Języki programowania

Typy

Co to jest typ?

- Każda nazwa w C++ zanim zostanie użyta musi zostać zadeklarowana.
- Deklarujemy, że dana nazwa opisuje obiekt jakiegoś typu – informuje to kompilator, jak ma postępować w przypadku napotkania tej nazwy i jaka jest jego organizacja wewnętrzna (w pamięci komputera).
- Obiekty różnych typów przechowywane są w pamięci komputera w różny sposób, różnie też można z nimi postępować. Stąd konieczność podania typu obiektu.

- Składnia definicji/deklaracji:

```
typ nazwa_zmiennej;
```

Jakie w C++ mamy typy?

- Podział pierwszego typu:
  - typy fundamentalne,
  - typy pochodne – są jakby wariacjami na temat typów fundamentalnych.
- Podział drugiego typu:
  - typy wbudowane – czyli te które udostępnia język C++,
  - typy zdefiniowane przez użytkownika – czyli typy, które można sobie samemu zaprojektować.

- Możliwość definiowania własnych typów jest jedną z największych zalet języka C++ i stanowi o jego sile.

Typy fundamentalne (1)

- Typy reprezentujące liczby całkowite:

Typ ze znakiem		Typ bez znaku	
nazwa typu	rozmiar	nazwa typu	rozmiar
`char`	1 bajt	`unsigned char`	1 bajt
`short int (short)`	2 bajty	`unsigned short int (unsigned short)`	2 bajty
`int`	4 bajty	`unsigned int`	4 bajty
`long int (long)`	4 bajty	`unsigned long int (unsigneg long)`	4 bajty

- Jawne reprezentowanie typu ze znakiem polega na poprzedzonemu nazwy typu słowem **`signed`**.
- Typ **`char`** najlepiej nadaje się do przechowywania i obsługi znaków alfanumerycznych.

Typy fundamentalne (2)

- Typy reprezentujące liczby rzeczywiste:

Nazwa typu	rozmiar	zakres
`float`	4 bajty	±3.4e ±38 (7 cyfr)
`double`	8 bajtów	±1.7e ±308 (15 cyfr)
`long double`	10 bajtów*	±1.2e ±4932 (19 cyfr)

- W systemach Windows NT i Windows 95 typ **`long double`** (80-bitów, 10-bitowa precyzja) jest mapowany bezpośrednio na typ **`double`** (64-bity, 8-bitowa precyzja).

Definiowanie obiektów „w biegu”

- W niektórych językach programowania definicje obiektów powinny znaleźć się na początku, przed wykonywanymi instrukcjami.
- Zasada ta nie obowiązuje w C++! – obiekt możemy definiować między kolejnymi instrukcjami, wtedy, kiedy uznamy, że jest on właśnie teraz potrzebny.
- Jednakże, nawet i w tym wypadku, obiekt musi być znany kompilatorowi przed jego użyciem.

```
#include <iostream.h>
void main() {
   //...
   cout << "Podaj x :”;
   float x;                    // definiujemy dopiero teraz!
   cin >> x;
   //...
}
```

Stałe (1)

- W programach często posługujemy się stałymi.
- Mogą to być:
  - liczby,
  - litery,
  - ciągi znaków (ang. *stringi*).
- Stałych tych używamy np.:
  - by wstawić je do jakichś zmiennych:

```
x = 10.52;
```

  - gdy występują one w wyrażeniach arytmetycznych:

```
i = i + 5;
```

  - gdy chcemy coś z nimi porównać:

```
if(m > 12)
```

Stałe (2)

- Stałe reprezentujące liczby całkowite:

System ósemkowy	System dziesiątkowy	System szesnastkowy
`00`	`0`	`0x0`
`010`	`8`	`0x08`
`014`	`12`	`0x0C`
`091 (błąd)`	`-`	`-`

- Stałe całkowite traktowane są jako int (chyba, że reprezentują tak wielkie liczby, które nie mieszczą się w int.
- Można wymusić typ stałej całkowitej:

  `20L` lub **`130l`** – wymuszają typ **`long`**,

  `13u` lub **`34U`** – wymuszają typ **`unsigned int`**,

  `34ul` lub **`1Lu`** – wymuszają typ **`unsigned long`**.

Stałe (3)

- Stałe reprezentujące liczby rzeczywiste:
  - zapis normalny, np. `1.23` `.23` `-1.` `0.0`
  - zapis naukowy, np. `8e2` `10.4e-8` `-5.2e3`

- W zapisie stałej wewnątrz nie może pojawić się spacja!

- Stałe takie traktuje się jako **`double`**.

- Jeśli potrzebna jest stała zmiennopozycyjna typu **`float`**, to można ją zdefiniować używając przyrostka **`f`**:

  `3.14159265f` `2.0f` `2.997925f`

Stałe (4)

- Stałe znakowe – reprezentują znaki alfanumeryczne.
- Zapisuje się je ujmując w apostrofy

  `'a'` `'7'` `' '` (spacja) `'\0'` (NULL) `'\n'` (nowa linia)
- Stałe znakowe przechowywane są w pamięci komputera jako liczby, będące kodem ASCII danego znaku.

```
char znak = 'a';      // wartość liczbowa = 61
```

- Znaki sterujące:

Escape Sequence	Represents
\a	Bell (alert) – sygnał dźwiękowy
\b	Backspace
\f	Formfeed – nowa strona
\n	New line – nowa linia
\r	Carriage return – powrót karetki
\t	Horizontal tab – tabulator poziomy
\v	Vertical tab – tabulator pionowy
\'	Single quotation mark - apostrof
\"	Double quotation mark - cudzysłów
\\	Backslash
\?	Literal question mark - pytajnik

Stałe (5)

- Stała tekstowa (*string*) – ciąg znaków ujęty w cudzysłów.

```
"Ala ma kota"                "Kot ma miske\n"
"Miska \tjest pusta"
```

- Stringi przechowywane są w pamięci komputera jako ciąg znaków (**`char`**) zakończony znakiem o kodzie 0 (NULL).

- Jeśli w stringu znajduje się ***n*** znaków, to cały string składa się z ***n+1*** znaków (bo trzeba doliczyć znak o kodzie 0).

string **`"Wisla"`** składa się ze znaków: **`'W'`**, **`'i'`**, **`'s'`**, **`'l'`**, **`'a'`**, **`'\0'`**

Typy pochodne (1)

- Typy pochodne tworzymy na podstawie typów fundamentalnych za pomocą operatorów deklaracji
  - `*` wskaźnik do danego typu,
  - `&` referencja do danego typu,
  - `[]` tablica elementów danego typu,
  - `()` funkcja zwracająca dany typ, o parametrach danego typu,

  oraz mechanizmu definiowania struktury.
- Przykłady:

```
int* a;              // wskaźnik do zmiennej typu int
float t[10];         // 10-elementowa tablica typu float
char& z;             // referencja do zmiennej typu char
double fun(long);// funkcja o parametrze typu long,
                     // zwracająca obiekt typu double,
struct napis { short dl; char* p; };
```

Typy pochodne (2)

- Podstawowa zasada użycia typu pochodnego:

> Deklaracja typu pochodnego odzwierciedla sposób jego użycia

np.:

```
int t[10];                    char *znak;
t[4] = 13;                    ...
x = t[9];                     *znak = 'a';
```

- Operatory `*` i `&` są przedrostkowe, a operatory `[]` i `()` – przyrostkowe.
- Trzeba użyć nawiasów, żeby wyrazić typy, w których pierwszeństwo operatorów jest niewygodne:

```
int* t[10];   // tabl. wskaźników do obiektów typu int
int (*p)[10]; // wskaźnik do tabl. o elem. typu int
```

Deklarowanie kilku obiektów

- W pojedynczej deklaracji można zadeklarować kilka nazw, zamiast jednego.
- Deklaracja zawiera wówczas listę nazw oddzielonych przecinkami:

```
double x, y, z; // double x; double y; double z;
```

- Podczas deklarowania typów pochodnych należy pamiętać, że operatory odnoszą się do pojedynczych nazw:

```
int* p, q;          // int* p;          int q;
int x, *y;          // int x;           int* y;
int t[10], *p;      // int t[10];       int* p;
```

Typ `void`

- Typ **`void`** (ang. *próżny*) składniowo zachowuje się jak typ podstawowy.
- Można go użyć jedynie jako części typu pochodnego – nie ma obiektów typu **`void`**!
- Użycie:
  - wskazanie, że funkcja nie zwraca żadnej wartości:

```
void main();
```

  - wskazanie, że funkcja nie przyjmuje żadnych parametrów:

```
int fun(void);     // int fun();
```

  - definicja wskaźnika do obiektu nieznanego typu:

```
void* ptr;
```

Zmiennej typu **`void*`** można przypisać wskaźnik dowolnego typu.

Zakres ważności nazw (1)

- Deklaracja wprowadza nazwę w pewien zasięg widoczności; tzn. nazwa może być używana w określonej części tekstu programu.
- Czas życia obiektu – od momentu, gdy obiekt został zdefiniowany do momentu, kiedy przestaje istnieć (w pamięci).
- Zakres ważności nazwy – część programu, w której nazwa znana jest kompilatorowi.
- Uwaga: obiekt w danej chwili może istnieć w pamięci komputera, ale może nie być dostępny (bo znajduje się poza zakresem ważności jego nazwy)!

Zakres ważności nazw (2)

- Zakres lokalny – zasięg rozciąga się od punktu deklaracji do końca bloku (nazwa lokalna).

```
void main() {
  //...
  {                  //otwarcie lokalnego bloku
    int i;           // zakres ważności nazwy i
  }                  //zamkniecie lokalnego bloku
  //...
}
```

- Zakres bloku funkcji – zakres ważności etykiety występującej w bloku funkcji (nazwa lokalna).

  Nie można instrukcją **goto** przeskoczyć z/do wnętrza funkcji do/z wnętrza innej!

Zakres ważności nazw (3)

- Zakres pliku – dla nazwy zadeklarowanej poza funkcją lub klasą (nazwa globalna) zasięg rozciąga się od punktu deklaracji do końca pliku zawierającego tę deklarację.

```
double pi = 3.14;  // zmienna globalna

void main() {
   //...
   double pole = pi * r * r;
}
```

- Zakres klasy – nazwa widziana tylko przez metody klasy oraz (jeśli klasa na to pozwala) klas pochodnych oraz funkcji lub klas zaprzyjaźnionych.

Zasłanianie nazw (1)

- Można zadeklarować nazwę lokalną o identyczną, jak istniejąca nazwa globalna (a nie jak inna nazwa lokalna!).
- Nowo definiowana zmienna lokalna przesłania wtedy (w lokalnym zakresie) zmienną globalną.
- Jeśli w lokalnym zakresie odwołamy się do danej nazwy, to kompilator uzna to za odniesienie się do zmiennej lokalnej.

```
#include <iostream.h>
int i = 10;                 // zmienna globalna
void main() {
   cout << „[1] i = " << i << endl;
   {                        // otwarcie lokalnego bloku
      int i = 100;          // zmienna lokalna
      cout << „[2] i = " << i << endl;
   }                        // zamkniecie lokalnego bloku
   cout << „[3] i = " << i << endl;
}
```

Ekran po wykonaniu programu:

```
[1] i = 10
[2] i = 100
[3] i = 10
```

Zasłanianie nazw (2)

- Można odwołać się do zasłoniętej nazwy globalnej, używając operatora zasięgu `::`.
- Nie ma sposobu na odwołanie się do zasłoniętej nazwy lokalnej.

```
#include <iostream.h>
int i = 10;                 //zmienna globalna
void main() {
   cout << „[1] i = " << i << endl;
   {                        // otwarcie lokalnego bloku
      int i = 100;          // zmienna lokalna
      cout << "[2] i = " << i << endl;
      cout << "[3] i = " << ::i << endl;
   }                        // zamkniecie lokalnego bloku
   cout << „[4] i = " << i << endl;
}
```

Ekran po wykonaniu programu:

```
[1] i = 10
[2] i = 100
[3] i = 10
[4] i = 10
```

Modyfikator `const`

- Jeśli chcemy w programie użyć obiektu, którego wartość ma się nie zmieniać możemy zdefiniować obiekt stały.
- Używany wtedy przy definicji modyfikatora **`const`**:

```
const double pi = 3.14159;
```

- Przy definicji obiektu stałego należy obiekt zainicjować (tzn. nadać mu wartość).
- Później, do obiektu **`const`** nie można przypisać żadnej wartości.

Obiekt **`const`** można zainicjować, ale nie można do niego nic przypisać!

Modyfikator `register`

- Używając modyfikatora **`register`** przed zmienną automatyczną typu całkowitego dajemy kompilatorowi do zrozumienia, że bardzo zależy nam na szybkim dostępie do tego obiektu.
- Kompilator może (ale nie musi) uwzględnić tę sugestię i przechowywać ten obiekt w rejestrze procesora.
- Z tego powodu nie można uzyskać adresu tego obiektu, ani odwoływać się do niego przez jego adres.

```
register int licznik = 0;
```

Modyfikator `volatile`

- Użycie przed definicją obiektu modyfikatora **`volatile`** jest dla kompilatora wskazówką, aby w operacjach dotyczących tego obiektu unikał wyszukanej optymalizacji, ponieważ wartość tego obiektu może zmieniać się pod wpływem metod, których kompilator nie potrafi wykryć.
- Za każdym razem, kiedy kompilator odwołuje się do tego obiektu, naprawdę sprawdza jego wartość, a nie polega na znajomości jego wcześniejszej wartości pomimo, że pozornie nic jego wartości nie zmieniło.
- Zazwyczaj obiekt taki jest używany do odczytu danych z zewnętrznych urządzeń pomiarowych.

Instrukcja `typedef`

- Instrukcja **`typedef`** pozwala na nadanie dodatkowej nazwy już istniejącemu typowi.

```
typedef typ nowa_nazwa_typu;
```

Instrukcja **`typedef`** nie wprowadza nowego typu, a jedynie synonim już istniejącego!

- Najczęściej instrukcji tej używa się do uczytelnienia zapisu skomplikowanych typów.

```
typedef unsigned int wiek;
wiek a, b;                          // = unsigned int a, b;
typedef float cena, * wsk_float;
cena x;                             // = float x;
wsk_float py;                       // = float *py;
typedef long (*ptrFun)(char, int);
ptrFun pf;                          // = long (*pf)(char, int);
```

Typ wyliczeniowy `enum`

- Jest to osobny typ dla liczb całkowitych. W zakresie tego typu znajdują się tylko zdefiniowane przez nas stałe.

```
enum figura {kwadrat, okrag, trojkat};
```

- W ten sposób zostaną zdefiniowane trzy stałe, którym zostały nadane kolejne wartości całkowite, zaczynając od *0*:

```
const kwadrat=0; const okrag=1; const trojakt=2;
```

- Można także nadać wartości dla poszczegól-nych stałych:

```
enum figura {kwadrat = 4, okrag = 2, trojkat = 3};
```

- Typ taki można niejawnie przekształcić do typu **`int`**.
- Konwersja odwrotna musi być jawna.

```
figura f = okrag;
switch(f) {
   case kwadrat: ...
   case trojkat: ...
...}
```

Struktury

- Tablica jest agregatem obiektów tego samego typu.
- Struktura jest agregatem obiektów dowolnego typu.

```
struct osoba {
    char imie[15];
    char nazwisko[20];
    unsigned shot wiek;
};
```

definiuje nowy typ o nazwie **osoba**, składający się z elementów opisujących daną osobę.

- Użycie – do obiektów typu strukturowego odwołujemy się za pomocą operatora . (kropka):

```
osoba panX = {"Jan", "Kot", 25};
cout << panX.imie << panX.nazwisko << endl;
panX.wiek = 26;
```

Referencje

- Referencja jest inną nazwą obiektu.
- Referencji używa się głównie do:
  - specyfikowania argumentów funkcji,
  - do przeciążania operatorów.

```
int i = 1;
int& r = i;       // r jest referencja do zmiennej i
int j = r;        // j = 1
r = 2;            // i = 2
```

- Referencja musi być zainicjowana (musi odnosić się do jakiegoś konkretnego obiektu).
- Referencje implementuje się jako stały wskaźnik do obiektu, który w miejscu użycia modyfikowany jest operatorem adresowania pośredniego (*).
- Operator nie działa na referencji, lecz na obiekcie, do którego odnosi się referencja.