Podstawy informatyki (3)

wykład : 15 godz.

ćwiczenia : 15 godz.

Prowadzący: dr inż. Jacek Piątkowski

Języki programowania

- Języki programowania

- dają możliwość zapisu algorytmów w postaci zarówno wygodnej dla człowieka, jak i łatwo przekształcalnej do postaci rozumianej przez komputer.

- pozwalają na tworzenie programów bez wnikania w zawiłości związane z rejestrami, adresami pamięci i cyklami maszynowymi *.

---

* Cykl maszynowy – algorytm wykonywany cyklicznie przez jednostkę sterującą komputera, polegający na:
- pobieraniu rozkazu z pamięci głównej (zgodnie z wartością licznika rozkazów)
- dekodowaniu ciągu bitów znajdujących się w rejestrze rozkazu,
- wykonywaniu operacji wg bieżącego rozkazu.

Języki programowania

- Początki

- kod maszynowy – ciągi bitów reprezentujące określone rozkazy maszynowe.
- opcody (*opcodes*) - zastępowały, wyrażane szesnastkowo, kody rozkazów ich odpowiednikami w postaci nazw mnemonicznych typu :
  - LD (load, przypisz)
  - ADDI (dodaj liczby całkowite)
  - ADDF (dodaj liczby rzeczywiste)
  - SUM

(Programy zapisywano na papierze i tłumaczono ręcznie na kod maszynowy)

Języki programowania

- Asemblery

  - programy komputerowe przeznaczone do tłumaczenia programów zapisanych w postaci mnemonicznej na kod maszynowy.

Języki asemblerowe (uznawane za języki drugiej generacji) pozwalały tworzyć programy dla konkretnej architektury komputera.

- Języki trzeciej generacji takie jak:

  - FORTRAN (*FORmula TRANslator*) ,
  - COBOL (*COmmon Busines Oriented Language*),

oferowały wyodrębniony zestaw wysokopoziomowych konstrukcji pierwotnych tłumaczonych później na język maszynowy.

Instrukcje tych języków nie odwoływały się wprost do cech konkretnego komputera, co nie oznaczało jednak pełnej niezależności od platformy sprzętowej.

Proces tłumaczenia języka

- Interpreter

  - przekłada kod źródłowy na czynności (mogące składać się z grup rozkazów komputera) , które są natychmiast wykonywane .

  zalety : szybkie i łatwe projektowanie programów
  wady : ograniczenia w zakresie tworzenia dużych projektów, ograniczona szybkość wykonywania programu

- Kompilator

  - tłumaczy kod źródłowy na kod rozkazów komputera zapisywany do pliku lub plików.

W przypadku większości kompilatorów właściwy proces generowania kodu wynikowego jest poprzedzony analizą składni kodu źródłowego.

Proces tłumaczenia języka

- Rozłączna kompilacja

  - mechanizm (dostępny np. w językach C, C++) umożliwiający niezależne kompilowanie oddzielnych części programu.

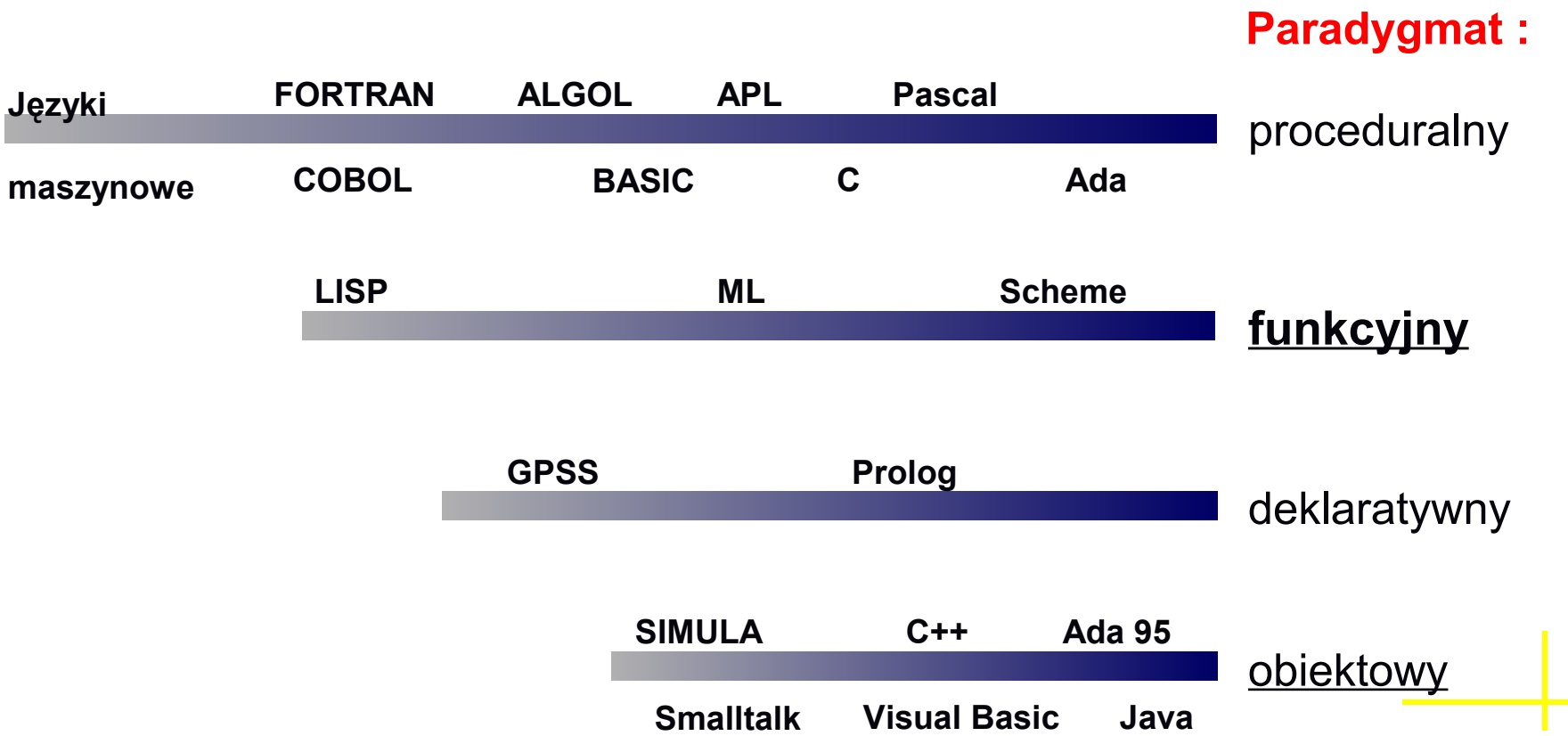
- Program łączący (*linker*)

  - łączy ze sobą poszczególne skompilowane fragmenty programu w jeden program wykonywalny (który może być załadowany i uruchomiony przez system operacyjny).

☞ Rozłączna kompilacja jest szczególnie przydatna do budowy dużych projektów, pozwala tworzyć i testować program partiami.
Zbiory przetestowanych i działających elementów można połączyć w *biblioteki* przeznaczone do używania przez innych programistów .

Paradygmaty programowania

- rozwój języków przebiegał wielotorowo w miarę rozwoju różnych podejść do procesów programowania (paradygmatów programowania).



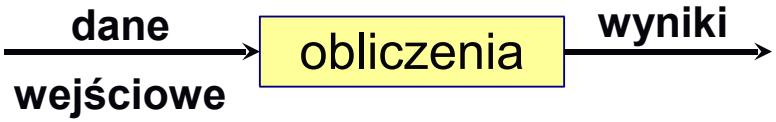
Paradygmaty programowania

- Paradygmat proceduralny
  - jest uznawany jako tradycyjne podejście do procesu programowania,
  - programowanie jest postrzegane jako określenie ciągu poleceń, które przetwarzają dane i generują wyniki,
  - jest z nim zgodny cykl maszynowy (pobierz – dekoduj – wykonaj).

Paradygmaty programowania

- Paradygmat funkcyjny

  - postrzega proces tworzenia programu jako konstruowanie „ czarnych skrzynek ” – funkcji, które traktowane jako konstrukcje pierwotne wykorzystywane są do tworzenia kolejnych coraz bardziej złożonych funkcji.



Rozwiązanie problemu polega na analizie danych wejściowych i wyjściowych i znalezieniu właściwej transformacji, wg której dany należy przetworzyć. Zazwyczaj dana transformacja składa się z wielu mniejszych transformacji.

☞ Jako zaletę paradygmatu funkcyjnego wskazuje się na konieczność podejścia do programowania w sposób modularny co wymusza tworzenie konstrukcji dobrze zorganizowanych, o strukturze blokowej.

Paradygmaty programowania

- Paradygmat deklaratywny

  - bazuje na dopasowaniu określonego problemu do znanego czy też uogólnionego sposobu rozwiązywania problemu,

Wczesne języki deklaratywne były językami specjalnego przeznaczenia, opracowanymi dla wąskiej klasy zastosowań np. symulacji procesów fizycznych, czy też ekonomicznych.

Zadaniem programisty korzystającego z podejścia deklaratywnego jest :

- dokładne określenie na czym polega problem,
- zaimplementowanie algorytmu, który umożliwiającego symulację danego procesu,
- określenie związków miedzy symulowanymi parametrami.

☞ Zaimplementowanie reguł logiki formalnej zwiększyło obszar zastosowań języków deklaratywnych.

Paradygmaty programowania

- Paradygmat obiektowy
  - bazuje na podejściu, w którym dane traktuje się jako aktywne obiekty, a nie jak pasywne jednostki (jak ma to miejsce w paradygmacie proceduralnym).

- Obiekt – to rodzaj zmiennej przechowującej dane i wobec której można można zgłosić żądanie, by wykonała określoną operację na samej sobie. Każdy obiekt posiada typ tzn. jest jest egzemplarzem jakiejś określonej ***klasy*** .

- Klasa – opisuje zbiór obiektów posiadających takie same cechy (elementy danych) i działania (funkcjonalność) i jest w rzeczywistości synonimem typu danych .

Program jest grupą obiektów, które za pomocą komunikatów, przekazują sobie wzajemne informacje o tym co należy zrobić.

Język C++

- C++
  - opracowany na początku lat 70 przez Bjarne`a Stroustrup`a w Laboratoriach Bella firmy AT&T,
  - powstał jako rozwinięci języka C, który rozszerzono o:
    - narzędzia pozwalające definiować i stosować abstrakcyjne typy danych,
    - narzędzia projektowania i programowania obiektowego,
    - ulepszenia istniejących konstrukcji języka C.

Typy danych

- Typy danych określają:
  - sposób użycia pamięci przez tworzony program,
  - sposób wykonywania operacji na poszczególnych fragmentach pamięci.

- W języku C++ :
  - istnieją zdefiniowane pierwotnie podstawowe typy danych (typy wbudowane),
  - istnieje możliwość tworzenia typów abstrakcyjnych,
  - dostępne są trzy podstawowe typy abstrakcyjne (string, vector i complex) .

Typy wbudowane

- Typ całkowity
  - `char` – znak lub mała liczba całkowita (1 bajt),
  - `short, int, long` – liczby całkowite
    zajmujące odpowiednio pół słowa, jedno słowo, jedno lub dwa słowa
    (t.j. 2, 4, 4 lub 8 bajtów, przy czym dla maszyn 32-bitowych
    rozmiar typów `int` i `long` jest zazwyczaj taki sam)

- Modyfikatory
  - `signed` – ze znakiem,
  - `unsigned` – bez znaku , (np. `unsigned int`)

  określają sposób interpretacji najbardziej znaczącego bitu.

Jeżeli liczba całkowita należy do typu ze znakiem to najbardziej znaczący bit służy jako *bit znaku* (**0** – liczba dodatnia , **1** – liczba ujemna) .

Przykładowo :

`char` – wartości od –128 do 127

`unsigned char` – wartości od 0 do 255

Typy wbudowane

- Typ zmiennopozycyjny
  - `float, double, long double` – liczby rzeczywiste o różnej dokładności zajmujące odpowiednio jedno, dwa, trzy lub cztery słowa (dla maszyn 32-bitowych: 4, 8 i 12 bajtów)

- Typ logiczny
  - `bool` – wartości logiczne posiadające dwa opisywane dwoma wbudowanymi stałymi *true* lub *false* , które można przekształcić w wartości całkowite **1** lub **0**
    (dla maszyn 32-bitowych: 1 bajt)

`true, false` – są to ***słowa kluczowe***
(są *literałami* typu logicznego `bool`)

Stałe

❖Stałe definiowane za pomocą dyrektywy preprocesora

`#define`

Przykład:

```
#define PI 3.14159
```

- każde wystąpienie w kodzie programu słowa **`PI`** jest zastępowane wartością `3.14159`,
- nie dokonywana jest żadna kontrola typu,
- nie można określić adresu a zatem
  nie można przekazać *wskaźnika* ani *referencji* do `PI`
- znaczenie `PI` rozciąga się od miejsca, w którym zostało zdefiniowane do końca pliku z kodem źródłowym
  (preprocesor nie rozróżnia *zasięgów*)

☞ zaleca się, by zamiast dyrektywy `#define`

używać *modyfikatora* `const`.

Stałe

- Literały – tzw. *stałe dosłowne* – dane występujące w programie w sposób bezpośredni

Przykład:

liczba `2.71`,

napis `"START"`

- każdy literał jest określonego typu,
- literały są nieadresowalne – ich wartości zajmują konkretne obszary pamięci, nie są jednak dostępne adresy tych obszarów.

Stałe

- Literały całkowite – (domyślnie typu `int`)
  - mogą być wyrażone jako liczby dziesiętne, ósemkowe, szesnastkowe
    (stałe zaczynające się od 0 są traktowane liczby ósemkowe,
    zaczynające się od 0x są traktowane jako liczby szesnastkowe)

Przykład:

- dziesiętnie: `10; 128u; 2048UL; 1L; 102Lu;`
  `( u – unsigned, L – long , UL – unsigned long )`
- ósemkowo: `017; 0123;` $(15, 83)_{10}$
- szesnastkowo: `0x2B; 0x1fe;` $(43, 510)_{10}$

- Literały zmiennopozycyjne – (domyślnie typu `double`)

przykład:

```
0.0; 3.14159F; 2.71f;  20.48L; 5e9; 15.6E-3
( f – float , L –long double )
```

Stałe

- Literały logiczne – (typu `bool`)

Przykład:

```
true; false;
```

- Literały znakowe – (typu `char`)

Przykład:

```
'a';  '2';  ',';  ' ';
```

niedrukowane znaki specjalne `'\n';  '\t';  '\v';  '\\';`

ósemkowo : `'\0';  '\7';  '\14';  '\062';`

szesnastkowo : `'\x0';  '\x7';  '\Xa';  '\x32';`

Stałe

- Literały napisowe – (typu `const char[]`)

Przykład:

```
"";  "a";  "To\ttaki\tprosty\ntekst"

"to literał \
 zapisany w \
 trzech wierszach"
```

Stałe

- Obiekty stałe – definiowane przy użyciu modyfikatora `const`

  - posiadają określony typ,
  - posiadają adres,
  - tak samo jak zmienne mają swój zasięg, co oznacza, że mogą być *ukrywane* (np. ukrycie stałej wewnątrz funkcji zapewni, że jej nazwa nie będzie miała wpływu na pozostałą część programu)

Przykład:

```
const int i = 10 ;
const float pi = 3,14159 ;
const char* txt = "akuku" ;

const short zx ; // błąd – brak zainicjowania
extern const int bufsize ; // prawidłowo
                           // jawna deklaracja stałej
```

Stałe

Przykład:

```
#include <iostream>
using namespace std;
const char* stala = "--==***==--";
void f()
{
  const char stala ='A';
  cout << stala << endl;
}
int main()
{
  cout << stala << endl;
  f();
  {
    const double stala = 3.14159;
    cout << stala << endl;
  }
  cout << stala << endl;
  return 0;
}
```

Zmienne

- Zmienna symboliczna - dana adresowana - *obiekt*
  - jest związana z określonym typem danych, wyznaczającym:
    - ***rozmiar i położenie*** danych w odpowiedniej pamięci,
    - ***zakres wartości***, które mogą być przechowywane,
    - ***zbiór operacji***, które można wykonywać na tych danych,

  - pozwala za pomocą nazwy odwoływać się do zawartości odpowiednich obszarów pamięci komputerowej

Przykład:

```
int i, j, k ;

char znak ;

double dzielna, dzielnik, iloraz ;

bool tak, nie ;
```

Zmienne

- p-wartość ($r-value$) – wartość znajdująca się w przydzielonym obszarze pamięci – *wartość pobrana* .
- l-wartość ($l-value$) – *lokalizacja wartości* – adres obszaru pamięci przydzielonego obiektowi .

Przykład:

```
int zi ;
zi = zi + 5 ;
```

- po prawej stronie operatora przypisania zarówno zmienna `zi` jak również literał 5 występują jako p-wartości – wartości do pobrania

- po lewej stronie operatora zmienna `zi` występuje jako l□wartość, następuje zastąpienie dotychczasowej wartości znajdującej pod adresem przydzielonym zmiennej `zi` wartością nową – wynikiem dodawania.

☞ literał może tylko występować jako p-wartość .

Zmienne

- Definicja obiektu (zmiennej)

(np.: `double dzielna;`)

- przekazanie kompilatorowi informacji o typie i identyfikatorze obiektu ,
- przydzielenie obiektowi odpowiedniego obszaru pamięci.

- Deklaracja obiektu (zmiennej)

(np.: `extern double dzielnik;`)

- przekazanie kompilatorowi informacji i typie i identyfikatorze obiektu, **bez przydzielenia pamięci obiektowi** .

Zmienne

- Definicja jest jednocześnie deklaracją obiektu .

Przykład:

```
int i, j = 5,  k(-2) ,  q = j + k ;
char c,  znak = 'A',  znak_spec = '\n';
bool jest,  stan0 = false ,  stan1(true) ;
double dzielna,  dzielnik = 2.71,  mnoznik(1e-6);
```

Modyfikatory

- `const` – ustawienie obiektu jako niemodyfikowalnego, tylko do odczytu – używanego jedynie jako p-value

  (np.: `const int i = 10;`)

- `static` – ustawienie obiektu tak, by pozostawał w pamięci przez cały czas wykonywania programu

  (np.: `static int licznik_obiektów;`)

- `register` – szybki dostęp do obiektu – informacja dla kompilatora, by obiekt umieścić w rejestrze procesora

  (np.: `register long licznik;`)

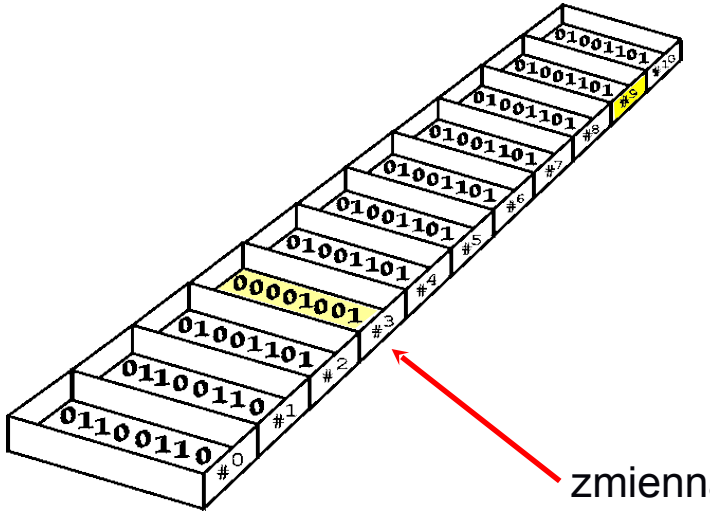
Modyfikatory

- `volatile` – obiekt ulotny – informacja dla kompilatora, że nie wiadomo kiedy może zmienić się wartość takiego obiektu; może to być obiekt przechowujący wartość rejestru urządzenia komunikacyjnego

  (np.: `volatile long int zegar;`)

Typy złożone

- Typ wskaźnikowy
  - przechowywanie adresu obiektu w pamięci (wartością wskaźnika jest adres innego obiektu),
  - typ wskaźnika określa sposób interpretowania obszaru pamięci pod konkretnym adresem,
  - dostęp do danych (wartości wskazywanego obiektu) jest uzyskiwany za pomocą operatora wyłuskania (przez *dereferencję*)



wartością wskaźnika jest adres innego obiektu

Przykłady

```
int     *wi, *wj, wk ;        // wk nie jest wskaźnikiem
double *w_iloraz ;
string *w_txt ;
vector<int> *w_wekt ;

int        *wi1 = 0;          // nie wskazuje na żaden obiekt
double       pi = 3.14 ;
double     *wpi = &pi  ;      // & - operator pobrania adresu

double  **wwpi = &wpi ;
int        i, *wi2 = &i;

wi1 = i  ; // błąd
wi1 = wpi; // błąd
wi1 = &pi; // błąd
```

Typy złożone

- Wskaźnik typu `void*`
  - związana z nim wartość jest adresem,
  - można mu przypisać adresy obiektów dowolnego typu (za wyjątkiem wskaźnika do funkcji),
  - można, przy jego pomocy, porównywać adresy różnych obiektów,
  - **nie można** działać na obiektach wskazywanych przez wskaźnik `void*`.

Przykład:

```
int *wi = &i ; double *wd = &d ; void *wv;

wv = wi;

wv = wd;
```

Typy złożone

Wskaźnik może wskazywać na obiekt stały.

```
const double   mnoznik = 2.71 ;
const double  *wstd = 0;    // wskaźnik do obiektu stałego
wstd = &mnoznik ;
```

jeśli nie jest stałą to :

```
double   pi = 3.14 ;
wstd = &pi;                 // wskazuje teraz na zmienną pi
```

nie można jednak wykonać :

```
*wstd = 5.24 ;              // błąd
                            // * - operator wyłuskania
```

Typy złożone

Wskaźnik może też być stały

(wskazywać na obiekt, który może, ale nie musi być stały).

```
int i1 = 0;  int i2 = 5;
int *const w_i1  = &i1 ;     // stały wskaźnik
w_i1 = &i2 ;                 // błąd
```

Wskaźnik stały do obiektu stałego

```
const int i3 = 8;
const int* const wcsti = &i3;
wcsti = &i2 ;                  // błąd
*wcsti = 2 ;                   // błąd
```

Typy złożone

- Typ referencyjny
  - umożliwia pośrednie odwoływanie się do obiektu za pomocą innego identyfikatora
  - obiekt referencyjny (referencja, alias) podobnie jak wskaźnik przechowuje adres innego obiektu, do którego się odnosi, z tą jednak różnicą, że wskaźnik jest zmienną a referencja nie.

Każdemu obiektowi referencyjnemu trzeba nadać wartość początkową:

```
int  z1 = 128, z2 = 64 , z3 = 5;
int  &ref_z1 = z1 ;    // od teraz ref_z1 odnosi się do z1
int  &ref ;            // błąd
int  &ref_z2 = &z2;    // błąd – ref_z2 nie jest typu int* lecz int

     ref_z1 = &z2;     // błąd
```

Typy złożone

Wszystkie operacje na wykonywane na referencji działają na obiekcie, do którego referencja się odnosi:

```
ref_z1 = 256;          // od teraz z1 == 256
ref_z1 += 4 ;          // od teraz z1 == 260
z2 = ref_z1 ;          // od teraz z2 == z1
int *wi = &ref_z1;     // wi zainicjowano adresem z1
int *&ref_wi = wi;     // referencja do wskaźnika do obiektu typu int
const int &cref_z2 = z2;  //ref. do obiektu int z modyfikatorem const
cref_z2 = 64;             // błąd
int &const ref_z3 = z3;   // błąd ref. sama w sobie jest stałą
```