WYDZIAŁ FIZYKI
UNIWERSYTET WARSZAWSKI

Programowanie zaawansowane FM i NI

Wykład 14

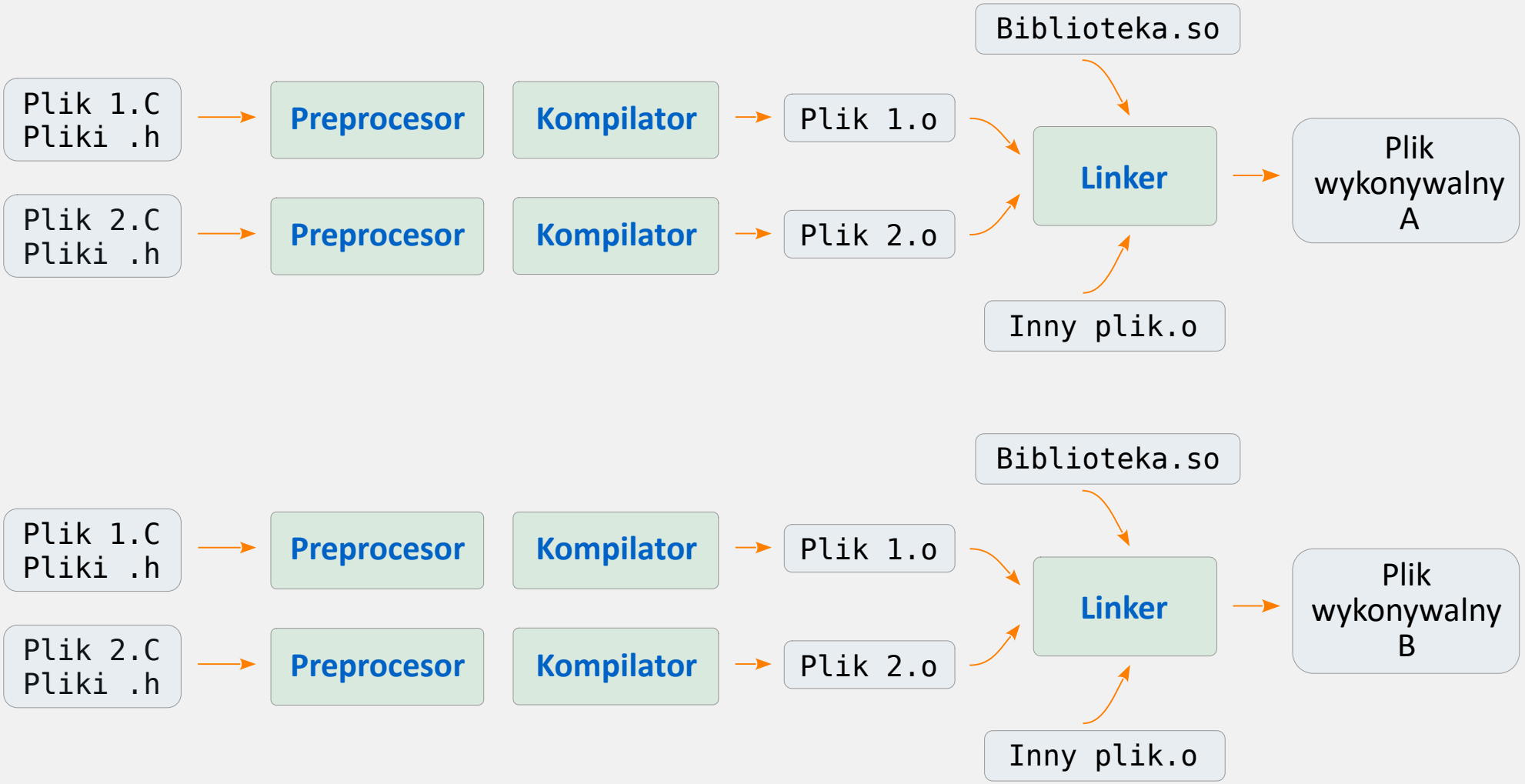
make, gnuplot_i

Krzysztof Piasecki

Semestr letni roku akad. 2023-24

krzysztof.piasecki@fuw.edu.pl

- **Schemat kompilowania aplikacji**



⇒ Kompilując każdorazowo większy kod, trzeba się sporo napisać... Czy tego nie można zautomatyzować?

`make` : program do automatyzacji kompilacji

[*wstęp 1*] [*wstęp 2*]

⊙ Program ten czyta plik sterujący **`Makefile`** , w którym zawieramy „**reguły**” kompilacji.
Każda **reguła** to zasada wykonania porcji zadania (np.: kompilacja kodu z kilku plików `.C` i `.h` do pliku `.o`)

⊙ Składnia prostej reguły wygląda tak:

```
Cel_Reguły: wymagane składniki
{tabulator}  Komenda shell'a (np. kompilacja)
```

Przykładowa reguła w środku pliku `Makefile` :

```
Notes: Notes.C Notes.h main_notes.C
    g++ Notes.C main_notes.C -o Notes.exe
```

⊙ Wystarczy teraz wpisać polecenie: **`make`**
Program wczyta `Makefile` i zajmie się powyższą regułą:
① sprawdzi, czy na obecnej ścieżce są wymagane składniki: pliki `Notes.C`, `Notes.h` i `main_notes.C`
② Jeśli tak, to zawoła `g++` , który zbuduje aplikację `Notes.exe` , kompilując `Notes.C` i `main_notes.C` .

⊙ ***Garść uwag*** :
▻ Plik sterujący możemy nazwać inaczej. Np. dla pliku `MyMakefile2` , piszemy: **`make -f MyMakeFile2`**
▻ Linijka z komendą do kompilacji *musi* zaczynać się od **tabulatora**.
▻ W regule linii z komendami może być więcej, ale są one niezależne od siebie.
▻ Ostatnia komenda pliku sterującego *musi* zawierać [Enter] .
▻ # na początku linijki oznacza komentarz .

- W `makefile` możemy zapisać więcej reguł.
  Spróbujmy zapisać reguły kompilacji kodu z ćwiczeń, dotyczącego notesu osób
  i podzielonego na pliki nagłówkowe, implementacyjne i kod klienta:

```
All: Osoba.o Notes.o
    g++ notes_klient.C Osoba.o Notes.o -o Notes.exe

Osoba.o: Osoba.h Osoba.C
    g++ -c Osoba.C

Notes.o: Notes.C Notes.h Osoba.h
    g++ -c Notes.C
```

  - `make` podejmie pierwszą regułę (`All`) i zorientuje się, że plików `.o` na dysku nie ma.
    - Poszuka, czy nie zapisaliśmy reguł z **celami**, jakimi są: `Osoba.o` i `Notes.o`
      - Są takie! Wstrzyma więc regułę `All` i wpierw wykona reguły `Osoba.o` i `Notes.o`.
  - Wówczas powróci do reguły `All`. Wymagane składniki już są, więc wywoła `g++` do kompilacji.

- ***Garść uwag*** :
  - `make` potrafi działać domyślne.
    Gdyby skasować reguły kompilacji plików `.o`, to domyślnie `make` i tak je skompiluje.
  - Możemy wskazać, od której reguły `make` ma zacząć. Np.: `make Osoba.o`
  - Bez wskazania reguły, `make` rozpatrzy tylko pierwszą (i ew. reguły dla wymaganych składników)
  - Pisząc `make -n`, możemy dla testu zobaczyć komendy do wykonania, ale bez wykonywania.

make: zmienne

⊙ make rozpoznaje zmienne środowiskowe. Np. unixowe powłoki znają zmienną PATH.
Można jej użyć w linii komendy w makefile, np.: echo $(PATH)

⊙ Ponadto, make zna kilka zmiennych domyślnych, np.:

CC	=	nazwa kompilatora języka C
CXX	=	nazwa kompilatora języka C++
CFLAGS	=	lista opcji kompilacji dla kompilatora C
CXXFLAGS	=	lista opcji kompilacji dla kompilatora C++
LDFLAGS	=	lista opcji dla linkera.

⊙ My też możemy zdefiniować nowe zmienne. Np. definiujemy: OBJS = Osoba.o Notes.o
co możemy później użyć tak: $(OBJS)

⇒ Nasz makefile może teraz wyglądać tak:

```
OBJS = Osoba.o Notes.o
CFLAGS = -c

All: $(OBJS)
    $(CXX) notes_klient.C $(OBJS) -o notes.exe

Osoba.o: Osoba.h Osoba.C
    $(CXX) $(CFLAGS) Osoba.C

Notes.o: Notes.C Notes.h Osoba.h
    $(CXX) $(CFLAGS) Notes.C
```

make : zmienne automatyczne

⊙ W linii komendy danej reguły możemy posłużyć się paroma sprytnymi skrótami:

$@	to zamiennik na cel reguły	(w nagłówku reguły, to przed :)
$^	to zamiennik na listę wymaganych składników	(w nagłówku reguły, to po :)
$<	to zamiennik na pierwszy z listy wymaganych składników	(w nagłówku reguły, to po :)

⇒ Teraz nasz plik `makefile` może wyglądać tak:

```
OBJS = Osoba.o Notes.o
CFLAGS = -c

All: $(OBJS)
    $(CXX) notes_klient.C $^ -o notes.exe

Osoba.o: Osoba.h Osoba.C
    $(CXX) $(CFLAGS) Osoba.C

Notes.o: Notes.C Notes.h Osoba.h
    $(CXX) $(CFLAGS) $<

clean:
    rm Osoba.o Notes.o notes.exe
```

- Wprowadziliśmy też regułę **`clean`** . Użytkownik może teraz napisać: `make clean` i efekty poprzednich działań `make` zostaną cofnięte. To powszechnie praktykowana reguła.

`gnuplot_i.hpp`: interfejs do gnuplot'a

Na wielu systemach zainstalowany jest **`gnuplot`** – program do rysowania z linijkowym interfejsem.
Dla C++ dostępna jest biblioteka **`gnuplot_i.hpp`**.
Pozwala ona, aby z poziomu kodu C++ wykonywać komendy `gnuplot'a`.

⊙ Do tego trzeba ściągnąć plik nagłówkowy, np. [*stgd*]
i w każdym kodzie go załączać przez `#include`.

[Link]

Na początek tworzymy obiekt klasy **`Gnuplot`**.
Otwiera on okno graficzne i nim zawiaduje.
Działania – to metody na tym obiekcie.

⊙ Aby **narysować punkty**,
① tworzymy `vector<double>` ze wsp. X i Y
② wykonujemy metodę `plot_xy`.

⊙ **Ustawienia** w tym przykładzie:

`set_xrange (min,max)`	ustawia zakres osi X
`set_yrange (min,max)`	ustawia zakres osi Y
`set_xlabel ("Os X")`	podpis osi X
`set_ylabel ("Os Y")`	podpis osi Y
`set_pointsize (1)`	rozmiar punktów
`set_style ("points")`	rysuje osobne punkty

Style to np.: `points`, `lines`, `linespoints`, `dots`

```
 6   vector<double> X1 = { 1   , 2   , 3   },
 7                  Y1 = { 0.8, 1.9, 3.2 };
 8   Gnuplot G ("My window");
 9   G.set_xrange (0, 4).set_yrange (0, 4);
10   G.set_xlabel ("X") .set_ylabel ("Y") ;
11   G.set_pointsize (3);
12   G.set_style ("points");
13   G.plot_xy (X1, Y1, "Punkty" ) ;
14
15   vector<double> X2 = { 1.0, 3.0 },
16                  Y2 = { 1.0, 3.0 };
17   G.set_style ("lines");
18   G.plot_xy (X2, Y2, "Linia") ;
```

`gnuplot_i.hpp`: rysowanie c.d.

W tym przykładzie rysujemy te same dane, ale inaczej:

[Link]

- Metodą `savetofigure` przekierowujemy wyświetlanie z okna do pliku graficznego.
  Tu: plik w formacie png.
  Gnuplot umożliwia *więcej formatów*.

- Nowe ustawienia stylu:

`set_xlogscale ()`	skala log na osi X
`set_ylogscale ()`	skala log na osi Y
`set_grid ()`	włącza siatkę

```
 6   vector<double> X1 = { 1  , 2  , 3   },
 7                  Y1 = { 0.8, 1.9, 3.2 };
 8   Gnuplot G ("My window");
 9   G.savetofigure ("myfig.png", "png");
10
11   G.set_xrange (0, 4).set_yrange (0, 4);
12   G.set_xlabel ("X") .set_ylabel ("Y") ;
13   G.set_pointsize (3);
14   G.set_style ("points");
15
16   G.set_ylogscale ();
17   G.set_yrange (0.5, 4.0);
18   G.set_grid ();
19   G.plot_xy (X1, Y1, "Punkty");
20
21   cout << "Press [Enter] to finish.";
22   cin.ignore();
```

⊙ Przy okazji, komenda `cin.ignore()` jest jednym ze sposobów **pauzy** w kodzie.

`gnuplot_i.hpp`: punkty doświadczalne (z niepewnościami)

⊙ W tym przykładzie pokazujemy, jak wykreślić **punkty danych z niepewnościami**:

① tworzymy 3 obiekty `vector<double>`, na współrzędne `X` i `Y` oraz `ΔY`,

② wykonujemy metodę `plot_xy_err`.

[Link]

```
6   vector<double> X = { 1 , 2   , 3   },
7                  Y = { 1 , 3.2, 2.5 },
8                 dY = {0.1, 0.2, 0.05};
9   Gnuplot g1 ("My window");
10  g1.set_xrange (0, 4).set_yrange (0, 4);
11  g1.set_xlabel ("X") .set_ylabel ("Y") ;
12  g1.set_style ("linespoints");
13  g1.set_pointsize (3);
14
15  g1.plot_xy_err (X , Y , dY, "Moje dane");
```

`gnuplot_i.hpp:` wykres punktowy w 3D

Tym razem narysujemy zestaw **punktów na wykresie 3D**:

① tworzymy 3 obiekty `vector<double>` , na współrzędne `X` i `Y` oraz `Z`,

② wykonujemy metodę `plot_xyz` .

[Link]

```
6   vector<double> X = { 0.3, -0.2, -1.5 },
7                  Y = { 0.8, -0.9,  1.2 },
8                  Z = { 1   ,  -1 ,  0.5 };
9   Gnuplot G ("My window");
10  G.set_xrange (-2, 2).set_yrange (-2, 2);
11  G.set_zrange (-2, 2);
12  G.set_xlabel ("X") .set_ylabel ("Y") ;
13  G.set_zlabel ("Z") ;
14  G.set_pointsize (3);
15  G.set_style ("points");
16  G.plot_xyz (X, Y, Z, "Linia") ;
```

- **gnuplot_i.hpp: wykresy funkcji y = f(x)**

⊙ Ten przykład ukazuje **wykres funkcji w dziedzinie ℝ**. Wpisujemy formułę w metodę **plot_equation** .

[Link]

```
6   Gnuplot g1 ("My window");
7   g1.set_xrange (0, 6).set_yrange (-1, 3);
8   g1.set_xlabel ("X") .set_ylabel ("Y") ;
9   g1.set_style ("lines");
10
11  g1.plot_equation ("2*(x**2) * exp(-x)" ) ;
12  g1.plot_equation ("tan(x)");
13  g1.plot_equation ("gamma(x)");
```

⊙ Co można wpisać w treść formuły? Każdą kombinację złożoną z:

```
+ - * / %
nawiasy:  () ,   potęgowanie: a**b  ,    silnia: !
abs, sgn, floor, ceil, int
sqrt, exp, log, log10
sin, cos, tan, asin, acos, atan
sinh, cosh, tanh, asinh, acosh, atanh
erf, gamma, ...
```

- **`gnuplot_i.hpp`: wykresy funkcji z = f(x , y)**

Ten przykład demonstruje **wykres funkcji w dziedzinie $\mathbb{R}^2$**.
Wpisujemy formułę w metodę **`plot_equation3d`** .

[Link]

```
6   Gnuplot g1 ("My window");
7   g1.set_xrange (-2, 2).set_yrange (-2, 2);
8   g1.set_zrange ( 0 , 1 );
9   g1.set_xlabel ("X") .set_ylabel ("Y") ;
10  g1.set_style ("lines");
11  g1.plot_equation3d ("exp(-0.5*(x**2+y**2))");
12
13  cout << "Rotate me on the plot, "
14       << "then press [Enter] to continue";
15  cin.ignore();
16
17  g1.reset_plot ();
18  g1.set_xrange (0, 1).set_yrange (0, 1);
19  g1.plot_equation3d ("sqrt(1- x*x -y*y)") ;
```

⊙ Przy okazji, metoda `reset_plot()` czyści okno graficzne.