
- **Użytkownik końcowy** klasy (klient) wykorzystuje gotową klasę w swojej aplikacji

Kontrola dostępu 2

Część atrybutów i metod ma charakter **ogólnie dostępnego interfejsu**, natomiast część pełni **funkcję pomocniczą** przy implementacji klasy.

Zazwyczaj twórca klasy stara się ukryć jej pomocnicze elementy:

- aby uchronić się przed błędami powstałymi w skutek **nieodpowiedniego dostępu**.
- tworząc bibliotekę klas ustalić i zawsze realizować pewien **publiczny interfejs**, natomiast dowolnie zmieniać **część prywatną implementacji** w kolejnych wersjach biblioteki

Słowa kluczowe określające dostęp

public	pola i metody składowe tego typu są ogólnie dostępne
private	pola i metody składowe tego typu nie są dostępne z zewnątrz
protected	pola i metody nie są dostępne z zewnątrz, natomiast są dostępne dla obiektów klas potomnych, dziedziczących po danej klasie.

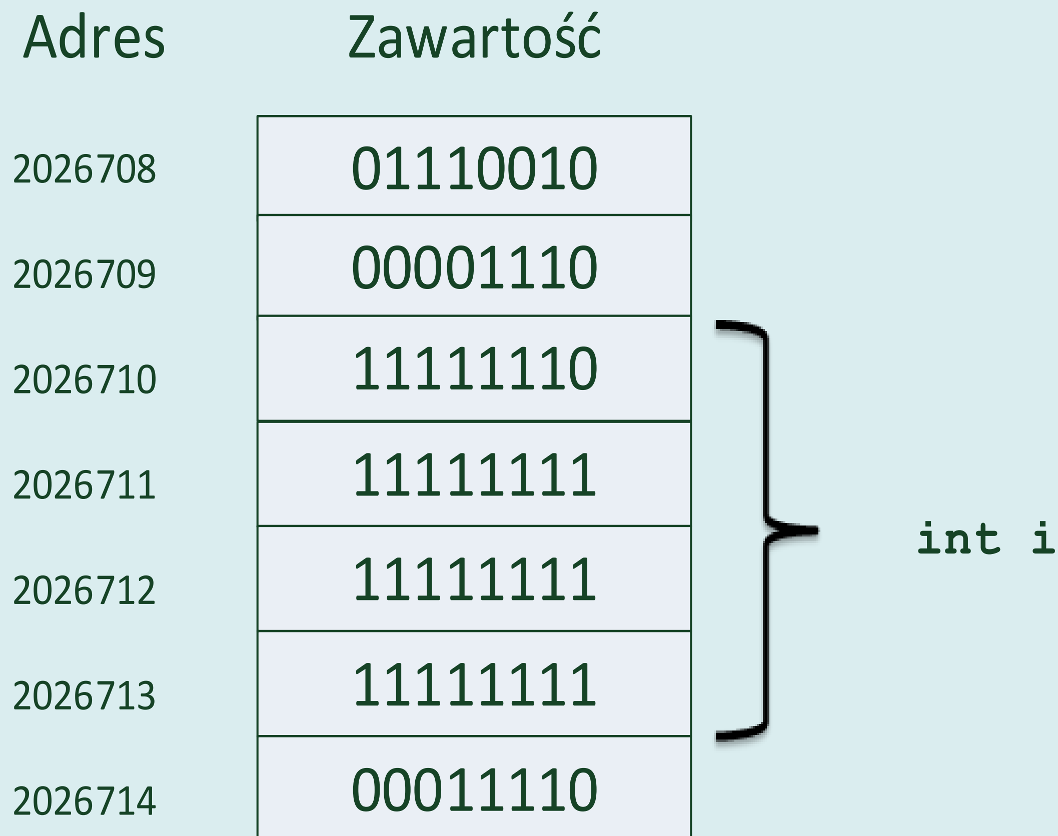
W klasie można zadeklarować klasy i funkcje poprzedzone słowem kluczowym `friend`. Nazywane są klasami (funkcjami) zaprzyjaźnionymi.

Klasy (funkcje) zaprzyjaźnione mają dostęp do prywatnych i chronionych składowych klas (pól i metod).

Wskaźniki
Przypomnienie języka C

Wskaźniki – wprowadzenie (1)

- Podczas wykonania programu wszystkie jego elementy (zmienne, wartości stałych, funkcje) umieszczone są w pamięci.
- Każdy z nich ma adres będący nieujemną liczbą całkowitą
- Adres jest pojęciem niskopoziomowym. Adresy są argumentami rozkazów procesora.

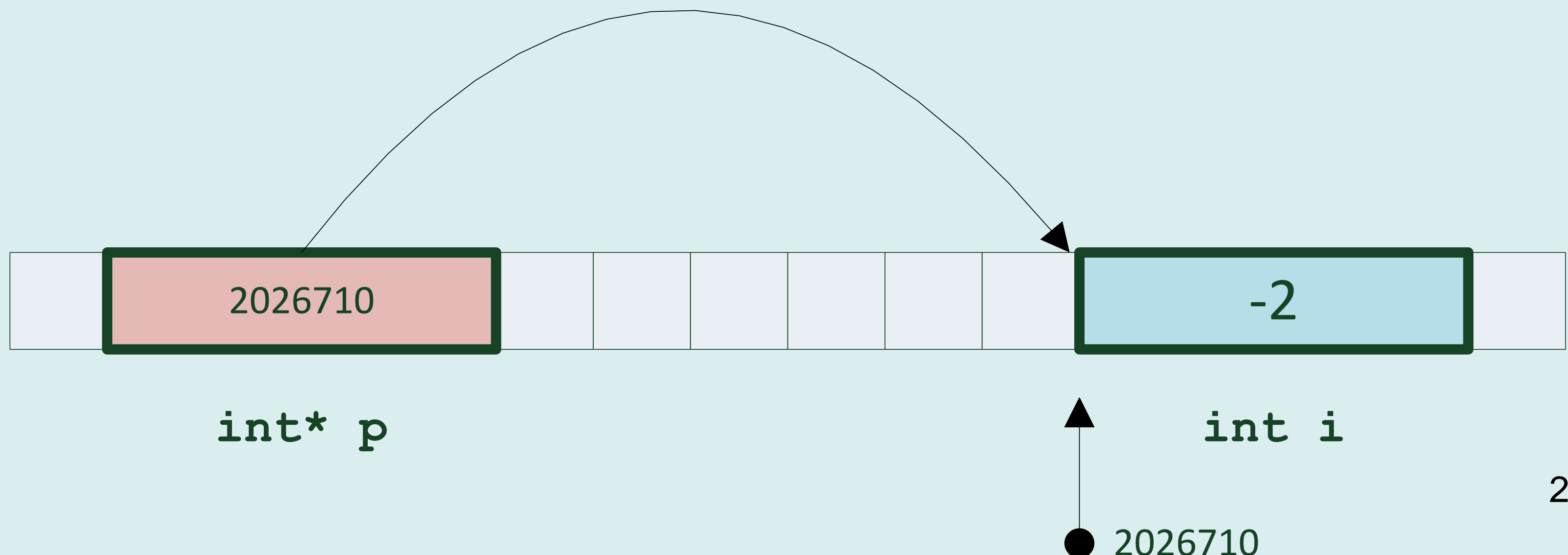


Wskaźniki – wprowadzenie (2)

Wskaźniki są to zmienne, których wartościami są adresy. Korzystając ze wskaźników możemy:

- odczytać lub zmodyfikować wartość zmiennej zajmująca pamięć identyfikowaną przez adres
- wywołać funkcję

Zmienne wskaźnikowe **mają określone typy**. Informacje o typie są uzupełnieniem informacji o adresie. Dzięki znajomości typu kompilator może określić ile bajtów zajmuje wskazywany element i w jaki sposób należy interpretować dane (np.: jako `float` albo `int`).



Wskaźniki - deklaracje

Składnia deklaracji:

`type-specifier * pointer`

`type-specifier`

definiuje typ wskazywanego obiektu

`pointer`

identyfikator zmiennej

```
int *pi, tab[10];
double *pd;
float*px,*py,x,y;
```

Operatory adresu i dereferencji (1)

Język C definiuje dwa operatory umożliwiające posługiwanie się wskaźnikami:

- Jednoargumentowy operator adresu **&** (ang. *address operator*)

```
int x;
int*px;
px=&x;
printf("%p ",&x);
```

- Dereferencji ***** (ang. *dereference*, *indirection operator*)

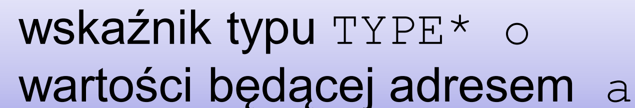
```
*px=7;
printf("%d ",*px+3)
```

Operatory adresu i dereferencji (2)

Operator adresu &

- Operator adresu & pobiera adres obiektu będącego jego argumentem i zwraca wskaźnik zgodny z typem argumentu;
- Argumentem operatora adresu musi być obiekt, który ma przypisaną lokalizację w pamięci (zmienna, identyfikator funkcji).
- Nie można pobrać adresów zmiennych rejestrowych lub pól bitowych.

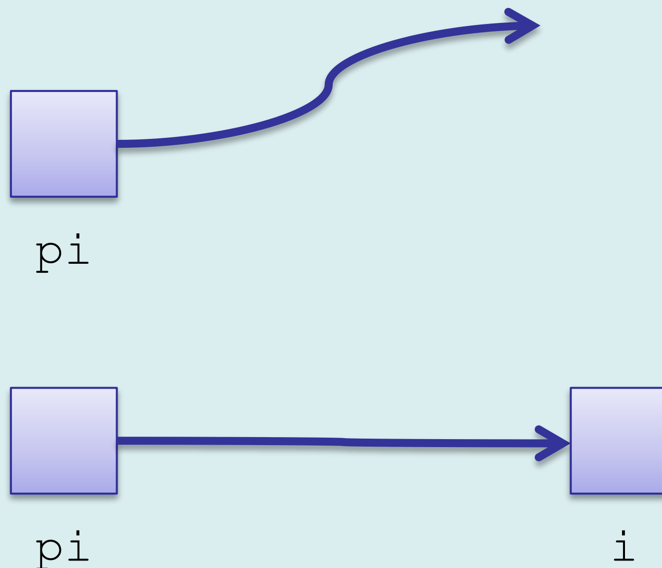
```
TYPE a;
&a;
```



Operatory adresu i dereferencji (3)

Deklaracja zmiennej wskaźnikowej przydziela dla niej pamięć, ale wskaźnik nie musi wskazywać jakiegokolwiek obiektu.

```
int *pi;
int i;

pi=&i;
```



Deklarując wskaźnik można nadać mu wartość będącą adresem istniejącego obiektu

```
int i,*pi=&i;
```

Operatory adresu i dereferencji (4)

Operator dereferencji *

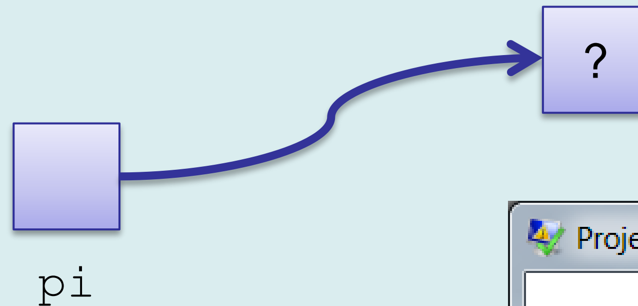
- W specyfikacji języka C terminem *obiekt* określany jest obszar pamięci, którego zawartość może być odczytywana/modyfikowana.
- *Lvalue* to wyrażenie identyfikujące taki obiekt. (Rozróżnienie lvalue i rvalue pochodzi z definicji operatora przypisania *lvalue = rvalue)*
- Operator dereferencji zwraca *lvalue* – wyrażenie identyfikujące wskazywany obiekt (mieszczący się pod wskazanym adresem)
- Typ argumentu określa typ zwracanego wyrażenia: jeżeli wskaźnik jest typu `TYPE*` zwracane wyrażenie jest typu `TYPE`

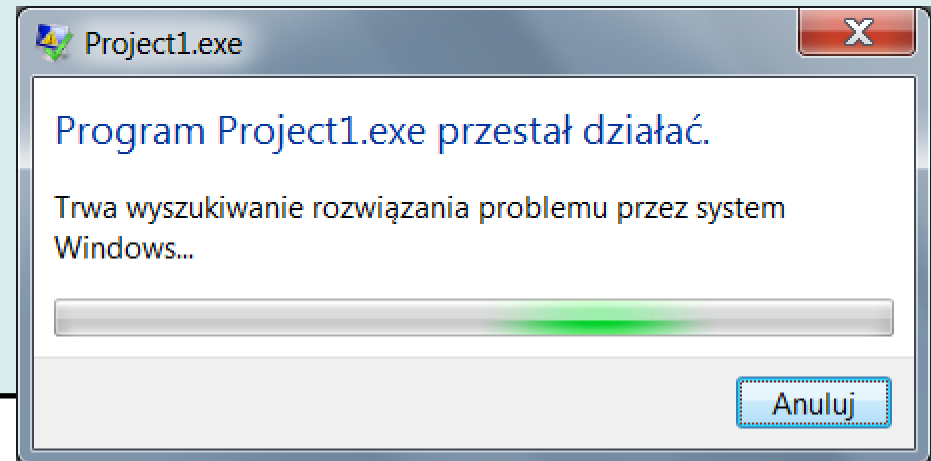
```
int i=7,j,*pi=&i;
printf("%d ",*pi);

*(int*)20000=7;
// ale nie *20000

j=*&i;  // znoszące się operatory
// ale nie j=&*i;
j=(int)&*(char*)i; //ok
```

zmienna typu `int` mieszcząca się pod adresem 20000

Operatory adresu i dereferencji (5)

Nigdy nie należy stosować operatora dereferencji do niezainicjowanych zmiennych (albo mających takie wartości jak 0 lub NULL) …



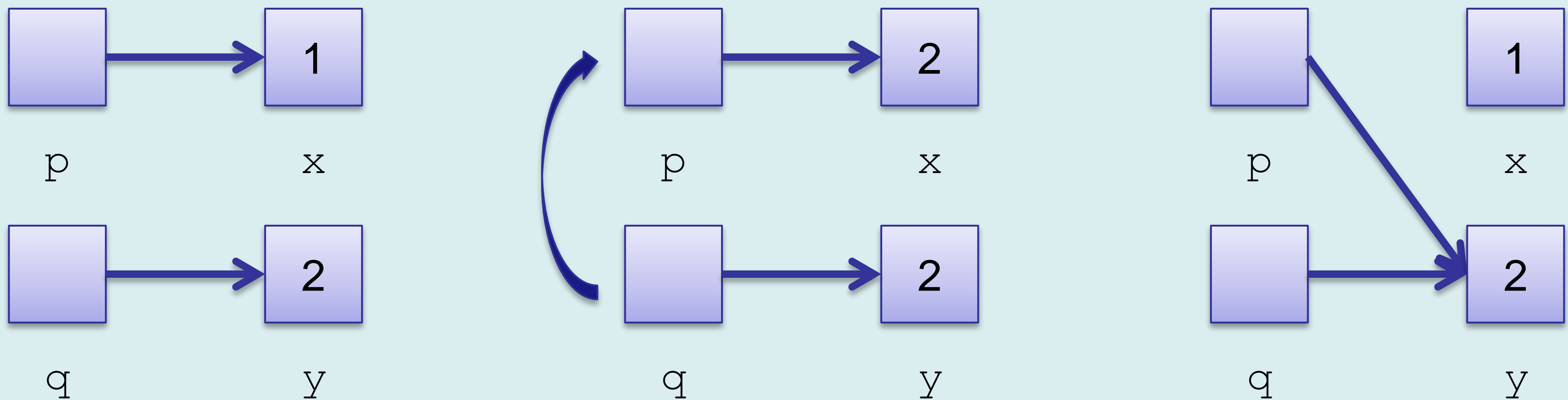



```
int *pi,*pj=NULL;
printf("%d ",*pi);
*pj=7;
```

Operatory adresu i dereferencji (5)

Przypisania:

```
int x=1, *p = &x, y=2, *q = &y;

*p=*q; // równoważne x = y
       // spełniona jest równość *p==*q
x=1;
p=q;   // wskaźniki wskazują ten sam obiekt (y)
       // spełnione jest p==q i *p==*q
```



Dostęp do pól struktur i unii (1)

Operator kropkowy dostępu do pól struktur ma większy priorytet niż operator dereferencji.

```
struct complex {double re; double im;};
struct complex vx={1,0};
struct complex *pc=&vx;
printf("(%f,%f)",*pc.re,*pc.im);
```

21 main.c request for member `re' in something not a structure or union
21 main.c request for member `im' in something not a structure or union

Rozwiązania:

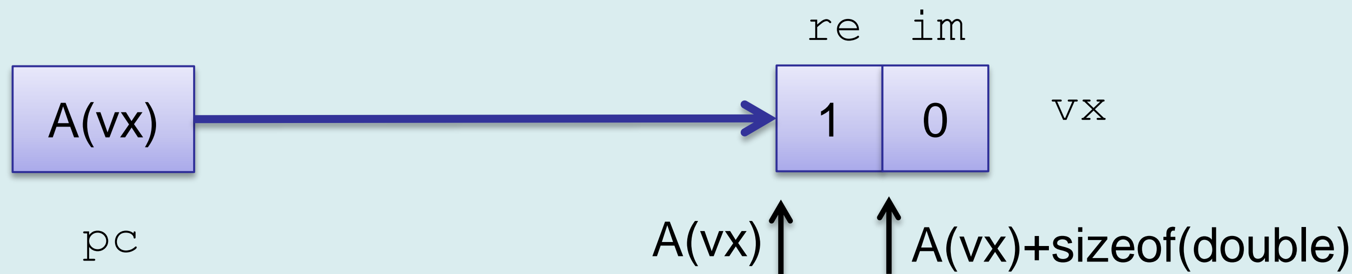
- Można zastosować nawiasy `(*pc).re`
- lub specjalny operator `->` : `pc->re`

```
printf("(%f,%f)",(*pc).re,(*pc).im);
pc->re=0;pc->im=1;
```

Dostęp do pól struktur i unii (2)

```
struct complex {double re; double im;};
struct complex vx={1,0};
struct complex *pc=&vx;
```

- **`(*pc)`** to *lvalue* identyfikująca zadeklarowaną wcześniej zmienną **`vx`**;
- **`pc->im`** to również wyrażenie *lvalue* równoważne **`(*pc).im`** oraz **`vx.im`**
- W wygenerowanym kodzie kompilator posługuje się adresami. (Zapewne pole `re` ma adres początku struktury, natomiast pole `im` adres przesunięty o 8B)



Zastosowania wskaźników (1)

Podstawowe zastosowania wskaźników to:

- Możliwość modyfikacji obiektu zdefiniowanego na zewnątrz funkcji
- Ustalanie powiązań pomiędzy obiektami

Inne zastosowania to:

- Zarządzanie danymi tworzonymi dynamicznie (tablicami, listami, drzewami)
- Realizacja polimorfizmu w C++

Zastosowania wskaźników (2)

Modyfikacja zewnętrznych obiektów

- Standardowo, zmienne przekazywane są do funkcji *przez wartość*. Oznacza to, że wartością parametru funkcji jest kopia argumentu. Działania na parametrze funkcji nie modyfikują oryginalnego obiektu.
- Jeżeli do funkcji przekazany zostanie wskaźnik zawierający adres zewnętrznego obiektu, możliwa jest modyfikacja jego zwartości.

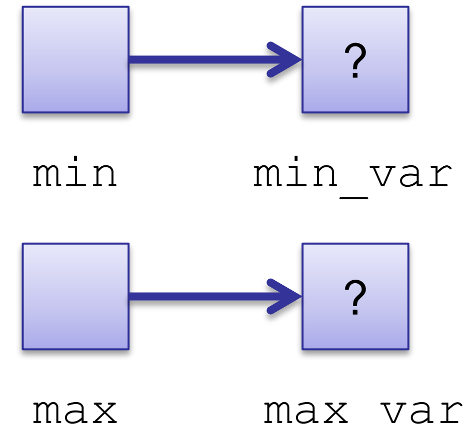
```
void foo(int * x){
        (*x)++;
        printf("x in foo=%d\n",*x);
}

int main(){
        int x = 2;
        printf("x in main=%d\n",x);
        foo(&x);
        printf("x in main=%d\n",x);
        return 0;
}
```

```
x in main=2
x in foo=3
x in main=3
```

Zastosowania wskaźników (3)

Przykład – obliczanie minimalnej i maksymalnej wartości elementu tablicy

```
void min_max(int tab[],int n,int*min,int*max){
     int i;
     *max=*min=tab[0];
     for(i=0;i<n;i++){
          if(*min>tab[i])*min=tab[i];
          if(*max<tab[i])*max=tab[i];
     }
}

int main()
{
    int min_var, max_var;
    int t[]={3,4,7,2,-2,234};
    min_max(t,sizeof t/sizeof t[0],&min_var,&max_var);
    printf("min = %d max = %d\n",min_var, max_var);
    return 0;
}
```




```
min = -2 max = 234
```

Zastosowania wskaźników (4)

Przykład – zbiór funkcji działających na strukturze complex

W języku C często wykorzystuje się wskaźniki przy tworzeniu bibliotek funkcji działających na określonych typach danych.

```
struct complex {double re,im;} ;

void init(struct complex*pc,double x,double y) {
       pc->re=x;
       pc->im=y;
}

void add(struct complex*c,
                    struct complex*a, struct complex*b){
       c->re=a->re + b->re;
       c->im=a->im + b->im;
}

double module(struct complex*c) {
       return(sqrt(c->re * c->re + c->im * c->im));
}
```

Zastosowania wskaźników (5)

Przykład – kontynuacja

```
void dump(struct complex*c) {
     printf("(%f, %f) ",c->re,c->im);
}
int main()
{
    struct complex c1,c2,c3;
    init(&c1,12.34,1.5);
    dump(&c1);
    init(&c2,-12.34,1.5);
    dump(&c2);
    add(&c3,&c1,&c2);
    dump(&c3);
    return 0;
}
```

```
(12.340000, 1.500000) (-12.340000, 1.500000)
(0.000000, 3.000000)
```

Referencje

Referencje w C++

Referencje w języku C++ są typem, który bardzo przypomina wskaźniki. Podobnie, jak w przypadku wskaźników, wartościami zmiennych typu referencyjnego są adresy, a także referencja przechowuje informacje o typie wskazywanego obiektu.

- W odróżnieniu od wskaźników, przy korzystaniu z referencji nie stosuje się operatora *dereferencji* (*). Jest on wywoływany automatycznie.
- Referencji nie można przestawiać, tak aby wskazywała inny obiekt.

Referencje w C++

Za pośrednictwem referencji możemy:

- odczytać lub zmodyfikować wartość (atrybuty) obiektu zajmującego pamięć identyfikowaną przez adres
- wywołać metodę obiektu.

Składnia deklaracji:

```
type-specifier & reference
```

`type-specifier`

definiuje typ wskazywanego obiektu

`reference`

identyfikator zmiennej

Referencje w C++

Referencje mogą być bezpośrednio używane jako zmienne:

```
int x=7;
int&r1 = x; // (1)
const int&r2 = 12; // (2)
r1++; // (3)
printf(”r1=%d, r2=%d”,r1,r2);
```

- Instrukcja (1) deklaruje referencję inicjując ją adresem obiektu x.
- Instrukcja (2) alokuje pamięć dla zmiennej typu int, inicjuje ją wartością 12 oraz deklaruje referencję, która wskazuje to miejsce.
- Wszelkie operacje na referencjach są w rzeczywistości operacjami na obiektach wskazywanych przez referencje. Instrukcja (3) zwiększy wartość zmiennej x.

Referencje w C++

- Referencje są najczęściej używane jako argumenty funkcji. Podobnie jak w przypadku wskaźników, obiekty przekazywane są przez adres.

```
void min_max(int tab[],int n,int&min,int&max){
    int i;
    max=min=tab[0];
    for(i=0;i<n;i++){
        if(min>tab[i])min=tab[i];
        if(max<tab[i])max=tab[i];
    }
}
```

Referencje w C++

- Referencje mogą być używane jako wartości zwracane przez funkcje. Najczęściej są to funkcje składowe obiektu i zwracają referencję do obiektu, do którego należą.

```
class A
{
public:
    A&foo() {
        //...
        return *this;
    }
};
```

Referencje w C++

- Referencje mogą być bezpośrednio używane jako pola klas, ale wymagają inicjalizacji poprzez **listę inicjalizacyjną** konstruktora klasy

```
class A
{
    int&r;
public:
    A(int a):r(a){}
};
```

Referencje w C++

- Podstawową różnicą pomiędzy referencjami i wskaźnikami jest to, że wartością referencji musi być adres istniejącego obiektu. Wartość referencji jest ustalana w momencie inicjalizacji i jest to statycznie sprawdzane przez kompilator.
- **Wartością referencji nie może być 0 (NULL).**

```
int &r1;     // błąd r1 nie wskazuje żadnego obiektu
int x=5;
int &r2=x;   // OK. r2 wskazuje zmienną x
```