

C++ - klasy

Związki z podręcznikami o modelowaniu obiektowym

- Struktury i klasy zawierające metody nazywane są *abstrakcyjnymi typami danych*
- Obiekty to „zmiennie” 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

© UKSW, WMP, SNS, Warszawa

84

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 wyczerpieniowe.

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 składowych obiektu zapisać wartość lub odczytać z nich wartość. Możemy też wywołać metodę należącą do obiektu.

© UKSW, WMP, SNS, Warszawa

85

85

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.

© UKSW, WMP, SNS, Warszawa

86

86

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)

© UKSW, WMP, SNS, Warszawa

87

87

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);
    };
};
```

© UKSW, WMP, SNS, Warszawa

88

88

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..

© UKSW, WMP, SNS, Warszawa

89

89

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;
```

© UKSW, WMP, SNS, Warszawa

90

90



Prawa dostępu do składowych klasy

PRAWA PRZYJACIÓŁ KLASY

© UKSW, WMP, SNS, Warszawa

91

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;
}
```

© UKSW, WMP, SNS, Warszawa

92

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;
}
```

© UKSW, WMP, SNS, Warszawa

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

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.

© UKSW, WMP, SNS, Warszawa

94

94

C++ - klasy

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

```
class X {
    private:
        int i;
    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
};
```

© UKSW, WMP, SNS, Warszawa

95

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**.

© UKSW, WMP, SNS, Warszawa

96

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..

© UKSW, WMP, SNS, Warszawa

97

97

C++ - klasy

Podsumowując dotychczasowe informacje:

- Nazewnictwo w C++ (w odniesieniu do ANSI C):
 - składowe klasy – odpowiedniki „zmiennych” i „funkcji”.
 - „zmiennie” zadeklarowane w klasie nazywane są **polami** klasy a „funkcje” – **metodami**
 - „zmiennie” zadeklarowane na podstawie typu klasy to **obiekty**

© UKSW, WMP, SNS, Warszawa

98

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.

© UKSW, WMP, SNS, Warszawa

99

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; }
...
```

© UKSW, WMP, SNS, Warszawa

100

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; }
...
```

© UKSW, WMP, SNS, Warszawa

101

101



Inicjalizacja obiektu

KONSTRUKTORY

© UKSW, WMP, SNS, Warszawa 102

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 inicjalizujących 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 wykonują się za każdym razem kiedy tworzony jest kolejny obiekt. Mogą one inicjalizować pola obiektu.

© UKSW, WMP, SNS, Warszawa 103

103

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.

© UKSW, WMP, SNS, Warszawa 104

104

C++ - klasy

Przykład deklaracji:

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

Po czym rozpoznajemy konstruktor?

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

Definicja konstruktora:

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

© UKSW, WMP, SNS, Warszawa 105

105

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?

© UKSW, WMP, SNS, Warszawa 106

106

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.

**Uwaga: składnia X a1(); jest myląca dla kompilatora – traktuje ją jako deklarację funkcji.*

© UKSW, WMP, SNS, Warszawa 107

107

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.

© UKSW, WMP, SNS, Warszawa

108

108

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?

© UKSW, WMP, SNS, Warszawa

109

109

C++ - klasy

Konstruktor kopiujący

Kiedy trzeba 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
    ...
};
```

© UKSW, WMP, SNS, Warszawa

110

110

C++ - klasy

Konstruktor kopiujący

```
class MojaKlasa {
    char **email;
    int ile;
public:
    MojaKlasa(int x);

};
MojaKlasa::MojaKlasa(int x) {
    email = new char*[ile = x];
    for (int i=0;i<ile;i++) email[i] = new char[100];
}
```

© UKSW, WMP, SNS, Warszawa

111

111

C++ - klasy

Konstruktor kopiujący

```
class MojaKlasa {
    char **email;
    int ile;
public:
    MojaKlasa(int x);
    MojaKlasa(MojaKlasa& mk); // konstruktor kopiujący
};
MojaKlasa::MojaKlasa(int x) {
    email = new char*[ile = x];
    for (int i=0;i<ile;i++) email[i] = new char[100];
}
MojaKlasa::MojaKlasa(MojaKlasa& mk) {
    email = mk.email; // czy to jest OK?
}
```

© UKSW, WMP, SNS, Warszawa

112

112

C++ - klasy

Konstruktor kopiujący

```
MojaKlasa::MojaKlasa(MojaKlasa& mk) {
    email = new char*[mk.ile];
    ile = mk.ile;
    for (int i=0;i<ile;i++) {
        email[i] = new char[100];
        strcpy(email[i],mk.email[i]);
    }
}
```

© UKSW, WMP, SNS, Warszawa

113

113

C++ - klasy

Domyślny konstruktor kopiujący

Przy tworzeniu kopii obiektów ZAWSZE używany jest konstruktor kopiujący. Jeżeli programista nie zadeklarował dla klasy/struktury konstruktora kopiującego, kompilator zrobi go sobie sam!

Jak będzie działał?

Jeżeli składowe klasy są typu wbudowanego (`int`, `float`,...), kompilator w konstruktorze kopiującym umieści kod kopiujący „bajt po bajcie”.

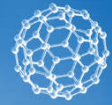
Jeżeli składowe reprezentują obiekty, kompilator wywoła rekurencyjnie konstruktory kopiujące tych obiektów składowych (jeżeli obiekty składowe będą zawierały inne obiekty, to ich konstruktory kopiujące również zostaną wywołane). Jeżeli składowe-obiekty nie mają zdefiniowanych konstruktorów kopiujących, kompilator utworzy je wg tej samej reguły.

Jest to tzw. *injalizacja za pośrednictwem elementów składowych*.

© UKSW, WMP, SNS, Warszawa

114

114



Usuwanie obiektu

DESTRUKTOR

© UKSW, WMP, SNS, Warszawa

115

115

C++ - klasy

Końcowe porządki

- Inną przyczyną wielu błędów w programach jest brak zwolnienia lub nieprawidłowe zwolnienie zmiennych dynamicznych zaalokowanych na początku lub w trakcie działania programu.
- Obiekt zawiera z reguły szereg pól, które mogą być wskaźnikami przechowującymi adresy zmiennych dynamicznych; przed usunięciem obiektu zmienne dynamiczne należy zwolnić.
- Umieszczanie instrukcji zwalnających zmienne dynamiczne w każdym miejscu, gdzie może być usuwany istniejący obiekt, mogłoby gmatwać kod programu i zwiększać znacznie jego rozmiar.
- Aby ułatwić sprzątnięcie dano programiście możliwość przypisania dowolnego zbioru instrukcji do czynności usuwania istniejącego obiektu. Te instrukcje wykonują się za każdym razem kiedy usuwany jest obiekt.

© UKSW, WMP, SNS, Warszawa

116

116

C++ - klasy

Przykład deklaracji:

```
class X {
    int* tab;
public:
    ~X(); // destruktor
};
```

Po czym rozpoznajemy destruktor?

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

Definicja destruktora:

```
X::~X() { if (tab!=NULL) delete []tab; }
```

© UKSW, WMP, SNS, Warszawa

117

117

C++ - klasy

- Destructur jest wywoływany zawsze podczas destrukcji obiektu przez system (destruktor obiektu nie niszczy, ale jest wykonywany tuż przed zniszczeniem)
- Obiekty utworzone jako zmienne lokalne są niszczone automatycznie w momencie opuszczenia bloku w którym zostały zadeklarowane przez sterowanie programu
- Obiekty dynamiczne utworzone poleceniem `new` nie są niszczone automatycznie – muszą być niszczone poleceniem `delete`
- Destructur musi być bezparametrowy, co oznacza, że nie może być przeciążony (nie można tworzyć wiele destruktów)

© UKSW, WMP, SNS, Warszawa

118

118



restrykcyjne reguły zgodności typów

TWORZENIE I USUWANIE OBIEKTÓW

119

C++ - tworzenie obiektów

Ogólnie mówiąc, kiedy w C++ tworzony jest obiekt, zawsze zachodzą **dwa** procesy:

1. przydzielana jest mu pamięć:
 - w obrębie obszaru danych statycznych – zanim rozpocznie się praca programu
 - na stosie – kiedy zostanie osiągnięty określony punkt realizacji programu (np. nawias klamrowy otwierający)
 - na stercie – kiedy wywołane zostanie polecenie utworzenia zmiennej dynamicznej
2. wywołany jest konstruktor inicjalizujący tę pamięć

© UKSW, WMP, SNS, Warszawa

120

120

C++ - tworzenie obiektów

Funkcje `malloc` i `calloc` są bardzo prymitywne. Aby utworzyć na stercie instancję klasy a potem ją usunąć, należałoby napisać coś w tym rodzaju:

```
class Obj {
    ...
};
int main() {
    Obj *obj = (Obj*)malloc(sizeof(Obj)); // alokacja pamięci
    if (obj==0) {
        perror("nie udało się zaalokować pamięci");
        exit(EXIT_FAILURE);
    }
    obj->initialize(); // zamiast wywołania konstruktora
    ...                // tu instrukcje naszego programu
    obj->destroy();    // zamiast wywołania destruktora
    free(obj);        // zwalnianie pamięci
    // TAK NIE NALEŻY TWORZYĆ OBIEKTÓW DYNAMICZNYCH!
```

© UKSW, WMP, SNS, Warszawa

121

121

C++ - tworzenie obiektów

W C++ zapominamy o `malloc` i `calloc`.

Ponieważ w C++ *wszystko* ma swój konstruktor, który musi być zawsze wywołany, używanie `malloc` i `calloc` naruszałyby tę zasadę, ponieważ one nie dają możliwości wywołania konstruktorów i destruktorów.

- Rozwiązaniem jest połączenie w jeden operator `new` wszystkich działań koniecznych do utworzenia obiektu.
- Podczas generowania obiektu dynamicznego poprzez wyrażenie `new` przydziela się na stercie niezbędną ilość pamięci i wywołuje dla niej właściwy konstruktor, np. :

```
MojTyp *MTP = new MojTyp(1,2,3);
```

© UKSW, WMP, SNS, Warszawa

122

122

C++ - tworzenie obiektów

Operator `new`

1.najpierw alokuje stosowny obszar pamięci.

2.Dopiero kiedy alokowanie zakończy się pomyślnie, przystępuje do wywołania konstruktora.

- Nie ma potrzeby sprawdzania, czy alokacja się powiodła.

• A jeżeli się nie powiodła – wywoływana jest specjalna funkcja `new handler`. Jej zadaniem jest zgłoszenie wyjątku (o wyjątkach będzie więcej informacji w dalszej części wykładu)

• Rezultatem działania `new` jest:

1. zainicjalizowany obiekt, albo
2. obiekt-wyjątek informujący o problemie.

© UKSW, WMP, SNS, Warszawa

123

123

C++ - tworzenie obiektów

- Wyrażeniem komplementarnym do `new` jest `delete`
- Wyrażenie `delete` najpierw wywołuje destruktor, a następnie zwalnia przydzieloną uprzednio pamięć, np. :

```
delete MTP;
```

- Operator `delete` może być wywołany wyłącznie w stosunku do obiektu utworzonego wcześniej za pomocą `new`
- Jeżeli wskaźnik usuwany za pomocą `delete` jest NULL, to nic się nie stanie.

Z tego względu niektórzy zalecają przypisanie wskaźnikowi wartości NULL zaraz po usunięciu obiektu, żeby w przypadku próby podjęcia usuwania drugi raz dla tego wskaźnika uniknąć problemów.

© UKSW, WMP, SNS, Warszawa

124

124

RESTRYKCYJNA POLITYKA
KONTROLI ZGODNOŚCI TYPÓW

© UKSW, WMP, SNS, Warszawa

125

125

C++ - zgodność typów

? Dlaczego w C++ mamy dużo bardziej restrykcyjną politykę kontroli zgodności typów?

W języku C funkcja `malloc` zwraca wskaźnik typu `void*` czyli po prostu adres w pamięci. Dlatego można było np. napisać:

```
struct ABC {
    ...
};
struct ABC *ptr = malloc( sizeof(ABC) );
```

Akceptowana może być konwersja różnych typów adresowych występujących po lewej i prawej stronie operatora przypisania.

W C++ taka konwersja jest niedozwolona.

© UKSW, WMP, SNS, Warszawa

126

126

C++ - zgodność typów

Kontrola typów - przykład:

```
class Obj {
    ...
};
Obj *ptr1 = new Obj;
void* ptr2 = ptr1;
ptr1 = new Obj;
... // tu instrukcje naszego programu
delete ptr1;
delete ptr2;
```

Problem: skąd operator `delete` ma wiedzieć jaki destruktor ma wywołać dla obiektu wskazywanego przez `ptr2`?

© UKSW, WMP, SNS, Warszawa

127

127

C++ - zgodność typów

Kontrola typów - przykład:

Ponieważ typ `void*` nie wskazuje na żaden konkretny obiekt, operator `delete` zwolni tylko pamięć, a żadnego konkretnego destruktora nie będzie próbował wywołać.

Ewentualna konwersja do typu `void*` nie budzi zastrzeżeń – kompilator zakłada, że programista, gubiąc informację o typie obiektu, wie co robi:

```
Obj *ptr1 = new Obj;
void* ptr2 = ptr1;
```

Ale próba konwersji w drugą stronę jest już traktowana w C++ jako błąd – nie tylko zgubimy wtedy informację o prawdziwym typie obiektu, ale wprowadzimy fałszywą informację o tym, że jest innego typu:

```
ptr1 = ptr2;
```

© UKSW, WMP, SNS, Warszawa

128

128

INNE SPOSOBY INICJALIZACJI SKŁADOWYCH OBIEKTU

© UKSW, WMP, SNS, Warszawa

129

129

C++ - klasy

Inicjalizacja agregatowa zmiennej tablicowej

```
int a[5] = {1,2,3,4,5};
```

Struktury są również agregatami, dlatego:

```
struct X {
    int i;
    double f;
    char c;
};
```

```
X x1 = {1, 2.2, 'c'};
```

Ale tylko pod pewnymi warunkami.

© UKSW, WMP, SNS, Warszawa

130

130

C++ - klasy

Inicjalizacja agregatowa obiektów

jest możliwa tylko kiedy spełnione są łącznie następujące warunki:

1. klasa nie zawiera składowych `private` ani `protected`,
2. programista nie zaimplementował w niej żadnych konstruktorów,
3. klasa po niczym nie dziedziczy (nie ma swojej klasy bazowej),
4. klasa nie ma metod polimorficznych.

© UKSW, WMP, SNS, Warszawa

131

131

C++ - klasy

Tablice struktur

```
struct X {  
    int i;  
    double f;  
    char c;  
};  
  
X x2[3] = {{1, 2.2, 'c'}, {2, 1.1, 'b'}};
```

Trzeci element tablicy zostanie zainicjowany wartością zerową dla każdego z jego pól