Wydział Elektroniki
i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Migracja do C++ pierwsze kroki

przygotowanie programu,
struktura programu,
zmienne, funkcje

Politechnika
Warszawska

Python – prosty przykład

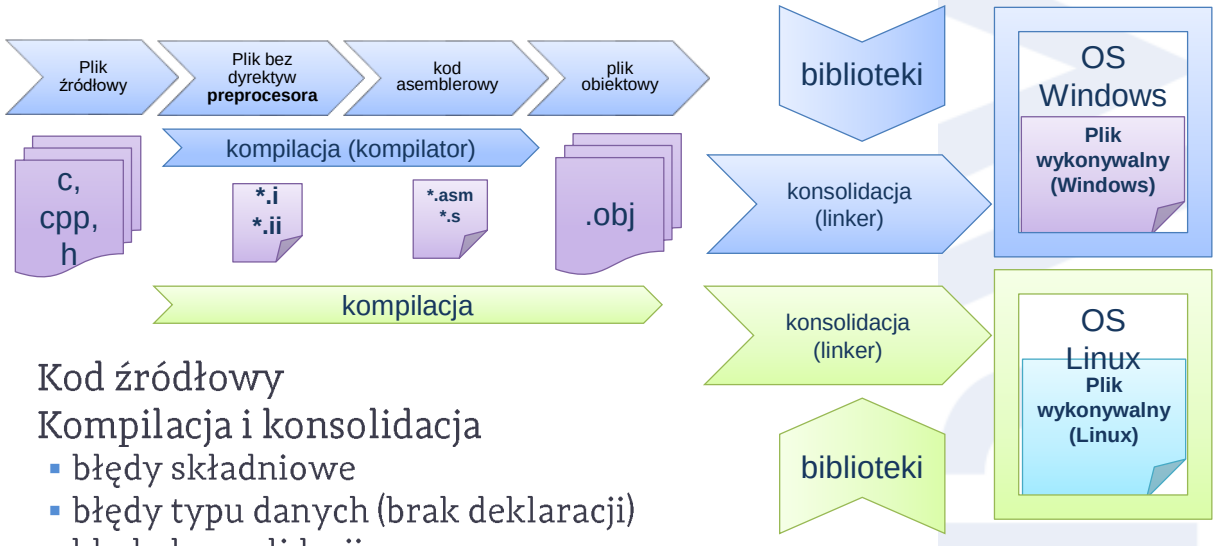
wiem, że nieoptymalny ☹

```
print([x**2 for x in range(10) if x % 2])
```

```
a = [x**2 for x in range(10) if x % 2]
print(a)
```

- Jakie operacje należy wykonać?
- W jakiej kolejności?
- Ile zmiennych będzie potrzebnych?
- Jakie operatory zostały wykorzystane?
- Jakie funkcje były użyte?

Budowanie programu



Kod źródłowy
Kompilacja i konsolidacja

- błędy składniowe
- błędy typu danych (brak deklaracji)
- błędy konsolidacji


```
#include <vector>
#include <iostream>
using namespace std;
int main() {
  vector<int> odd_squares;
  for (int ii = 0; ii < 10; ++ii)
  { // zbędne ale bezpieczniejsze
    if (ii % 2) {
      odd_squares.push_back(ii * ii);
    }
  }
 // wyświetlanie
  for (int ii = 0; ii < odd_squares.size(); ++ii) // Czy obiektowe?
     cout << odd_squares[ii] << ' ';
     cout << endl;
  return 0;
}
```

Tego będę używał

Funkcja main() jest obowiązkowa

Jeśli chcę użyć zmiennej (lub funkcji) muszę ją **zadeklarować**

Wcięcia nic nie znaczą!

Refaktoryzacja
Wprowadzanie zmian w kodzie programu, bez zmiany jego funkcjonalności

Nasz prosty przykład

C++ vs Python

	Python	C++
przygotowanie programu (szybkość działania)	maszyna wirtualna *bytecode*	komputer docelowy kod natywny
struktura programu	tabulatory	białe znaki
typowanie	dynamiczne – w trakcie wykonania	statyczne – w trakcie kompilacij
zarządzanie pamięcią	automatyczne odśmiecanie	bezpośrednie

Preprocesor

1. Zastępowanie tekstu
2. Wybiera fragmenty do kompilacji

#include
dyrektywa włącza tekst innego pliku źródłowego,

#define
definiuje stałą i makroinstrukcję (pseudofunkcję)

#undef
usuwa definicje stałej lub makra

#ifdef, ..., #ifndef, #if, #elif, #else, #endif

#pragma
np. ustawia parametry dla kompilatora

Uwaga:
dyrektywy przetwarzane są w sposób sekwencyjny, a nie rekurencyjny

```
/**
* Name: PrepUtils.h
* Purpose: Preprocesor utilities
*
* @author KGr
* @version 0.9 5/01/2015
*/
#define DOTPERINCH 300
```

Stałe w stylu C

```
#ifdef  _MSC_VER
 // ("**  _MSC_VER ** ")
#endif

#if 0
  // to nie będzie kompilowane
#endif

#if __GNUG__
   // ("** __GNUG__ ** ")
#endif

#if defined(WIN32) || defined(_WIN32)
   // ("** WIN32 ** ")
#endif

#if defined(_DEBUG) || !defined(NDEBUG)
 #pragma message ("DEBUG MODE")
#else
    #undef DISPLAY_FILE_NAME
#endif
```

Pierwszy program

```
// Pierwszy program

#include <iostream>

int main()
{
    std::cout << "Hello world!\n";
    return 0;
}
```

- zaczyna się od litery
- składa się z liter, cyfr i podkreślenia
- nie jest słowem kluczowym
- małe i wielkie litery są rozróżniane!

Komentarze
`// Pierwszy program`
`// C++, C99`
`styl C */`

Słowa kluczowe
`int, double, for, return`

Identyfikatory (nazwy)
`std, cout, x, nazwa_zmiennej`

Pojedynczy znak
Np. `'\r', '\n', '\"'`

Literały
`"Hello", 'A' , 2.4`

Operatory
`<<, +, &&, %`

Separatory
`{}, ;`

Białe znaki i komentarze są ignorowane przez kompilator

Politechnika Warszawska

Analiza programu (1)


```
// Pierwszy program
#include <iostream>
using namespace std;
int main()
{
   cout << "Hello world!\n";
   cout << 345 << endl;
   return 0;
}
```

Hello world!
345

Komendy preprocesora
`#include`
`#ifdef`

Przestrzenie nazw

Funkcja `main()`
`{`
`  blok funkcji`
`}`

Instrukcje

- Program to zbiór funkcji
- Funkcja to sekwencja instrukcji
- Instrukcje – wykonują jakąś operacje, budują bloki programu (instrukcje złożone)

Politechnika Warszawska

Analiza programu (2)


```
// Wyrażenia
#include <iostream>
using namespace std;
int main()
{
  cout << "Oblicz: 3+5*2 = ";
  cout << 3+5*2 << endl;
  cout << "Oblicz: 4+5/2 = " << 4 + 5/2 << endl;
  return 0;
}
```

Oblicz: 3+5*2 = 13
Oblicz: 4+5/2 = 6

operatory
<<, +, &&, %

Wyrażenie

Wyrażenie „posiada” wartość określonego typu

Działanie operatora zależy od **typu** obiektu

Operatory
- arytmetyczne np. *, / , %, bitowe: |, &, ^
- logiczne np. !, &&, ||, (także and, or)
- porównania np. ==, !=, <

Politechnika Warszawska

Zmienne

```
// typy wbudowane
#include <iostream>
#define PRINT(x) cout << #x" = " << (x) << endl
using namespace std;
int main()
{
  char bigA = 'A';
  double a = 2.4;
  char letter = 'd';
  cout << "bigA = " << bigA << endl;
  cout << "3 * 2/3 = " << 3 * 2 / 3 << endl;
  PRINT(3 * 2/3);
  PRINT(3 * (2/3));

  int i = letter;
  bigA = bigA + 1;
  PRINT(letter);
  PRINT(i);
  PRINT(bigA);
  return 0;
}
```

Tylko w celu skrócenia zapisu !!!

Deklaracja definiująca (**definicja**) z inicjalizacją

```
bigA = A
3 * 2/3 = 2
3 * 2/3 = 2
3 * (2/3) = 0
letter = d
i = 100
bigA = B
```

Automatyczna konwersja typu

	53F982		33	
	53F983		3	
a	53F984		51	2.4
	53F985		51	
	53F986		51	
	53F987		51	
	53F988		51	
	53F989		51	
	53F98A		3	
	53F98B		64	
	53F98C		39	
	53F98D		27	
letter	53F98E		100	d
bigA	53F98F		65	A
	53F990		46	
	…			
	2DE4F			
main()	2DE50		4	
	2DE51		34	mov ax, data
	…		22	mov ds, ax

Typ – opisuje w jaki sposób interpretowane będą dane

Zmienne/obiekty – nazwane miejsca w pamięci służące do przechowywania danych określonego typu

warning C4244: '=': conversion from 'double' to 'char', possible loss of data

Funkcje - deklaracja

- Funkcje (podprogramy) wydzielają fragment programu - mogą być wykorzystywany wielokrotnie!
- Deklaracja (prototyp) funkcji musi poprzedzać jej wywołanie

Typ zwracanej wartości

```
int salary(int hours);
int main()
{
  int  hours = 8;
  cout <<  "Dniowka = " << salary(hours) << endl;
  return 0;
}
```

Lista parametrów funkcji

Wywołanie funkcji

Error LNK2019 unresolved external symbol "int __cdecl salary(int)". …

Funkcje - definicja

- Definicja zawiera jeszcze ciało funkcji (jest także jej deklaracją)
- Definicje mogą być umieszczane w innych plikach (`*.cpp`, `*.c`), a deklaracje w plikach nagłówkowych (`*.`, `*.h`)
- Zmiana definicji nie wymaga kompilacji plików ją wykorzystujących

```
int salary(int hours)
{
   return 200 * hours;
};
```

nagłówek funkcji

ciało funkcji

Wprowadzanie danych


```
// pominięte dyrektywy np. #include <iostream>
#include <string>

void main() {
  cout << "Podaj imie: ";
  string imie;
  cin >> imie ;   // wprowadź imie
  cout << "Podaj wiek: ";
  int wiek;
  cin >> wiek;   // wprowadź wiek

  cout << "Witaj " + imie + " masz lat " << wiek << endl;

  cout << "Witaj " << imie << " masz lat " << wiek << endl;
}
```

Użyjemy klasy string

Jaka różnica ?

Operatory - zestawienie

Podstawowe operatory				
Przypisania	**Inkrementacja/ dekrementacja**	**arytmetyczne**	**logiczne**	**porównania**
a = b a += b a -= b a *= b a /= b a %= b a &= b a |= b a ^= b a <<= b a >>= b	++a --a a++ a--	+a -a a + b a - b a * b a / b a % b ~a a & b a | b a ^ b a << b a >> b	!a a && b a || b	a == b a != b a < b a > b a <= b a >= b

alteratywnie:
or, not_eq, and, ...

Uwaga - operatorów jest więcej !

Projektowanie prostych klas

Lista pytań

- Jakie są istotne cechy (atrybuty, stan wewnętrzny) obiektów i jak je reprezentować?
- Czy obowiązują jakieś *niezmienniki* stanu wewnętrznego obiektów?
- Które atrybuty można udostępniać publicznie a które powinny być kontrolowane?
- Jak będą tworzone i inicjowane obiekty; czy dopuszczamy istnienie obiektów z nieokreślonym stanem wewnętrznym?
- Czy likwidacja obiektu wymaga czynności porządkowych?
- Jakie operacje będą wykonywane na obiektach?
  Które mają być prywatne, które publiczne?
- Jakie algorytmy zastosować w operacjach?
- Jak program ma korzystać z definicji klasy?

Klasa Fraction

Decyzje projektowe:

- Atrybuty: `long l,m;` (licznik, mianownik)
- Niezmiennik: l,m względnie pierwsze (nieskracalne); m>=0
- Dziedzina: liczby wymierne w zakresie wynikającym z reprezentacji typu long z dodatkową konwencją:
  $1/0 \Rightarrow +\infty$; $-1/0 \Rightarrow -\infty$; $0/0 \Rightarrow$ wartość nieokreślona
- Inicjowanie obiektów (4 sposoby):
  (1) bez argumentów (inicjowanie domyślne);
  (2) przez podanie wartości całkowitej $\Rightarrow$ mianownik = 1;
  (3) przez podanie pary wartości całkowitych l, m;
  (4) przez podanie innego obiektu `Fraction`.
- Usuwanie obiektu - bez żadnych czynności porządkowych

Klasa Fraction (cd)


```
#include<iostream>
using namespace std;
class Fraction
{ long l,m;
  long nwp(long p, long q) const;
  void initFraction(long a, long b);
public:
  Fraction(long ll = 0, long mm = 1L)
  { initFraction(ll, mm); }
  Fraction operator+(const Fraction &f) const;
  /*. . . */
  Fraction operator/(const Fraction &f) const;
  friend ostream& operator<< (ostream &os,const Fraction &f);
  friend istream& operator>> (istream &is, Fraction &f);
}; //Fraction;
```


Klasa Fraction: przykładowa metoda


```
Fraction Fraction::operator+( const Fraction &f) const
{ long tl, tm;
  if(m==0 && f.m==0)
  {  tl =(l+f.l)/2;
     tm = 0;
  }
  else
  { tl =l*f.m + m*f.l;
    tm = m*f.m;
  }
  return Fraction(tl, tm);
}
```

Uwagi o stylu kodowania – plik .cpp

- Komentarz ogólny z identyfikacją autora
- dołączenie standardowych plików nagłówkowych **`#include <iostream>`**
- dołączenia własnych plików nagłówkowych **`#include "Stos.h"`**
- definicje typów, stałych, zmiennych globalnych
- deklaracje (prototypy) / definicje funkcji

```
/**
* Name: stos.cpp
* Purpose:  Stos w konwencjach C++
* @author KGr
**/
#include <iostream>
#include <string>

using namespace std; // Unikać !

static void error(string msg)  // C++17 std::string_view
{ cout << msg << endl;  // std::cout << msg;
  exit(1);
}

class Stack {
 // private (default)
    static constexpr int SIZE = 100;
    int S[SIZE] = {}, sp{0}; // std::array<int, SIZE> S{};
public:
    Stack() = default;
    void push(int e) {
        if (!full()) S[sp++] = e;
        else error("Stack overflow");
    }
    int pop() {
        if (!empty()) return S[--sp];
        else error("Stack underflow");
    }
    int empty() const { return sp == 0; }
    int full()  const { return sp == SIZE; }
}; // Wszystkie funkcje inline
// Wykorzystanie stosu
// ===================
int main()
{
    Stack A, B;// Konstruktor Stack() wywołany automatycznie
    B.push(10);
    A.push(B.pop());
    return 0;
}
```

komentarz

pliki nagłówkowe

Funkcje/zmienne globalne

klasa to jest nazwa typu

Klasa powinna znaleźć się w innym pliku

Politechnika Warszawska

Uwagi o stylu kodowania (cd1)

Nazewnictwo ułatwiające zrozumienie kodu

Komentarze - krótkie i pomocne dla czytającego

- Komentarze ogólne dla ważnych, nietrywialnych funkcji (dla funkcji prostych nazwa powinna być wystarczająco sugestywna)
- Komentarze lokalne w miejscach ważnych
- Komentować, nawet szkicowo, podczas pisania kodu (nie ex post)

Uwzględniać strukturę hierarchiczną kodu

- stosować konsekwentnie wyróżnienie zapisu dla tekstu „podległego” (unikać tabulatorów; wcięcie = 2 spacje):
  - wsunięcie dla zawartości bloku
  - wsunięcie dla wnętrza instrukcji strukturalnych
  - wsunięcie dla zawartości klasy

Politechnika Warszawska

Uwagi o stylu kodowania (cd2)

Konsekwentne stosowanie konwencji nazw np.:

- Makrodefinicje (stałe) **UPPERCASE**
- Nazwy typów **PascalCase**
- Nazwy metod **camelCase**
  - prefiks dla metod dostępu do właściwości **get**, **set**
- Nazwy zmiennych **camelCase**
  - prefiks zmiennych numeryczne **n** (*number of* ...)
  - nazwy zwyczajowe dla int: **i**, **j**, **k**, **n**
  - długość nazwy proporcjonalna do jej zasięgu ☺
  - zmienne bool traktowane jako odpowiedź np. `isEmpty`
- zmienne prywatne można zakończyć podkreśleniem (_)
  (nie używać znaków specjalnych _, $ na początku nazwy)
- Nazwy przestrzeni nazw **lowercase**

Uwagi o stylu kodowania (cd3)

Możliwie najmniej dowolności tam, gdzie jest ona zbędna lub szkodliwa

Przestrzegać zasady umiejscawiania deklaracji lub definicji składników kodu źródłowego w standardowych miejscach

Przeczytać to, co się napisało także po uznaniu, że program "działa" (programowanie nie jest twórczością "write-only")

Jeżeli nie widzisz błędu tam gdzie patrzysz, to znaczy, że patrzysz w złe miejsce