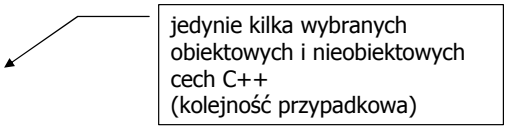
Język C++

część 3
klasy

Jarosław Gramacki
Instytut Informatyki i Elektroniki

- C vs. C++ - słowo komentarza
  - C jest stosunkowo prostym językiem programowania !
  - w codziennej praktyce do rozwiązania w zasadzie każdego problemu programistycznego wystarczy operować jedynie makrami, wskaźnikami, strukturami, tablicami, funkcjami (+ ew. parę innych rzeczy tu nie wymienionych)
  - C++ jest dużo trudniejszy, choć i tu dalej można operować jedynie makrami, wskaźnikami, strukturami, tablicami, funkcjami

- C vs. C++ - słowo komentarza
  - C++ obsługuje jednak także:
    - klasy + prywatne i chronione składowe obiektów
    - konstruktory, destruktory
    - operatory zdefiniowane przez programistę
    - dziedziczenie i polimorfizm
    - funkcje wirtualne
    - funkcje zaprzyjaźnione
    - funkcje przeciążone
    - parametry domyślne
    - funkcje z atrybutem `inline`
    - referencje
    - szablony
    - wyjątki
    - itd.



  - Mamy więc znacznie więcej opcji do wyboru i **nauczenia się**

- klasy - wprowadzenie i podstawy
  - pojęcie **klasy** stoi u podstaw programowania obiektowego w C++
  - klasy są najważniejszym narzędziem definiowania nowych typów danych
  - dzięki istnieniu dwóch sekcji: `private` i `public` (tzw. kwalifikatory dostępu), następuje rozdzielenie interfejsu dostępu do danych od implementacji tegoż dostępu
  - dzięki klasom możemy więc "panować" nad dostępem do danych (zabezpieczenie przed przypadkowym / niezamierzonym / niepowołanym / nie... itd. dostępem do danych)
  - programując w C++ korzysta się zazwyczaj z:
    1. klas i funkcji z biblioteki standardowej C++ (np. iostream)
    2. własnych klas i funkcji
    3. klas i funkcji z popularnych bibliotek niestandardowych (np. STL)
  - przykład:
    zaimplementować stos liczb całkowitych (struktura danych stosu, operacje na stosie)

- klasy, implementacja stosu - przykład nieobiektowy

ANSI C

```
#define MAX_SIZE 5

typedef enum {OK, FULL, EMPTY}
stack_state;

typedef struct
         {
          int table[ MAX_SIZE ];
          int top;
          stack_state state;
         } Stack;

// ------------------------------------
void push( Stack *s, int element )
{
    if( s->state != FULL )
      s->table[ (s->top)++ ] = element;
    else
     printf( "Pelny stos !\n" );

    if( s->top >= MAX_SIZE )
      s->state = FULL;
    else
      s->state = OK;
}
```


- klasy, implementacja stosu - przykład nieobiektowy

ANSI C

```
// ---------------------------------
int pop( Stack *s )
{
    int element = 0;

    if( s->state != EMPTY )
      element = s->table[ --(s->top) ];
    else
      printf( "Pusty stos !\n" );

    if( s->top <= 0 )
      s->state = EMPTY;
    else
      s->state = OK;

    return element;
}
// ---------------------------------
void print_stack( Stack *s )
{
    for( int i = s->top-1; i >= 0; i-- )
      printf( "[ %d ]\n", s->table[ i ] );
}
```

- klasy, implementacja stosu - przykład nieobiektowy

ANSI C

```
void main( void )  {
    Stack s1;
    int i;

    s1.top = 0;
    s1.state = EMPTY;

    push( &s1, 5 );
    push( &s1, 10 );
    push( &s1, 15 );

    printf( "Liczba elementow = %d\n", s1.top );
    print_stack( &s1 );

    printf( "Pop 1 = %d\n", pop( &s1 ));
    printf( "Pop 2 = %d\n", pop( &s1 ));
    printf( "Pop 3 = %d\n", pop( &s1 ));
    printf( "Liczba elementow = %d\n", s1.top );
    printf( "Pop 4 = %d\n", pop( &s1 ));

    for( i = 0; i <= MAX_SIZE+1; i++ )
    {
      printf( "Push : %d\n", i );
      push( &s1, i );
    }
    print_stack( &s1 );
    printf( "Liczba elementow = %d\n", s1.top );  }
```


- klasy, implementacja stosu - przykład nieobiektowy - wady !
  - definicja typu `Stack` (stosu) nie jest w żaden sposób połączona z funkcjami, które operują na tym stosie. Brak więc związku dane <-> operacje na tych danych
  - składniki stosu (elementy struktury) można bez problemu (np. w sposób niezamierzony) zmodyfikować bez użycia funkcji `push()`, `pop()` i wydrukować inaczej niż poprzez funkcje `print_stack()`. Takie beztroskie operacje na stosie jak poniżej, mogą w rozbudowanym programie bardzo "uprzykrzyć" życie
  - istota problemu polega na tym, że kompilator w żaden sposób nie kontroluje sposobu dostępu do zmiennej **s1**. Nikt i nic nie zmusi programisty do używania np. funkcji **`init()`** !!!

```
void main( void ) {
    Stack s1;              // nasz stos

    s1.top = 0;  s1.state = EMPTY;
    ...
    printf( "Liczba elementow = %d\n", s1.top );

    // lub
    s1.table[0] = s1.table[1] = 7;
    ...

void init (Stack *s) {
  s->top = 0;
  s->state = EMPTY   }
```

- klasy, implementacja stosu - przykład obiektowy

C++

```
const int MAX_SIZE = 5;
enum stack_state {OK, FULL, EMPTY};

class Stack { // deklaracja klasy
public:       // kwalifikator dostępu
    void push( int );     // metody
    int pop();            // (ich
    void print_stack();   // prototypy)
    void init();
    int ile_elementow();
private:
    int table[ MAX_SIZE ]; // dane
    int top;               // (pola)
    stack_state state; };

void Stack::push( int element ) {
    if( state != FULL )
      table[ top++ ] = element;
    else
      cout << "Pelny stos !" << endl;

    if( top >= MAX_SIZE )
      state = FULL;
    else
      state = OK;
}
```

Deklaracje klas z reguły znajduje się w pliku nagłówkowym.

Próba inicjacji danych składowych klasy bezpośrednio w jej deklaracji jest błędem składniowym. Do tego z reguły (ale nie koniecznie) służą konstruktory (patrz następny wykład)

- klasy, implementacja stosu - przykład obiektowy

C++

```
int Stack::pop() {
    int element = 0;

    if( state != EMPTY )
      element = table[ --top ];
    else
      cout << "Pusty stos !" << endl;

    if( top <= 0 )
      //nie ma potrzeby pisania Stack::top
      state = EMPTY;
    else
      state = OK;

    return element; }

void Stack::print_stack() {
    for( int i = top-1; i >= 0; i-- )
       cout << "[ " << table[ i ] << " ]" << endl;  }

inline void Stack::init() {
    top = 0;
    state = EMPTY; }

inline int Stack::ile_elementow() {
    return top;  }
```

tzw. "**get**er function" Lepsza nazwa: gettop() lub **getTop()** i zmiana nazwy pola z top na **Top**

- klasy, implementacja stosu - przykład obiektowy

C++

```
void main() {
    Stack s1;      // UTWORZENIE OBIEKTU

    s1.init();     // wywołanie metody Stack::init() dla obiektu s1

    s1.push( 5 ); // tylko metody mają dostęp do danych
    s1.push( 10 );
    s1.push( 15 );

    cout << "Liczba elementow = " << s1.ile_elementow() << endl;
    s1.print_stack();

    cout << "Pop 1 = " << s1.pop() << endl;
    cout << "Pop 2 = " << s1.pop() << endl;
    cout << "Pop 3 = " << s1.pop() << endl;
    cout << "Liczba elementow = " << s1.ile_elementow() << endl;
    cout << "Pop 4 = " << s1.pop() << endl;

    for(int i = 0; i < MAX_SIZE+1; i++ )
    {
      cout << "Push : " << i << endl;
      s1.push( i );
    }

    s1.print_stack();
      cout << "Liczba elementow = "<< s1.ile_elementow() << endl; }
```


- klasy, implementacja stosu - przykład obiektowy - zalety !
  - obecnie fragment jak poniżej już nie zadziała poprawnie (i o to chodzi !). Kompilator zgłosi błąd: **Stack::table** jest niedostępny
  - powodem odrzucenia jest kwalifikator `private` dla sekcji z danymi stosu. Wskazuje on, że tylko odpowiednia metoda danej klasy ma dostęp do danych
  - ochrona ta dotyczy zarówno modyfikacji jak i odczytu danych
  - Tak więc:
    + dane z sekcji `public` (u nas nie występują) oraz funkcje z tej sekcji są dostępne dla całego programu

    + dane z sekcji `private` oraz funkcje (u nas nie występują) z tej sekcji są dostępne jedynie poprzez metody rozważanej klasy
  - Z reguły (ale nie ma takiego wymogu) w sekcji `public` umieszcza się tylko metody, a w sekcji `private` tylko dane

```
void main( void ) {
    Stack s1;

    s1.top = 0;
    s1.state = EMPTY;
    ...
    printf( "Liczba elementow = %d\n", s1.top );

    // lub
    s1.table[0] = s1.table[1] = 7;
    ...
```

- klasy - wskaźniki do obiektów i tablice obiektów
  - wskaźniki do obiektów - notacja bez zmian (w stosunku do ANSI C)
  - tablice obiektów - notacja bez zmian (w stosunku do ANSI C)

```
void main() {
    Stack s1;          // utworzenie obiektu
    Stack *ps1 = &s1   // wskaźnik do obiektu
    Stack sTab[10];    // tablica 10 obiektów


    s1.init();         // wywołanie metody Stack::init() dla obiektu s1
    ps1->init();       // j.w ale poprzez wskaźnik
    (*ps1).init();     // j.w ale inny zapis

    // Przetwarzanie tablicy.
    for( int i = 0; i < 5; i++ )
    {
      sTab[ i ].init();
      sTab[ i ].push( i );
      sTab[ i ].push( 2 * i );
    }

    for( i = 0; i < 5; i++ )
    {
      cout << " Stos " << i << " :" << endl;
      sTab[ i ].print_stack();
    }
```


- klasy - obiekty tworzone dynamicznie
  - operatory `new` i `delete` pozwalają na dynamiczne tworzenie obiektów

```
void main() {
    Stack *ps1 = new Stack;      // Dynamiczne utworzenie obiektu typu Stack

    ps1->init();
    ps1->push( 5 );
    ps1->push( 10 );
    ps1->print_stack();

    delete ps1;                   // Zwolnienie pamieci.

    Stack *ps5 = new Stack[ 5 ]; // Dynamiczne utworzenie pięciu obiektów

    for( int i = 0; i < 5; i++ )      {
      ps5[ i ].init();
      ps5[ i ].push( i );
      ps5[ i ].push( 2 * i );
    }

    for( i = 0; i < 5; i++ )      {
      cout << " Stos " << i << " :" << endl;
      ps5[ i ].print_stack();
    }

    delete [] ps5;               // Zwolnienie pamieci przydzielonej tablicy.
```

- klasy - metody z kwalifikatorem inline
  - poniżej znajduje się fragment programu z przed paru slajdów wstecz
  - metody init() i ile_elementów() tak w zasadzie służą tylko do odczytu / zapisu pól klasy Stack. Wykonanie tych czynności "normalną" funkcją było by nieefektywne ("ciężarówka do przewozu jednej cegły").
  - `s1.init();` <-równoważne-> `s1.top = 0; s1.state = EMPTY`
    `n = s1.ile_elementow()` <-równoważne-> `n = s1.top;`
  - mimo wszystko zastosowanie metody zamiast bezpośredniego odwołania się do pól jest dobrym rozwiązaniem. Inaczej trzeba by przenieść pola `top` i `state` do sekcji `public`. Kwalifikator inline zwiększa jedynie efektywność użycia funkcji.
  - Uwaga! Kwalifikator `inline` jest tylko wskazówka dla kompilatora. Zbyt duże funkcje (w ocenie kompilatora !) i tak nie będą "typu" `inline`

```
.
.
inline void Stack::init() {
    top = 0;
    state = EMPTY;
}

inline int Stack::ile_elementow() {
    return top;
}
.
.
```


- klasy - metody z kwalifikatorem inline
  - w języku C++ często stosuje się wiele bardzo krótkich funkcji takich jak `Stack::init()` i `Stack::ile_elementow()`
  - wprowadzono więc ułatwienie, polegające na możliwości połączenia deklaracji funkcji z jej definicją

```
class Stack
{
public:
    void push( int );
    int pop();
    void print_stack();

    // Poniższe dwie funkcje zdefiniowane są
    // w deklaracji klasy, a wiec kompilator automatycznie
    // uzna je za funkcje inline.

    void init() { top = 0; state = EMPTY; }
    int ile_elementow() { return top; }

private:
    int table[ MAX_SIZE ];
    int top;
    stack_state state;
};
```

zwróć uwagę gdzie tutaj stoi średnik ! (prototyp) ...

... a gdzie tutaj (brak średnika)

- klasy - wskaźnik this
  - w języku C++ każda funkcja składowa klasy otrzymuje ukryty argument wskazujący na obiekt, który ją wywołuje. Argument ten to tzw. wskaźnik `this`
  - "teoretyczna" ilustracja wskaźnika może wyglądać jak poniżej

```
void Stack::push( int element ) {
    if( state != FULL )
      table[ top++ ] = element;
    else
      cout << "Pelny stos !" << endl;

    if( top >= MAX_SIZE )
      state = FULL;
    else
      state = OK; }
```

```
void Stack::push( Stack* this, int element ) // zawsze jako pierwszy argument
{
    if( this->state != FULL )
        this->table[ this->top++ ] = element;
    else
      cout << "Pelny stos !" << endl;

    if( this->top >= MAX_SIZE )
        this->state = FULL;
    else
        this->state = OK; }
```


- klasy - wskaźnik this
  - wskaźnik `this` jest przekazywany automatycznie podczas wywołania funkcji. Ilustracja wywołania funkcji w drugiej ramce poniżej jest więc czysto teoretyczna
  - wskaźnik `this` (niejawnie) używany jest do np. zwracania pola skojarzonego z obiektem - patrz trzecia ramka choć w praktyce nikt tak nie robi, zapis jest poprawny.

```
void main() {
    Stack s1, s2;

    s1.push( 5 );
    s1.push( 10 );
    s2.push( 15 );
```

```
void main() {
    Stack s1, s2;

    push(&s1, 5);
    push(&s1, 10 );
    push(&s2, 15 );
```

```
inline int Stack::ile_elementow()
{
    // return top;
    return this->top;
    return (*this).top;
}
```

- klasy - wskaźnik this

```
// przykład podstawowy #2
class Test {
public:
   Test( int = 0 );                // konstruktor domyślny
   void print() const;
private:
   int x;
};

Test::Test( int value )  { x = a }         // konstruktor, inicjalizacja x

// print x using implicit and explicit this pointers;
// parentheses around *this required
void Test::print() const   {
   // implicitly use this pointer to access member x
   cout << "        x = " << x;

   // explicitly use this pointer to access member x
   cout << "\n  this->x = " << this->x;

   // explicitly use dereferenced this pointer and
   // the dot operator to access member x
   cout << "\n(*this).x = " << ( *this ).x << endl;
}
int main() {
   Test testObject( 12 );
   testObject.print();    return 0; }
```


- klasy - wskaźnik this
  - przypadki gdy należy użyć wskaźnika `this`
    - gdy zwracamy wskaźnik do obiektu wywołującego funkcję (używamy `return this`)
    - gdy zwracamy referencję do obiektu wywołującego funkcję (używamy `return *this`)
  - ostatni przypadek umożliwia eleganckie "składanie" operacji w jedno wyrażenie (kaskadowe wywołanie funkcji)

- klasy - wskaźnik this

```
Stack& Stack::push( int element )
//w oryginalnej wersji bylo void Stack::push( int element )

{
    if( state != FULL )
      table[ top++ ] = element;
    else
      cout << "Pelny stos !" << endl;

    if( top >= MAX_SIZE )
      state = FULL;
    else
      state = OK;
    return *this;  // zwraca referencje do bieżącego obiektu
}

// i teraz możliwe jest kaskadowe wywołanie funkcji
Stack s1;
s1.push(5).push(10).push(15);
```

zwraca `*this` typu `Stack&`

```
//... zamiast
s1.push(5);
s1.push(10);
s1.push(15);
```

Inny przykład: kaskadowe wywołania przeciążonych operatorów !

```
String a, b, c;
// String& String::operator=(const String&);
a = b = c = "Napis";
a.operator=(b.operator=(c.operator=("Napis")));
```


- klasy - wskaźnik this

```
// jak to działa ?

...
// i teraz możliwe jest kaskadowe wywołanie funkcji
Stack s1;
s1.push(5).push(10).push(15);
...
```

  - operator kropki (.) wiąże od strony lewej do prawej, więc wyrażenie `s1.push(5).push(10).push(15);` najpierw wywoła funkcję `push(5)`, która zwróci referencję do obiektu s1
  - dzięki temu pozostała część wyrażenia może zostać przedstawiona jako `s1.push(10).push(15);`
  - itd.

```
// tak też jest poprawnie
...
// i teraz możliwe jest kaskadowe wywołanie funkcji
Stack s1;
s1.push(5).push(10).push(15).print_stack()
...
```

- klasy - kwalifikator friend
  - motywacja pierwsza: czasami istnieje potrzeba (ze względu na szybkość wykonania) udostępnienia pola `private` pewnej klasy jako argumentu (jakiejś) funkcji zewnętrznej
  - funkcja `friend` (zaprzyjaźniona) NIE otrzymuje (w przeciwieństwie do funkcji składowych klasy) wskaźnika `this`. Aby mogła ona oddziaływać na obiekt (tu: MyClass), musi w sposób jawny otrzymać do niego odniesienie
  - miejsce pojawienia się deklaracji `friend` (sekcja `public, private, protected`) nie ma znaczenia (patrz też uwaga w ramce)

- klasy - kwalifikator friend

```
class MyClass
{
 friend void fun ( MyClass& _p, int _a );
 // funkcja zewnętrzna (na zewnątrz klasy)
 public:
   ...
 private:
   int a, b, c;
};
------------------------------
void fun (MyClass& _p, int _a)
{
  // fun() posiada dostęp do pól private klasy
  _p.a = _a;
  _p.b = _a + _p.c;
  ...
}

int main()
{
  MyClass obj;
  fun ( obj, 10 ); // a nie Myclass.fun ( 10 ); gdyż fun() NIE jest
elementem klasy
  ...
```

jawne odniesienie do obiektu

mimo, że prototyp funkcji zaprzyjaźnionej pojawia się w definicji klasy, nie jest ona jej funkcją składową

wskazanie **na konkretny obiekt klasy,** gdyż każdy obiekt posiada przecież swoje własne pola

- klasy - kwalifikator friend
  - motywacja druga: można również udostępniać funkcjom jednej klasy pola prywatne innej klasy
  - Uwaga!: to klasa wybiera sobie funkcje zaprzyjaźnione, a nie odwrotnie

```
class MyClass
{
public:
  void f();
  void g();
};

class InnaKlasa
{
   // Myclass::f() posiada pełny dostęp do pól prywatnych klasy Inna
   friend void MyClass::f();
 private:
   int a, b, c;
};
```

"zaprzyjaźniam" się z funkcją f() klasy MyClass. MyClass::f() posiada pełny dostęp do InnaKlasa

- klasy - kwalifikator friend
  - gdy wszystkie funkcje powinny uzyskać dostęp do pól prywatnych innej klasy, stosuje się inny zapis. Mówimy wtedy o zaprzyjaźnionej klasie
  - A --> B --> C
    Gdy klasa A jest zaprzyjaźniona z klasą B, a klasa B jest zaprzyjaźniona z klasą C, nie można zakładać, że B jest zaprzyjaźniona z A (nie ma symetrii), ani C z A (nie ma przechodniości)

```
class One
{
   friend class Two;
   ...
}
```

Funkcje zaprzyjaźnione będą jeszcze często używane przy przeciążaniu operatorów. Będziemy o tym mówić !!!

Funkcje zaprzyjaźnione będą jeszcze często używane do tworzenia klas iteratorów. Będziemy o tym mówić !!!

- klasy - składowe statyczne klasy, pola statyczne
  - w ANSI C gdy zmienna miała być udostępniana kilku funkcjom, deklarowana była jako zmienna globalna. Wady takiego rozwiązania są znane.
  - w C++ można stosować pola statyczne.
  - pozwala to dzielić informacje w obrębie różnych obiektów danej klasy
  - uzyskujemy więc rodzaj równoważnika zmiennej globalnej, ale tylko w stosunku do obiektów jednej klasy
  - przykład: globalny licznik obiektów klasy X utworzonych przez program

```
class X
{
public:
  ...
  static int count;
};
```

pole static można zainicjować podczas deklarowania w ciele klasy ?

  - bez `static` zmienna `count` jest duplikowana w każdym obiekcie X
  - bez `static` każdy obiekt ma dostęp tylko do własnego licznika (a miał być globalny)
  - ze `static` dla wszystkich obiektów klasy X istnieje tylko jedna zmienna `count`

```
X::X()
{
  Count++;
};
```

```
X::~X()
{
  Count--;
};
```

dla sprawdzenia działania, zwiększamy licznik w konstruktorze klasy X a zmniejszamy w destruktorze

- klasy - składowe statyczne klasy, pola statyczne
  - jak zainicjalizować zmienną statyczną?
  - w konstruktorze? NIE, gdyż zmienna byłaby inicjalizowana wraz z każdym kolejno tworzonym obiektem
  - C++ umożliwia inicjalizowanie takiej zmiennej "globalnie" (tak jak w ANSI C dla zwykłej zmiennej globalnej)
  - to, że jest to zmienna `private`, nie przeszkadza w tym

```
int X::Count = 0; // gdzieś poza klasą
```

  - statyczne dane składowe klasy (ale i statyczne funkcje składowe klasy - patrz kolejne slajdy) istnieją i mogą być użyte nawet wtedy, gdy NIE został utworzony żaden obiekt tej klasy !

- klasy - składowe statyczne klasy, funkcje statyczne
  - jak odczytać wartość licznika `count` ? można to zrobić "klasycznie" (`getCount()` w sekcji `public`)
  - taką funkcje można jednak wywołać tylko za pośrednictwem istniejącego obiektu klasy X. A przecież count to globalny licznik obiektów. Gdy jest ich kilka, to którego z nich użyć ? A gdy w danej chwili nie ma żadnego obiektu ?

```
class X
{
public:
  ...
  int getCount() const { return Count }
private:
  static int Count;
};

X obj1;
X obj2;
X obj3;

int licznik = obj1.getCount(); // a dlaczego nie obj3.getCount();
```

o stałych (`const`) funkcjach składowych będzie też mowa na następnym wykładzie (dotyczącym konstruktorów i destruktorów)

- klasy - składowe statyczne klasy, funkcje statyczne

```
class X
{
public:
  ...
  static int getCount() const { eturn Count }
private:
  static int Count;
};
int licznik = X::getCount();
```

funkcja statyczna

przyjęło się, że statyczne funkcje składowe są wywoływane z wykorzystaniem nazwy klasy, a NIE nazwy obiektu (bo tak też można)

  - funkcja statyczne NIE pobiera wskaźnika this. Gdy użyjemy go gdzieś w funkcji, pojawi się błąd !
  - dostępne są jej tylko pola statyczne i funkcje statyczne klasy
  - wywołanie funkcji statycznej NIE wymagane pośrednictwo obiektu danej klasy

- klasy - składowe statyczne klasy, funkcje statyczne
  - metody statyczne mają bezpośredni dostęp jedynie do pól statycznych (gdyż nie pobierają jak wiemy wskaźnika this). NIE maja więc (bezpośredniego) dostępu do innych elementów klasy. Można jednak to zmienić przekazując na przykład "ręcznie" stosowny wskaźnik do obiektu (patrz przykład niżej)
  - metody NIE-statyczne mają dostęp zarówno do statycznych jak i nie-statycznych elementów klasy

```
#include <iostream>

using namespace std;

class st
{
    private:
        int a;
    public:
        static int s_test ( st* obiekt ) { obiekt->a = 5; return (obiekt->a);
};

int main() {
    st ob1;
    cout << ob1.s_test( &ob1 );
    return 0;
}
```


- klasy - składowe statyczne klasy, przykład
  - kompletny przykład użycia pól i metod statycznych

```
// emp.h

class Employee {

public:
   Employee( const char *, const char * );          // constructor
   ~Employee();                                     // destructor
   const char *getFirstName() const;                // return first name
   const char *getLastName() const;                 // return last name

   // static member function
   static int getCount();                   // return # objects instantiated

private:
   char *firstName;
   char *lastName;

   // static memeber data
   static int Count;                        // number of objects instantiated

};
```

- klasy - składowe statyczne klasy, przykład

```
// emp.cpp
int Employee::Count = 0;  // define and initialize static data member
int Employee::getCount() { return Count; }

Employee::Employee( const char *first, const char *last ) { // konstruktor
   firstName = new char[ strlen( first ) + 1 ];
   strcpy( firstName, first );
   lastName = new char[ strlen( last ) + 1 ];
   strcpy( lastName, last );
   ++count;  // increment static count of employees
 }
Employee::~Employee() {                                    // destruktor
   delete [] firstName;
   delete [] lastName;
   --count;  // decrement static count of employees
 }
const char *Employee::getFirstName() const {
   // const before return type prevents client from modifying
   // private data; client should copy returned string before
   // destructor deletes storage to prevent undefined pointer
   return firstName; }

const char *Employee::getLastName() const  {
   // const before return type prevents client from modifying
   // private data; client should copy returned string before
   // destructor deletes storage to prevent undefined pointer
   return lastName; }
```

o skomentowanych tu powodach użycia `const` będziemy jeszcze mówić

- klasy - składowe statyczne klasy, przykład

```
// main.cpp
int main() {
   cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;   // use class name Employee

   Employee *e1Ptr = new Employee( "Susan", "Baker" );
   Employee *e2Ptr = new Employee( "Robert", "Jones" );

   cout << "Number of employees after instantiation is "
        << e1Ptr->getCount();         // this time use object's pointer


   cout << "\n\nEmployee 1: "         // employees and their data
        << e1Ptr->getFirstName()
        << " " << e1Ptr->getLastName()
        << "\nEmployee 2: "
        << e2Ptr->getFirstName()
        << " " << e2Ptr->getLastName() << "\n\n";

   delete e1Ptr;
   e1Ptr = 0;     // disconnect pointer from free-store space
   delete e2Ptr;
   e2Ptr = 0;     // as above

   cout << "Number of employees after deletion is "
        << Employee::getCount() << endl;   // use class name Employee
   return 0;  }
```