Klasy, struktury, unie

- W C++ niema różnicy podstawowej pomiędzy klasą a strukturą. Struktura może zawierać nie tylko dane, a i funkcje. Różni się struktura od klasy tym, że domyślne składowe klas są prywatne, a struktur – publiczne. Żeby element struktury zadeklarować jako składową prywatną, należy użyć kluczowe słowo *private*. Chociaż struktura może zawierać i metody, zwykłe programiści używają struktury dla przedstawienia danych, a dla przedstawienia obiektów – klasy.

- Instrukcja *union* podobnie jak struktura może być wykorzystywana do definiowania klas – unie mogą zawierać zarówno funkcje składowe, jak i dane. Mogą mieć konstruktor i destruktor. Najważniejsze jest umieszczanie wszystkich danych w tym samym obszarze pamięci. Składowe unii domyślne są publiczne.

Example 12.

- Na różnice od klas unie nie mogą dziedziczyć ani żadnych klas, ani typów. Unia nie może być klasą bazową.

- Składowymi unii nie mogą być:
  - funkcje wirtualne
  - zmienne statyczne
  - zmienne referencyjne

- obiekt z przeciążonym operatorem = .
- obiekty, dla których w sposób jawny zdefiniowano konstruktor lub destruktor.

➢ Unie anonimowe – nie zawierają nazwę typu. Nie można deklarować obiektów typu, definiowanego przez taką unię. Unie anonimowe informują kompilator, że ich zmienne składowe będą znajdowali się w tym samym obszarze pamięci. Dostęp do składowych unii anonimowej odbywa się bezpośrednio:

```
union
{
    int i;
    char ch[4];
};

i = 4;
ch[0] = 'X';
```

- Globalne unie anonimowe mogą być zadeklarowane tylko jako *static.*
- Unii anonimowe nie mogą zawierać składowych prywatnych.
- Nazwy składowych unii anonimowej nie powinni konfliktować z innymi nazwami zmiennych w zasięgu deklaracji.

Funkcje wplatane (inline)

- To są krótkie funkcje, które w rzeczywistości nie są wywoływane. Zamiast tego ich ciało pozostaje wbudowano w kod programu w miejscu jej wywołania.

- Taka technika nadaje możliwość obejść mechanizm wywołania funkcji, związany z procedurą tworzenia stosu argumentów formalnych, przekazania wartości argumentów aktualnych, zwracania wartości (return value), itp. Dla tego inline – funkcji będą wywołane znacznie szybszej od wywołania zwykłego. To jest zaletą funkcji wplatanych.

- Wadą funkcji wplatanych jest to, że jeśli kod takich funkcji jest długi i funkcje wplatane są wywołane zbyt często, objętość programu będzie drastycznie rosła.

Definicja inline funkcji:

```
#include <iostream>
using namespace std;

inline int even(int x)
{
        return (x%2);
}
```

```
void main()
{
        int retv = even(10);  //retv = 0;
}
```

- Inline funkcja musi być zdefiniowaną przed pierwszym wywołaniem. (Inaczej kompilator nie wie, który kod ma wplatać).

- Specyfikator inline nie jest poleceniem, tylko żądaniem – kompilator może jego ignorować. Przy ignorowaniu inline funkcja będzie wywołana jako zwykła.

- W zależności od typu kompilatora (patrz dokumentacje) mogą być szereg ograniczeń:

    - funkcja nie musi być rekursywną
    - nie musi zawierać zmienne statyczne
    - nie musi mieć instrukcje pętli lub *switch* lub *goto*

- Jeśli takie ograniczenie istnieje, zamiast inline będzie generowane wywołanie zwykłe (standard C++).

- Kompilatorzy zaawansowane mogą zgłosić błąd, jawnie informując developera, że jako inline podana funkcja nie może być skompilowana (VS compilers).

- **Funkcje – składowe klasy również mogą być wplatane.**

```
#include <iostream>
using namespace std;

class samp
{
        int i;
public:
        samp(int a);
        int get_i();
};

samp::samp(int a)
{
        i = a;
}

inline int samp::get_i()
{
        return i;
}
```

- **Funkcji inline można używać w deklaracji klasy:**

```
class  samp
{
      int  i, j;
public:
      samp(int  a, int b)        //kilka linij kodu inline - funkcje
      {
            i = a;
            j = b;
      }

      int get_i() { return i; }  //inline funkcja umieszczona w jednej linii kodu
      int get_j() { return j; }  //to jest najbardziej wspierany styl śród programistów
}
```

Przy takim stylu specyfikator inline jest pominięty.

- Funkcje inline można przeciążyć, przy tym wszystkie realizacje mają być też inline.

Domyślne argumenty funkcji

W C++ dopuszcza się przypisywanie parametrom funkcji pewnych wartości domyślnych. Jeśli przy wywołani funkcji taki argument nie jest zadany, jego wartość będzie pobrana jako wartość domyślna. To jest postać ukryta przeciążenia funkcji.

```
void fun(int i =0; int j=0);

int main()
{
     ……….
     fun();          //OK,  is passed  i=0; j=0
     fun(5);         //OK,  is passed  i=5, j=0
     fun(24, 32);  //OK,  is passed  i=24, j=32
     return 0;
}

void fun(int i, int j)
{
     ……….
}
```

- Dla podanego przykładu jest niemożliwe pierwszy parametr przekazać domyślnie, a drugi – nie.

- Argumenty domyślne muszą być zadane albo w prototypie funkcji, albo w definicji, jeśli pierwsze wywołanie tej funkcji idzie po definicji. Nie wolno zadawać jednocześnie argumenty domyślne i w prototypie, i w definicji.

- Wszyscy parametry domyślne muszą być umieszczone sprawa od parametrów, przekazywanych w sposób zwykły:

```
void fun(int i, int j=0);        //OK.
void fun(int i=0, int j); //błąd: parametr domyślny jest zlewa od
                          //parametru zwykłego
```

- Argumenty domyślne myszą być stałymi lub parametrami globalnymi.

Argumenty domyślne a funkcje przeciążone.

Przykład: Pole kwadratu: S=a*a; Pole prostokąta: S = a*b. Zamiast tego, żeby pisać funkcje przeciążone typu

```
double get_area(double a, double b);    //pole prostokąta
double get_area (double a);             //pole kwadratu
```

można napisać tylko jedną funkcje:

```
double get_area (double a, double b = 0)
{
    if(b != 0.0)
        return  a*b;    // pole prostokąta
    else
        return  a*a;    // pole kwadratu
}

int main()
{
    double aa = 10.0;
    cout <<  " area of square    : "  << get_area(aa) << endl;
    cout <<  " area of rectangle: "  << get_area (aa, 20.0) << endl;
 }
```

- Przekazanie argumentów domyślnych konstruktorom klas:

```
class myclass
{
        int i;
public:
        myclass(int ii = 0);
        ……….
};

int main()
{
        //The constructor with a default argument is used to create both: initialized and
        //uninitialized objects.
        myclass ob(10);     //ob::i = 10 – parameterized constructor is required.
        myclass *ptr_ob;

        try
        {
             ptr_ob = new myclass [20];   //default constructor is required.
        }
        catch(bad_alloc aa)
        {
         ……
        }
……..
        delete [ ] ptr_ob;
        return 0;
}
```

- Konstruktor kopiujący z argumentami domyślnymi: można tworzyć konstruktor kopii z argumentami dodatkowymi pod warunkiem, że wszystkie argumenty dodatkowe są argumentami domyślnymi:

```
class myclass
{
        int i;
public:
        myclass(int ii = 0) : i(ii)  { }
        myclass(const myclass &aa, double b = 0, char str[] = "Hello!");
        int get_i() { return i; }
        void set_i(int ii) { i = ii; }
};

myclass::myclass(const myclass &aa, double b, char *str)
{
        i = aa.get_i();
        cout << “copy constructor = " << i << " b = " << b << " str = " << str << endl;
}

int main()
{
        myclass tt(10);
        myclass tt1 = tt;   //copy constructor is called
        ……….
}
```

Trzeba pamiętać, że nadużycie przy wykorzystaniu argumentów domyślnych doprowadzi do komplikowania kodu oraz czytelności programu.

Można używać argumenty domyślne w takich przypadkach: mamy duży program, w którym wielokrotnie jest używana funkcje

```
void draw_rect(double x1, double y1, double x2, double y2);
```

Po upłynięciu kilku lat okazało się niezbędnym dołączenie do listy argumentów formalnych flagi int my_flag. Dla tego, żeby nie wprowadzać zmiany w starej części kodu, która nie używa tej flagi (a może źródła tej części kodu i nie są dostępne dla nas) modyfikujemy tak:

```
void draw_rect(double x1, double y1, double x2, double y2, int my_flag = 0);
```

Funkcje draw_rect przepisujemy tak, żeby przy my_flag = 0 jej działanie było takie same, jak w wersji poprzedniej. Wtedy poprzednia część kodu nie potrzebuje żadnej zmiany, a nowy kod można pisać z wykorzystaniem nowych możliwości funkcji draw_rect.

Jeszcze jeden przykład przeciążenia funkcji:

```
int AfxMessageBox( LPCTSTR lpszText,  UINT nType = MB_OK,  UINT nIDHelp = 0 );

int AFXAPI AfxMessageBox( UINT nIDPrompt,  UINT nType = MB_OK,  UINT nIDHelp = (UINT ) -1 );
```

Dla WINDOWS: UINT – unsigned int; DWORD - unsigned long;
LPCTSTR lpszText – wiersz tekstowy (const char *), wyświetlany na dialogu.

nIDPrompt – ID resursu (string table) – ten samy wiersz tekstowy może być umieszczony w resursach (tabele wierszy tekstowych).

nType – typ przycisków (OK; Yes, No; Yes, No, Cancel; Abort, Retry, Ignore; ..).

nIDHelp – kontekst pomocy (jeśli istnieje).

Example: **resource**.

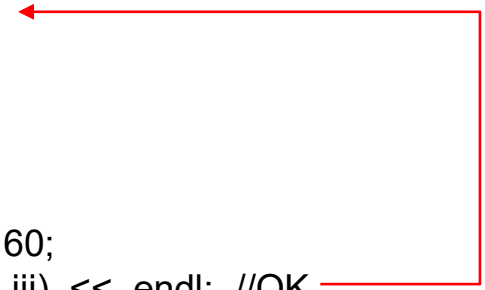
Funkcje przeciążone a niejednoznaczność - sytuacje, kiedy kompilator nie jest w stanie podjąć decyzje, którą z dwóch (lub więcej) funkcji przeciążonych ma wybrać. Skończy to błędem kompilacji.

Najczęściej to powstaje przy automatycznej konwersji typów:

```
float fun(float  a)
{
        return a/20.0;
}

double fun(double a)
{
      return a/20.0;
}

int main()
{
      float        x = 40.0;
      double     y = 40.0;
      cout  <<  fun(x)  <<  endl;      //OK – float
      cout  <<  fun(y)  <<  endl;      //OK – double
      cout  <<  fun(40)  << endl;     // do którego typu konwertować liczbę całą ? - błąd
      cout  <<  fun((double)40)  << endl;     // OK – stosujemy jawną konwersje typu
}
```

Argumenty domyślne mogą powodować niejednoznaczność:

```
int fun(int  ii)
{
        cout  <<  ii << endl;
        return ii;
}

int fun(int ii, int jj = 0)
{
        cout  <<  ii << jj  << endl;
        return ii;
}

int main()
{
       int  iii = 40, jjj = 60;
       cout  <<  fun(iii, jjj)  <<  endl;  //OK,
       cout  <<  fun(iii)  <<  endl;      //błąd: która funkcja ma być wybrana?
       ……
}
```



- **Pobieranie adresu funkcji przeciążonych.**

W języku C funkcja o podanej nazwie jest unikalna (brak przeciążenia).

W języku C++ sposób deklarowania wskaźnika do funkcji decyduje o to, która z przeciążonych funkcji będzie wywołana:

```
double funtt(double aa, double bb);
int funtt(int ii, int kk);

double (*ptr_fun_d)(double aa, double bb);
int (*ptr_fun_i)(int ii, int kk);

int main()
{
        ptr_fun_d = &funtt;
        ptr_fun_i  = &funtt;
        cout << " double " << (*ptr_fun_d)(10.0, 10.5) << endl;
        cout << " int    "     << (*ptr_fun_i)(15, 16) << endl;
……
}

double funtt(double aa, double bb) {  return bb - aa;  }

int funtt(int ii, int jj) {  return jj - ii;  }
```