Język C++

część 4
konstruktory i destruktory

Jarosław Gramacki
Instytut Informatyki i Elektroniki

- konstruktory i destruktory - wprowadzenie i podstawy
  - konstruktory i destruktory są to specjalne kategorie metod danej klasy
  - czym różnią się dwa poniższe fragmenty ?
    Jaki będzie wynik działania drugiego fragmentu ?

```
void main()
{
    Stack s1;
    s1.init();
    ...
    s1.push( 5 );
    s1.print_stack();
    ...
```

```
void main()
{
    Stack s1;
    ???
    ...
    s1.push( 5 );
    s1.print_stack();
    ...
```

  - wyniku nie da się przewidzieć, gdyż obiekt `s1` nie został prawidłowo zainicjalizowany
  - bez dodatkowych mechanizmów, nikt i nic nie zmusi programisty aby wywołał stosowną metodę (tu: `init()`)
  - rozwiązaniem problemu są konstruktory, które gwarantują samoczynną inicjalizację zmiennych (obiektów)
  - konstruktor klasy wywoła się **samoczynnie** w chwili utworzenia obiektu tej klasy
  - destruktor klasy wywoła się **samoczynnie** w chwili gdy obiekt przestaje być dostępny
  - Uwaga! Nie ma obowiązku definiowania konstruktora i destruktora klasy.
    Jest to jednak w większości przypadków zalecane (wygodne) !!!

- konstruktory
  - konstruktor posiada zawsze taką samą nazwę co klasa, której dotyczy
  - nie zwraca żadnej wartości (nawet `void`), często jest `inline`
  - problematyczny fragment programu z poprzedniej strony teraz jest już OK

```
#include <iostream.h>
const int MAX_SIZE = 5;
enum stack_state {OK, FULL, EMPTY};

class Stack {
public:
    Stack();      // Konstruktor klasy Stack.
    void push( int );
    int pop();
    void print_stack();
    int ile_elementow();
private:
    int table[ MAX_SIZE ];
    int top;
    stack_state state;
};

// Konstruktor: inicjuje strukture Stack.
inline Stack::Stack() {
    top = 0;
    state = EMPTY;
 }
```

```
void main()
{
    Stack s1;
    ???
    ...
    s1.push( 5 );
    s1.print_stack();
    ...
}
```

wersja poprzednia

- konstruktory
  - konstruktor nie zwraca żadnej wartości, ale ...
  - konstruktor (jak każda funkcja) może pobierać argument(y)

```
class Stack
{
public:
    Stack();
    Stack( int );
    Stack( int, int );
    void push( int );
    int pop();
    void print_stack();
    int ile_elementow();
private:
    int table[ MAX_SIZE ];
    int top;
    stack_state state;
};
```

```
// Konstruktor bez argumentu.
inline Stack::Stack()
{
    top = 0;
    state = EMPTY;
}

// Konstruktor z jednym argumentem.
inline Stack::Stack( int _element )
{
    table[ 0 ] = _element;
    top = 1;
    state = OK;
}

// Konstruktor z dwoma argumentami.
Stack::Stack( int _elem1, int _elem2 )
{
    top = 2;
    state = OK;
    push( _elem1 );
    push( _elem2 );
}
```

- konstruktory
  - każdy z konstruktorów (działa tutaj mechanizm przeciążania funkcji) można teraz wywołać indywidualnie

```
void main() {
    Stack s1;           // Wywoluje Stack::Stack().
    cout << "s1 :" << endl;
    s1.push( 5 );
    s1.print_stack();

    Stack s2( 10 );     // Wywoluje Stack::Stack( int ).
    cout << "s2 :" << endl;
    s2.print_stack();

    Stack s3( 15, 20 ); // Wywoluje Stack::Stack( int, int ).
    cout << "s3 :" << endl;
    s3.print_stack();
    ...
```

  - inna składnia tego samego przypadku. Używana głównie przy dynamicznym tworzeniu obiektów za pomocą operatora `new`

```
void main()
{
    Stack s1 = Stack();
    Stack s2 = Stack( 10 );
    Stack s3 = Stack( 10, 20 );
    ...
```


- konstruktory
  - każdy z konstruktorów może obsługiwać również parametry domyślne

```
class Data
{
public:
    // Konstruktor trójargumentowy.
    Data( int _dzien, int _miesiac = 10, int _rok = 2000 );
private:
    int dzien, miesiac, rok;
};
```

```
void main()
{
    Data d1(26, 01, 2000);
    Data d2;              // BLAD, nie ma konstruktora bezparametrowego
    Data d3 (12)          // OK, data: 12-10-2000
    Data d4 = d1;         // OK, inicjalizacja innym obiektem
    ...
```

  - w ostatniej deklaracji użyto tzw. niejawnego konstruktora kopiującego. Jest on zawsze zdefiniowany (niejawnie) dla każdej klasy
  - konstruktor ten można modyfikować (zaawansowane zagadnienie - pomijamy je w tej chwili)
  - w praktyce własny konstruktor kopiujący będziemy pisali dla klas z pamięcią przydzielaną dynamicznie (na przykład klasy zawierające pola typu `char *str`)

- konstruktory
  - dla tablic obiektów, każdy element tablicy inicjalizowany jest konstruktorem domyślnym (konstruktor bezparametrowy lub taki, który da się wywołać bez argumentów)
  - możemy też każdy element tablicy inicjalizować indywidualnie dowolnym konstruktorem

```
class Data
{
public:
    // Konstruktor trójargumentowy.
    Data( int _dzien, int _miesiac = 10, int _rok = 2000 );

    // Konstruktor bezparametrowy
    Data();
private:
    int dzien, miesiac, rok;
};
```

```
void main()
{
    Data dTab1[3];          // wywołano konstruktor domyślny
                            // (zakładamy, że istnieje)
    Data dTab2[3] = {
                      Data(12, 08, 2003),
                      Data(15),
                      Data()
                    };
```


- konstruktory
  - obiekty tworzone dynamicznie też musza zostać poprawnie zainicjalizowane. Tu nie ma wyjątku !
  - w przypadku tablic obiektów mogą pojawić się pewne komplikacje

```
void main()
{
    Stack *ps1 = new Stack();
    Stack *ps2 = new Stack( 10 );
    Stack *ps3 = new Stack( 10, 20 );

    Data *pd1 = new Data(26);  // data: 26-10-2000

    ...
    Stack *ps5 = new Stack[5];  // pojawi się błąd, gdy NIE będzie istniał
                                //  konstruktor domyślny
```

- destruktory, główne zasady (1)

  - destruktory gwarantują wykonanie wszystkich koniecznych czynności (np. zwolnienie zaalokowanej pamięci) zanim obiekt przestanie być dostępny (i będzie już na to za późno !)
  - w praktyce będzie to miało miejsce gdy np. zmienna lokalna (obiekt) przestaje być dostępna
  - destruktory obiektów globalnych wykonują się po zakończeniu funkcji main() (czyli przed zakończeniem się programu)
  - destruktory obiektów lokalnych wykonują się pod koniec funkcji `main()` (zakładamy, że obiekty utworzono wewnątrz funkcji `main()`)

- destruktory, główne zasady (2)

  - destruktor posiada zawsze taką samą nazwę co klasa, której dotyczy poprzedzoną znakiem (**~** tylda)
  - można definiować (inaczej niż dla konstruktorów) tylko jeden destruktor dla danej klasy
  - destruktor nic nie pobiera i nic nie zwraca (zawsze).
    Nie może być więc przeciążany.
  - o ile konstruktor z reguły jest zawsze wymagany, destruktor stosuje się z reguły wtedy, gdy alokujemy (ręcznie) pamięć dla obiektu (lub jego fragmentu - np. pola typu `char*)`

- destruktory

w bibliotece standardowej C++ jest klasa string (pisana z małej litery). Tu jest nasza własna wersja obsługi napisów

```
class String
{
public:
    String( const char* _str );
    String( int _size );
    ~String();    // Destruktor klasy String.
    void set( const char* _str ); // seter
    const char* get();            // geter
    void print();
private:
    int size;
    char* str;
};

// Konstruktor obiektu w oparciu o łańcuch
String::String( const char* _str ) {
    size = strlen( _str );
    str = new char[ size + 1 ];
    strcpy( str, _str );
}

// Rezerwacja pamięci dla obiektu String
String::String( int _size ) {
    size = _size;
    str = new char[ size + 1 ];
    *str = '\0';
    str[ size ] = '\0';
}
```

```
// Destruktor obiektu String zwalnia
// pamięć przydzielona
// za pomocą operatora new należącego
// do konstruktora.

inline String::~String()
{
    delete [] str;
}


String s( "Obiekt globalny" );

void main()
{
    s.print();

    String s1( "Obiekt lokalny" );
    s1.print();
    cout << s1.get() << endl;

    String s2( 12 );
    s2.set( "Test obcinania" );
    s2.print();
}
```

obcięcie do 11+1 znaków

- konstruktory i destruktory - wybrane zagadnienia pokrewne

- inicjalizacja a przypisanie, pola const
  - język C++ odróżnia fazę inicjalizacji zmiennej od fazy przypisania tej zmiennej wartości (w języku ANSI C NIE ma takiego ścisłego rozróżnienia)

```
const int i1 = 7; // OK, inicjalizacja
i1 = 12;          // BLAD, przypisanie wartości zmiennej const int
const int i2      // BLAD, dla const konieczna inicjalizacja
```

  - błędem (kompilatora) zakończy się więc poniższy fragment:

```
class String2 {
public:
    String2( int _size );
    ~String2();
    void set( const char* _str );
    const char* get();
    void print();
private:
    const int size;              // pole size jest teraz const.
    char* str;
};

// Zarezerwowanie pamięci dla String2.
String2::String2( int _size ) {
    size = _size;                // BLAD: przypisanie obejmuje daną const.
    str = new char[ size + 1 ];
    *str = '\0';
}
```

zwróć uwagę na czytelne nazewnictwo:
_size
size

- inicjalizacja a przypisanie, pola const - lista inicjalizująca
  - tworząc obiekt, wyróżnić można dwie fazy wykonywania się konstruktora:
    inicjalizacja pól obiektu (alokacja pamięci dla pól)
    przypisanie polom wartości (instrukcje zawarte pomiędzy {} konstruktora)
  - faza inicjalizacji może zostać obsłużona poprzez listę inicjalizacyjną konstruktora
  - znajdują się tam wartości z jakimi należy zainicjalizowac pole (pola). Oddzielamy je przecinkami
  - nie ma obowiązku inicjalizować wszystkich pól, można tylko wybrane

```
class String2 {
public:
    String2( int _size );
    ~String2();
    void set( const char* _str );
    const char* get();
    void print();
private:
    const int size;
    char* str;
};

// Rezerwuje pamiec dla obiektu String2.
String2::String2( int _size ) : size( _size ) {     // Inicjalizacja pola
    //Faza przypisania.
    str = new char[ size + 1 ];
    *str = '\0';
}
```

teraz możemy zainicjować pole **const przed** wykonaniem instrukcji konstruktora

pole

wartość pola

- obiekty const
  - obiekty (tak jak i inne zmienne) mogą być deklarowane z kwalifikatorem `const`

```
const String s( "Zmienna globalna" );
const Data d1(26, 01, 2000);
```

  - jednak wtedy NIE będzie można używać żadnej metody klasy w stosunku do obiektu `const`. Pojawi się ostrzeżenie lub błąd kompilatora

```
void main()
{
 const String s1( "Staly obiekt String" );
 cout << s1.get() << endl;
}
```

  - Są jednak metody (tu: `get()`), które nie modyfikują obiektu, a jedynie manipulują nim (pobierają dane). Metod takich można używać w stosunku do obiektu `const`. Metody takie należy w deklaracji klasy oraz w ich definicji specjalnie wyróżnić

- metody const

porównaj też przykład podany na wykładzie 3, gdzie pojawiają się metody const

```
class String
{
public:
    String( const char* _str );
    String( int _size );
    ~String();
    void set( const char* _str );
    const char* get() const;  // Funkcja składowa const
    void print() const;       // Funkcja składowa const
private:
    int size;
    char* str;
};
...
//Zwraca wskaźnik do łańcucha znakowa ASCII zawartego w obiekcie
String.
inline const char* String::get() const // Nie zapomnij wstawić const.
{
    return str;  }

//Wyswietla obiekt String.
inline void String::print() const      // Nie zapomnij wstawić const.
{
    cout << str << endl;  }
void main()  {
    const String s1( "Staly obiekt String" );
    cout << s1.get() << endl;  }
```

Funkcje składowe, które NIE modyfikują obiektu, zawsze powinny być deklarowane jako `const`. Pozwoli to uniknąć wielu błędów

bez **const** jak powyżej pojawił by się tutaj błąd

- metody const
  - przypadek szczególny występuje w odniesieniu do konstruktorów i destruktorów, które z reguły modyfikują obiekty (bo taka jest ich rola)
  - dla obiektów (`const` i NIE-const), **NIE** można deklarować konstruktorów i destruktorów jako `const`
  - konstruktor musi mieć możliwość zmiany danych składowych obiektu, gdyż tylko w ten sposób może je zainicjalizować
  - destruktor musi natomiast wykonać pewne operacje końcowe zanim obiekt zostanie usunięty
  - Przykład: (następny slajd)

- metody const
  - kompletny przykład różnych wariantów użycia obiektów `const` i funkcji `const`

```
// time.h

class Time {
public:
   Time( int = 0, int = 0, int = 0 );  // default constructor

   // set functions
   void setTime( int, int, int );  // set time
   void setHour( int );            // set hour
   void setMinute( int );          // set minute
   void setSecond( int );          // set second

   // get functions (normally declared const)
   int getHour() const;            // return hour
   int getMinute() const;          // return minute
   int getSecond() const;          // return second

   // print functions (normally declared const)
   void printUniversal() const;    // print universal time
   void printStandard();           // print standard time

private:
   int hour;    // 0 - 23 (24-hour clock format)
   int minute;  // 0 - 59
   int second;  // 0 - 59    };
```

- metody const

```
// time.cpp

#include <iostream>
#include "time.h"

// set hour, minute and second values
void Time::setTime( int hour, int minute, int second )  {
   setHour( hour );
   setMinute( minute );
   setSecond( second );
}
...
// return hour value
int Time::getHour() const {
   return hour;
}

// print Time in standard format
void Time::printStandard()  {
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << setfill( '0' ) << setw( 2 ) << minute
        << ":" << setw( 2 ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```


- metody const

```
// main.cpp

// Attempting to access a const object with
// non-const member functions.

#include <iostream>  #include "time.h"

int main() {
   Time wakeUp( 6, 45, 0 );        // non-constant object

   const Time noon( 12, 0, 0 );    // constant object

                          // OBJECT      MEMBER FUNCTION
   wakeUp.setHour( 18 );  // non-const   non-const

   noon.setHour( 12 );    // const       non-const // BŁĄD, modyfikacja
                                                    // obiektu const

   wakeUp.getHour();      // non-const   const

   noon.getMinute();      // const       const
   noon.printUniversal(); // const       const

   noon.printStandard();  // const       non-const // BŁĄD
                                                    // pritnStandard NIE const

   return 0; }
```

obiekt NIE-const; zmienny czas pobudki

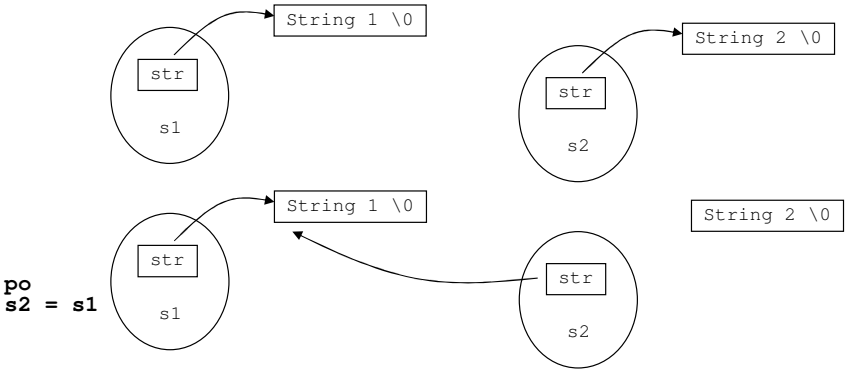
obiekt const; północ to północ

"tabelka" różnych przypadków

- klasy z polami wskaźnikowymi
  - klasa `String` posiada pole(wskaźnikowe) `char *str`
  - użycie klasy z polem wskaźnikowym (samo w sobie nie jest to oczywiście błędne) może jednak prowadzić do błędów

```
String s1( "String 1" );
String s2( "String 2" );
s2 = s1;  // ???
```

  - wewnątrz obiektu `s1` znajduje się wskaźnik do bloku pamięci zawierającego napis `"String 1"`. Istnieje także odrębny obiekt `s2` zawierający napis `"String 2"`.
  - po instrukcji `s2 = s1;` otrzymamy więc



- klasy z polami wskaźnikowymi
  - przedstawiona sytuacja wiąże się z (co najmniej) trzema problemami

    1. pamięć dla s2 została na zawsze "utracona" - nigdy już jej nie zwolnimy (tzw. "wyciek pamięci z programu")

    2. oba obiekty zawierają wskaźniki do tego samego napisu. Gdy jeden z obiektów nie będzie już potrzebny, jego destruktor usunie pamięć nadal wskazywaną przez pole drugiego obiektu !

    3. ponadto: każda modyfikacja jednego obiektu (np. poprzez `String::set()`) modyfikuje też drugi obiekt (a z reguły od razu tego błędu nie zauważymy !)

  - Rozwiązanie:

    1. domyślne zachowanie się operatora przypisania `=` jest w tym przypadku poprawne

    2. należy przedefiniować (przeciążyć) działanie tego operatora wobec klasy String

    3. cel osiągniemy deklarując funkcję `operator=()`

Szczegóły związane z przeciążaniem operatorów będą tematem oddzielnego wykładu.

- przeciążanie operatora =
  - język C++ pozwala na zmianę zachowania się predefiniowanych operatorów języka w zależności od typów argumentów
  - do klasy dodamy więc funkcję `operator=()`, której zadaniem będzie kopiowanie zawartości pamięci wskazywanej przez `str` zamiast kopiowania tylko wskaźnika

```
class String {
public:
    String( const char* _str );
    String( int _size );     ~String();
    void set( const char* _str );
    const char* get() const;
    void print() const;
    String& operator=( const String& _string ); // Operator przypisania.
private:
    int size;
    char* str;
};
String& String::operator=( const String& _string )
    if( this == &_string )                  // Sprawdza s1 = s1;
       return *this;

    delete [] str;                  // Usuwa stary łańcuch.
    size = _string.size;
    str = new char[ size + 1 ];  // Przydziela niezbędna pamięć.
    strcpy( str, _string.str );  // Kopiuje zawartość łańcucha.

    return *this; }
```

asekuracyjnie, aby obsłużyć wyrażenie takie jak np. `s1 = s1;`

porównaj uwagę na kolejnym slajdzie

- przeciążanie operatora =
  - teraz obie instrukcje są równoważne

```
s2 = s1;
s2.operator=(s1);  // zwykłe wywołanie funkcji
                   // o nazwie operator=
```

operator = jest operatorem dwu-argumentowym. U nas, w wersji przeciążonej, również. Pierwszy argument to obiekt na rzecz którego wywołujemy funkcję operator=(), drugi to argument tejże funkcji

- przeciążanie operatora =
  - łączenie w jeden łańcuch operacji kilka kolejnych instrukcji przypisania, czyli korzystanie z funkcjonalności analogicznej jak dla zwykłego operatora przypisania zastosowanego do zmiennych prostych typów

```
String s1( "String 1" );
String s2( "String 2" );
String s3( "String 3" );
s3 = s2 = s1;
```

referencje: porównaj też informacje podane na wykładzie 2

  - powyższe stwierdzenie łatwiej zrozumieć gdy użyjemy innej (ale równoważnej) formy
    (operator = jest łączny prawostronnie)

```
s3.operator=(s2.operator=(s1));
```

  - widać teraz wyraźnie, że argumentem funkcji `operator=()` jest wartość zwracana z poprzedniego wywołania funkcji `operator=()`. Oznacza to, że zwracany przez funkcję `operator=()` typ musi być akceptowany jako wejście tej samej funkcji
  - sygnatura domyślnej wersji funkcji `operator=()` wygląda następująco:

```
AClass& AClass::operator=(const AClass&);
```


- konstruktor kopiujący
  - Z podobną sytuacją spotkamy się przy wyrażeniu

```
String s2 = s1;  // używamy domyślnego
                 // konstruktora kopiującego
```

  - powyżej mamy do czynienia z inicjalizacją a NIE przypisaniem, więc używany jest NIE operator przypisania ale konstruktor kopiujący
  - wyrażenie wykona się poprawnie gdyż dla każdej klasy kompilator automatycznie definiuje konstruktor kopiujący, umożliwiający utworzenie pewnego obiektu tej klasy biorąc za punkt wyjścia inny obiekt tej samej klasy (duplikujący obiekt "pole po polu")
  - prototyp takiego domyślnego konstruktora wygląda następująco:

```
AClass (const AClass &);
```

  - gdy jednak działamy na klasie z polami wskaźnikowym (klasa `String`), to wystąpią analogiczne problemy jak opisane wcześniej z operatorem przypisania (tam rozwiązaniem było przeciążenie operatora `=` poprzez definicje funkcji `operator=()`)
  - rozwiązaniem obecnego problemu będzie odpowiednie przedefiniowanie konstruktora kopiującego

- konstruktor kopiujący

```
class String
{
public:
    String( const char* _str );
    String( int _size );
    String( const String& _string );            // Konstruktor kopiujący.
    ~String();
    void set( const char* _str );
    const char* get() const;
    void print() const;
    String& operator=( const String& _string ); // Operator przypisania.
private:
    int size;
    char* str;
};

//Konstruktor kopiujacy.
String::String( const String& _string )
{
    size = _string.size;
    str = new char[ size + 1 ];  // Przydziela potrzebna pamięć.
    strcpy( str, _string.str );  // Kopiuje zawartość łańcucha.
}
```

Nigdy nie przekazuj do konstruktora kopiującego argumentu przez wartość.
Wtedy wywołanie konstruktora kopiującego zakończy się wejściem w nieskończoną pętlę.
Dlaczego? Gdyż wykonywana jest kopia argumentu, a to spowoduje ponowne wywołanie konstruktora.

- konstruktor kopiujący i przeciążanie operatora =
  - operacji takich jak poniżej można w sprytny sposób zabronić (zakładamy, że istnieje ku temu powód)

```
s2 = s1;         // ZABRONIONE
String s4 = s3;  // ZABRONIONE
```

  - w tym celu wystarczy zadeklarowanie funkcji `operator=()` oraz konstruktora kopiującego BEZ ich definiowania (implementowania) w programie. Dodatkowym zabezpieczeniem będzie umieszczenie ich w sekcji `private`

- konstruktor kopiujący i przeciążanie operatora =

```
class String {
public:
    String( const char* _str );
    String( int _size );
    ~String();
    void set( const char* _str );
    const char* get() const;
    void print() const;
private:
    String& operator=( const String& _string ); // Operator przypisania.
    String( const String& _string );            // Konstruktor kopiujący
    int size;
    char* str;
};

void main() {
    String s1( "Pierwszy lancuch" );
    String s2( "Drugi lancuch" );
    String s3( "Trzeci lancuch" );

    String s4 = s3;  // BLAD funkcja String::String(const String&)
                     // jest niedostępna
    s3 = s2 = s1;    // BLAD funkcja String:operator=(const String&)
                     // jest niedostępna }
```


- konstruktor kopiujący i przeciążanie operatora =

konstruktor(y),
destruktor,
przeciążony operator przypisania,
własny konstruktor kopiujący

są zwykle tworzone dla każdej klasy, która dynamicznie alokuje pamięć