PROGRAMOWANIE OBIEKTOWE

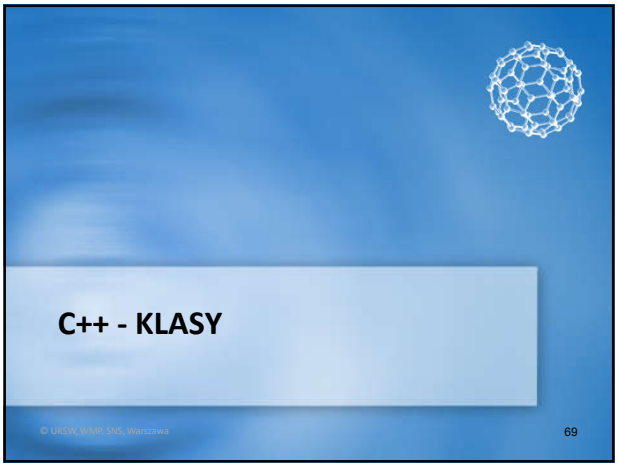
67

Programowanie obiektowe

- Język programowania C++ pozwala na obiektową realizację projektu systemu
- Język C++ jest nadzbiorem języka C

  (*prawie..*)
- Występują różnice między C i C++ w obszarze właściwości programowania strukturalnego, jednak program napisany wg reguł C w większości przypadków zostanie zaakceptowany przez kompilator C++.

68



C++ - KLASY

69

C++ - klasy

- C++ jest językiem programowania obiektowego – pozwala na budowaniu programów działających na obiektach.
- Obiekty, to egzemplarze/instancje utworzone wg opisu zawartego w definicji typu.
- Typy obiektowe w C++ są rozszerzeniem typów używanych do tworzenia zmiennych w C
- Rozszerzenie polega na:
  - dodaniu reguł dostępu do składników/atrybutów/pól obiektu
  - dodaniu właściwości usługowych – metod, które mogą wykonywać działania wykorzystując składniki obiektu
  - dodaniu usługi inicjalizacji przy tworzeniu i usługi kończenia przy usuwaniu obiektów

70

C++ - klasy

Deklarowanie klasy przypomina deklarowanie typu strukturalnego:

```
struct osoba1
{

  char imie[20];
  char nazwisko[30];
  int wiek;
};
```

```
class osoba2
{
public :
  char imie[20];
  char nazwisko[30];
  int wiek;
};
```


71

C++ - klasy

Deklaracja instancji:

```
osoba1 PanX;
osoba2 PanY;
```

Odwołanie:

```
strcpy(PanX.imie,"Mel");
strcpy(PanX.nazwisko, "Gibson");
PanX.wiek = 58;
strcpy(PanY.imie, "Danny");
strcpy(PanY.nazwisko, "Glover");
PanY.wiek = 68;
```

Czym różnią się te dwa obiekty?

72

C++ - klasy

Projektując klasę należy odpowiedzieć sobie na kilka pytań:

1. Co charakteryzuje stan wewnętrzny obiektów i jak go reprezentuje?
2. Czy można definiować jakieś niezmienniki stanu wewnętrznego obiektów?
3. Które z cech obiektu można udostępniać publicznie, a które powinny być kontrolowane?
4. Jak będą tworzone i inicjowane obiekty; czy dopuszczamy istnienie obiektów z nieokreślonym stanem wewnętrznym?
5. Czy likwidacja obiektu wymaga czynności porządkowych?
6. Jakie operacje będą wykonywane na obiektach?
7. Jak program ma korzystać z definicji klasy?

C++ - klasy

Odpowiedzi dla klasy reprezentującej *butelkę mleka*:

1. *Co charakteryzuje stan wewnętrzny obiektów i jak go reprezentuje?*
   - Maksymalna pojemność butelki i jej zawartość.
2. *Czy można definiować jakieś niezmienniki stanu wewnętrznego obiektów?*
   - Zawartość niemniejsza od zera i nie większa od pojemności.
3. *Które z cech obiektu można udostępniać publicznie, a które powinny być kontrolowane?*
   - Pojemność musi pozostać niezmienna, zawartość może być modyfikowana tylko tak, aby zachować niezmienniki.
4. *Jak będą tworzone i inicjowane obiekty; czy dopuszczamy istnienie obiektów z nieokreślonym stanem wewnętrznym?*
   - Tworzony obiekt – butelka mleka musi być na początku pusta (zawartość równa zero). Nie ma butelek z niekreślonym stanem wewnętrznym.

C++ - klasy

Odpowiedzi dla klasy reprezentującej *butelkę mleka*:

5. *Czy likwidacja obiektu wymaga czynności porządkowych?*
   - Jeżeli w butelce jest mleko, to należy najpierw je wylać.
6. *Jakie operacje będą wykonywane na obiektach*?
   - Wlać mleko i wylać mleko. Można jeszcze umyć, wstawić kwiatki ..
7. *Jak program ma korzystać z definicji klasy?*
   Może tworzyć obiekty, a może to być tylko klasa bazowa (o tym później).

C++ - klasy

Prawa dostępu:

- **public** - składniki są dostępne wszędzie gdzie jest użyty obiekt danej klasy
- **private** - składniki są dostępne tylko wewnątrz danej klasy. W praktyce oznacza to, że tylko metody danej klasy mogą czytać i zapisywać tego typu składniki
- **protected** – w porównaniu z *private*, zakres widoczności jest poszerzony o klasy wywodzące się z aktualnej klasy, tzw. klasy dziedziczące (*to zostanie wyjaśnione przy okazji omawiania dziedziczenia*)

Po co komu prawa dostępu?

C++ - klasy

kapsułkowanie/hermetyzacja

ukrywaniu pewnych danych składowych lub metod obiektów danej klasy tak, aby były one dostępne tylko metodom wewnętrznym danej klasy lub funkcjom z nią zaprzyjaźnionym (*o funkcjach zaprzyjaźnionych będzie jeszcze mowa*).

Kapsułkowanie uodparnia tworzony model systemu na błędy nieprawidłowego zapisu lub niepowołanego odczytu składowych klasy (*kto te błędy może popełnić? – nieumiejętny programista*)

Kapsułkowanie nie ma sensu w przypadku struktur w C, bo struktury nie mają własnych metod, stąd kapsułkowanie zamknęłoby całkowicie dostęp do składowych struktury

C++ - klasy

- Abstrakcja proceduralna i hermetyzacja to fundamentalne założenia modelu obiektowego: na danym poziomie szczegółowości użytkownikowi (programiście) udostępnia się tylko te właściwości obiektu, które są istotne z punktu widzenia jego potrzeb na tym poziomie szczegółowości.
- Pozostałe właściwości, choć istnieją i są konieczne dla poprawnego funkcjonowania obiektu, mogą pozostać dla użytkownika całkowicie nieznane.

Przykład z życia: znajomość budowy silnika samochodowego i swobodny dostęp do jego elementów nie są konieczne do poprawnego korzystania z samochodu

C++ - klasy

Różnice miedzy `struct` i `class`

- Domyślnie wszystkie składowe obiektu typu `struct` są publicznie dostępne – można ich używać (odczyt/zapis, wywołanie) wszędzie tam, gdzie widoczna jest definicja struktury tego typu
- Domyślnie wszystkie składowe obiektu typu `class` są publicznie niedostępne. Aby stały się dostępne, musimy je zadeklarować jako *dostępne*. Kod deklaracji klasy jest podzielony na trzy rodzaje sekcji. Przynależność składowych do sekcji określa prawa dostępu.
- Prawa dostępu do składowych klasy dzielą się na:
  - typu **public**
  - typu **private**
  - typu **protected**

79

C++ - klasy

```
class JakasKlasa {
  public:
      int s1;
      int s2;
  private:
      int s3;
      int s4;
      int s5;
  protected:
      int s6;
};
```

```
class JakasInnaKlasa {
      int d1;// domyślnie: private
   public:
      int d2;
  protected:
      int d3;
  private:
      int d4;
   public:
      int d5;
};
```

(*Tak też można, ale większy bałagan w kodzie..*)

80

C++ - klasy

Metody

Skoro składowe klasy mogą być niedostępne z zewnątrz (**`private`**), to po co je w ogóle deklarować – i tak nikt z nich nie skorzysta (nie będzie do nich zapisywał, ani też je odczytywał)?

- Składowymi klasy mogą być nie tylko zmienne (pola), ale również funkcje. Takie funkcje nazywane są ***metodami klasy***.
- Metody podlegają takim samych ograniczeniom praw dostępu co pola.
- Wszystkie metody będące składowymi danej klasy mają prawa dostępu do pól zadeklarowanych jako *private* i *protected* w tej klasie.

81

C++ - klasy

Przykład:

```
class water_temp {
  double t;                 // składowa private
public:
  double get_temp() {       // metoda - składowa klasy (inline)
    return t;
  }
  double set_temp(double nt);     // metoda - składowa klasy
};

double water_temp::set_temp(double nt) {
  if (nt<100 && nt>0)
    return (t = nt);
  else
    return t;
}
```


82

C++ - klasy

Przykład:

Mam pustą szklankę oraz gąbkę.

Nasączam gąbkę 100 ml wody, po czym trzy razy wyciskam wodę z gąbki do szklanki. Jedno wyciśniecie powoduje, że z gąbki ubywa połowa zawartości wody.

Ile wody jest teraz w szklance?

Visual Studio
Program #5

83

C++ - klasy

Związki z podręcznikami o modelowaniu obiektowym

- Struktury i klasy zawierające metody nazywane są *abstrakcyjnymi typami danych*
- Obiekty to *zmienne utworzone za pomocą abstrakcyjnych typów danych*
- wywoływanie metod składowych obiektów określa się mianem *wysyłania do nich komunikatów*
- rezultatem *wysyłania komunikatów* jest uruchamianie metod działających na atrybutach obiektów
- wysyłanie komunikatów do obiektów to podstawowe działanie związane z programowaniem obiektowym

84

C++ - klasy

Zmienne:

Typy zmiennych są określone w specyfikacji języka C i C++. Są to np.: `int`, `double`, `char`.

Zdefiniować można sobie typy reprezentujące struktury oraz stałe wyliczeniowe.

Zmienne to instancje.

W kodzie programu możemy do zmiennej zapisać wartość lub odczytać ze zmiennej wartość. Wobec zmiennej strukturalnej możemy to zrobić dla każdej z jej składowych.

Obiekty:

Typy obiektów są definiowane wyłącznie przez użytkownika. Każda klasa napisana przez programistę jest nowym typem obiektowym.

Obiekty to instancje.

W kodzie programu możemy do pól składowych obiektu zapisać wartość lub odczytać z nich wartość. Możemy też wywołać metodę należącą do obiektu.

C++ - klasy

- Definiując klasę (typ obiektowy) możemy w niej zadeklarować jedną lub więcej metod.
- Deklarując obiekt, możemy te metody wywołać na rzecz tego obiektu.
- Mając kilka obiektów tego samego typu, na rzecz każdego z nich możemy wywołać tę samą metodę. Ale..
  jej działanie może być różne dla każdego z obiektów, ponieważ pola składowe tych obiektów nie muszą być zapisane tymi samymi wartościami.

C++ - klasy

Słownik programisty:

1. klasa
2. obiekt (instancja)
3. składowe obiektu: pola i metody
4. definicja vs. deklaracja
5. wywołanie metody na rzecz obiektu
6. kapsułkowanie (hermetyzacja)

C++ - klasy

Jeszcze kilka słów na temat struktur: typy strukturalne jako typy abstrakcyjne

- W C++ typy strukturalne też mogą mieć składowe metody.
- Te metody są domyślnie deklarowane jako **`public`**.
  Przykład:

```
struct Stru {
        int a;
        void prin(int b) {
             printf("%i",c);
             };
};
```


C++ - klasy

Co jeszcze mogą mieć struktury w C++?

Mogą mieć definiowane prawa dostępu do swoich składowych, np.:

```
struct X {
  private:
    int x;
  public:
    void init();
    int fun(int y);
};
```

Struktury w C++ różnią się od klas tym, że ich domyślne składowe są publiczne, natomiast w klasach – prywatne.

Więcej różnic nie ma..

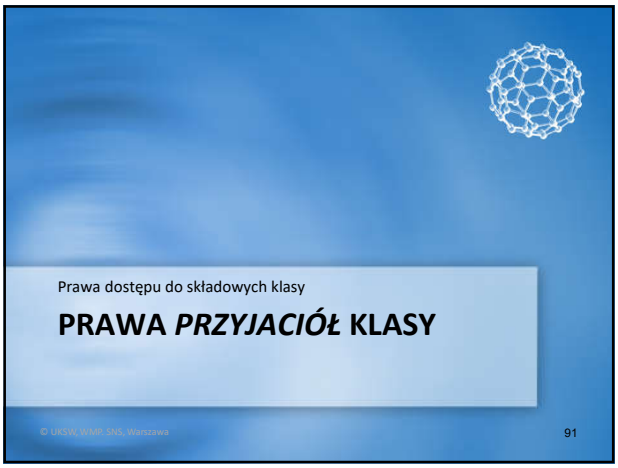
C++ - klasy

Zagnieżdżanie struktur

```
struct Bloczek {
        struct Scianka {
                int a;
        };
        int a;
        int b;
};
```

`Scianka` należy do przestrzeni nazw typu strukturalnego `Bloczek`. Obiekt typu `Bloczek` zawiera pola `a` i `b`. Dostęp do typu strukturalnego `Scianka` jest tylko za pośrednictwem `Bloczek`:

```
Bloczek::Scianka s;
```



Prawa dostępu do składowych klasy

PRAWA *PRZYJACIÓŁ* KLASY

91

C++ - klasy

Dostęp z zewnątrz:

```
class water_temp {
   double t;
public:
   double limit;
   double get_temp() {
        return t;
   }
   …
};

int main() {
   water_temp T;
   T.limit = 100;
```

Dostęp z wewnątrz:

```
class water_temp {
   double t;
public:
   …
   double set_temp(double nt) {
        if (nt<limit && nt>0)
              return (t = nt);
        else
              return t;
   }
};
int main() {
   water_temp T;
   T.set_temp(36.6);
```


92

C++ - klasy

Przykład:

```
class water_temp {
    double t;
public:
    double get_temp() {
      return t;
    }
    double set_temp(double nt);
};

double water_temp::set_temp(double nt) {

    if (nt<100 && nt>0)
      return (t = nt);
    else
      return t;
}
```

Składowe prywatne obiektu są dostępne tylko dla metod składowych tego obiektu (tj. metod: get_temp i set_temp).

Składowe publiczne są dostępne wszystkim, tj. zarówno w kodzie metod składowych jak i funkcji zewnętrznych oraz metod innych klas.

```
int main() {
  water_temp T1,T2;
  double x = T1.t;    // nie wolno!
  T1.set_temp(36.6);
  double y = T1.get_temp();
  T2.set_temp(T1.get_temp());
…
```


93

C++ - klasy

Kim są przyjaciele i co im wolno?

Przyjaciele, to ci, z którymi jesteśmy gotowi się podzielić tym, co nie jest publicznie dostępne dla wszystkich (tj. np. tym, co jest private)

Gdy chcemy udzielić pozwolenia na dostęp funkcji, niebędącej składową klasy **A**, wskazujemy tę funkcję za pomocą słowa kluczowego **`friend`** wewnątrz deklaracji klasy **A** (*to nie jest deklaracja funkcji!*)

Nie ma żadnego innego sposobu włamania się z zewnątrz, tj. przyznania sobie prawa bycia przyjacielem klasy **A** bez zmiany jej kodu.

Przyjaźń nie jest wzajemna – jeżeli klasa **A** twierdzi, że klasa **B** jest jej przyjacielem, to nie daje to klasie **A** prawa do wzajemności, tj. to nie oznacza, że **B** też musi twierdzić, że **A** jest jej przyjacielem.

94

C++ - klasy

Jako przyjaciela można wskazać funkcje globalną, składową innej klasy albo nawet całą klasę:

```
class X {
  private:
   int x;
  public:
   void init();
   int fun(int y);
   friend void global(X*, int);    // przyjaciel funkcja globalna
   friend void Y::fun(X*); // przyjaciel - składowa innej klasy
   friend class Zeta;              // cała klasa jako przyjaciel
};
```


95

C++ - klasy

- Umieszczenie wewnątrz klasy nagłówka funkcji/metody przyjaznej nie oznacza jej deklaracji. Właściwa deklaracja musi być wykonana niezależnie od tego, w innym miejscu kodu.
- Funkcja zaprzyjaźniona z klasą nie jest metodą tej klasy.
- Deklarację przyjaźni można umieścić w dowolnej sekcji definicji klasy (**`public`**, **`private`**..).
- Przyjaźń nie jest przechodnia – jeżeli **A** jest zaprzyjaźniona z **B**, a **B** jest zaprzyjaźniona z **C**, to nie znaczy, że **A** jest zaprzyjaźniona z **C**.

96

C++ - klasy

```
class X {
  private:
   int i;   // składowa prywatna
  public:
   void init();
   int fun(int y);
   friend void global(X*,int);
   friend void Y::fun(X*);
   friend class Zeta;
};

void global(X *x, int i) {
   x->i = i;
}
```

```
class Y {
   void fun(X*);
};
void Y::fun(X *x) {
   x->i = 0;
}

class Zeta {
   int h(X*);
};
int Zeta::h(X *x) {
   return x->i > 0 ? x->i : 0;
}
```

Ale ten kod się nie skompiluje..

97

C++ - klasy

Podsumowując dotychczasowe informacje:

- Nazewnictwo:
  - składowe klasy – odpowiedniki „zmiennych" i „funkcji".
  - „zmienne" zadeklarowane w klasie nazywane są **polami** klasy a „funkcje" – **metodami**
  - „zmienne" zadeklarowane na podstawie typu klasy to **obiekty**

98

C++ - klasy

Podsumowując dotychczasowe informacje:

- kapsułkowanie – ograniczanie dostępu klientowi-programiście do składowych klasy. Utworzony obiekt będzie klientowi-programiście udostępniał tylko te swoje pola i metody, które zostały zadeklarowane w klasie jako **`public`**.
- prawa dostępu do składowych chronionych mają tylko inne składowe tej samej klasy oraz przyjaciele klasy.
- struktury mają podobne możliwości co klasy, jednak w dalszych rozważaniach do definiowania typów abstrakcyjnych będą wykorzystywane głównie klasy.

99

C++ - klasy

Dobra praktyka – cz.1:

jeżeli istnieje potrzeba udostępnienia klientowi-programiście chronionych pól klasy tylko do odczytu, dla każdego z tych pól pisze się metodę (nazywaną potocznie „**geterem**" od ang. słowa 'get'), której zadaniem jest wyłącznie zwrócić wartość chronionego pola, np.:

```
...
private:
  double objetosc;
public:
  double get_objetosc() { return objetosc; }
...
```


100

C++ - klasy

Dobra praktyka – cz. 2:

jeżeli istnieje potrzeba udostępnienia klientowi-programiście chronionego pola klasy do zapisu, dla tego pola pisze się metodę (nazywaną potocznie „**seterem**" od ang. słowa 'set'), której zadaniem jest zmienić wartość chronionego pola, ale tylko pod warunkiem, że proponowana nowa wartość spełnia ograniczenia, np.:

```
private:
  double objetosc;
public:
  double set_objetosc(double obj) {
   if (obj > 0) return objetosc = obj;
   else return objetosc; }
...
```


101

Inicjalizacja obiektu

KONSTRUKTORY

102

C++ - klasy

Inicjalizacja

- Przyczyną wielu błędów w programach jest nieprawidłowe zainicjalizowanie zmiennych na początku działania programu.
- Obiekt zawiera z reguły szereg pól – ich wartości *powinny* zostać określone przed rozpoczęciem używania tego obiektu.
- Umieszczanie instrukcji incjalizujących pola obiektu przed każdą instrukcją tworzącą nowy obiekt mogłoby gmatwać kod programu i zwiększać znacznie jego rozmiar.
- Aby ułatwić inicjalizację programista dostał możliwość przypisania dowolnego zbioru instrukcji do czynności tworzenia nowego obiektu. Te instrukcje wykonają się za każdym razem kiedy tworzony jest kolejny obiekt. Mogą one inicjalizować pola obiektu.

C++ - klasy

Inicjalizacja

- zbiór instrukcji, który ma być wykonany przy każdym utworzeniu nowego obiektu, jest umieszczany w specjalnej „metodzie" nazywanej *konstruktorem.*
- konstruktor może mieć argumenty wywołania (ale nie musi).
- konstruktor nie ma nazwy.
- dla jednej klasy można zadeklarować kilka różnych konstruktorów.
- Konstruktory jednej klasy muszą się różnić sygnaturą, która w przypadku konstruktora oznacza *wyłącznie* listę typów argumentów wywołania.

C++ - klasy

Przykład deklaracji:

```
class X {
  int i;
  public:
  X();          // konstruktor
}
```

Po czym rozpoznajemy konstruktor?

1. **nie ma nazwy** – zamiast nazwy jest powtórzona nazwa klasy
2. **nie może zwracać żadnych wartości** – dlatego nie deklarujemy żadnego typu przed nazwą klasy

Definicja konstruktora:

```
X::X() { i = 0; }
```


C++ - klasy

- Konstruktor może mieć dowolną liczbę argumentów.
- Konstruktor bezargumentowy jest specjalnym typem konstruktora. Jest to tzw. *konstruktor domyślny.*
- Kiedy deklarujemy klasę i nie deklarujemy w niej żadnego konstruktora, konstruktor domyślny tworzy się automatycznie.
- Kiedy konstruktor domyślny utworzony jest automatycznie, to nie zawiera żadnego kodu. Mimo to JEST.
- Dlaczego konstruktor domyślny jest konieczny?
  - ponieważ w każdym miejscu kodu, gdzie obiekt tworzony jest bez jawnego wskazywania konstruktora, kompilator „po cichu" zawsze dodaje wywołanie konstruktora domyślnego.

A jak napisać kod, żeby przy deklaracji obiektu został wywołany inny konstruktor?

C++ - klasy

Przykład deklaracji:

```
class X {
  int i;
  public:
  X();          // konstruktor domyślny
  X(int a);     // inny konstruktor
};
…
X a1;     // tu zostanie wywołany konstruktor domyślny
X a2(1);  // a tu zostanie wywołany inny konstruktor
```

O tym, który konstruktor ma być wywołany, decyduje liczba argumentów przy nazwie reprezentującej tworzony obiekt.

C++ - klasy

- Konstruktor domyślny jest tworzony zawsze, jeżeli programista nie utworzył żadnego konstruktora.
- Jeżeli programista utworzył choć jeden konstruktor w klasie, to nawet jeżeli nie jest to konstruktor domyślny, tj. ma jeden lub więcej argumentów, konstruktor bezargumentowy nie będzie już automatycznie tworzony.
- Dlatego programista, decydując się na tworzenie konstruktorów w klasie, bierze na siebie obowiązek utworzenia również konstruktora domyślnego.

C++ - klasy

Konstruktor kopiujący

Rozważmy fragment kodu:

```
int fun(int x, int y);
…
int g = fun(a,b);
```

Kompilator przy wywołaniu tworzy kopie zmiennych. Na koniec kopiuje wartość zwracaną przez funkcję do zmiennej po lewej stronie równania. Skąd może wiedzieć w jaki sposób zrobić te kopie – tj. przekazać i zwrócić wartości zmiennych?

Po prostu wie. Bo mu jego autorzy wpisali to kopiowanie na sztywno dla wszystkich typów wbudowanych.

A co ma zrobić kompilator, kiedy taka sama sytuacja dotyczy typów utworzonych przez programistę?

C++ - klasy

Konstruktor kopiujący

Kiedy potrzeba przekazać argument przez wartość, kompilator dokonuje bezpośredniego przekopiowania bajtów ze zmiennej podanej w wywołaniu do nowo utworzonej zmiennej lokalnej wykorzystywanej wewnątrz funkcji. W przypadku struktur i klas o bardziej złożonym charakterze, takie przekopiowanie może dać fałszywy rezultat.

Kompilator nie musi jednak zawsze kopiować bajtów. Zanim to zrobi, najpierw sprawdza, czy istnieje konstruktor, którego argumentem wywołania jest referencja do obiektu tego samego typu, np.:

```
class MojaKlasa {
  public:
  MojaKlasa();                  // konstruktor domyślny
  MojaKlasa(MojaKlasa& mk);     // konstruktor kopiujący
  …
}
```