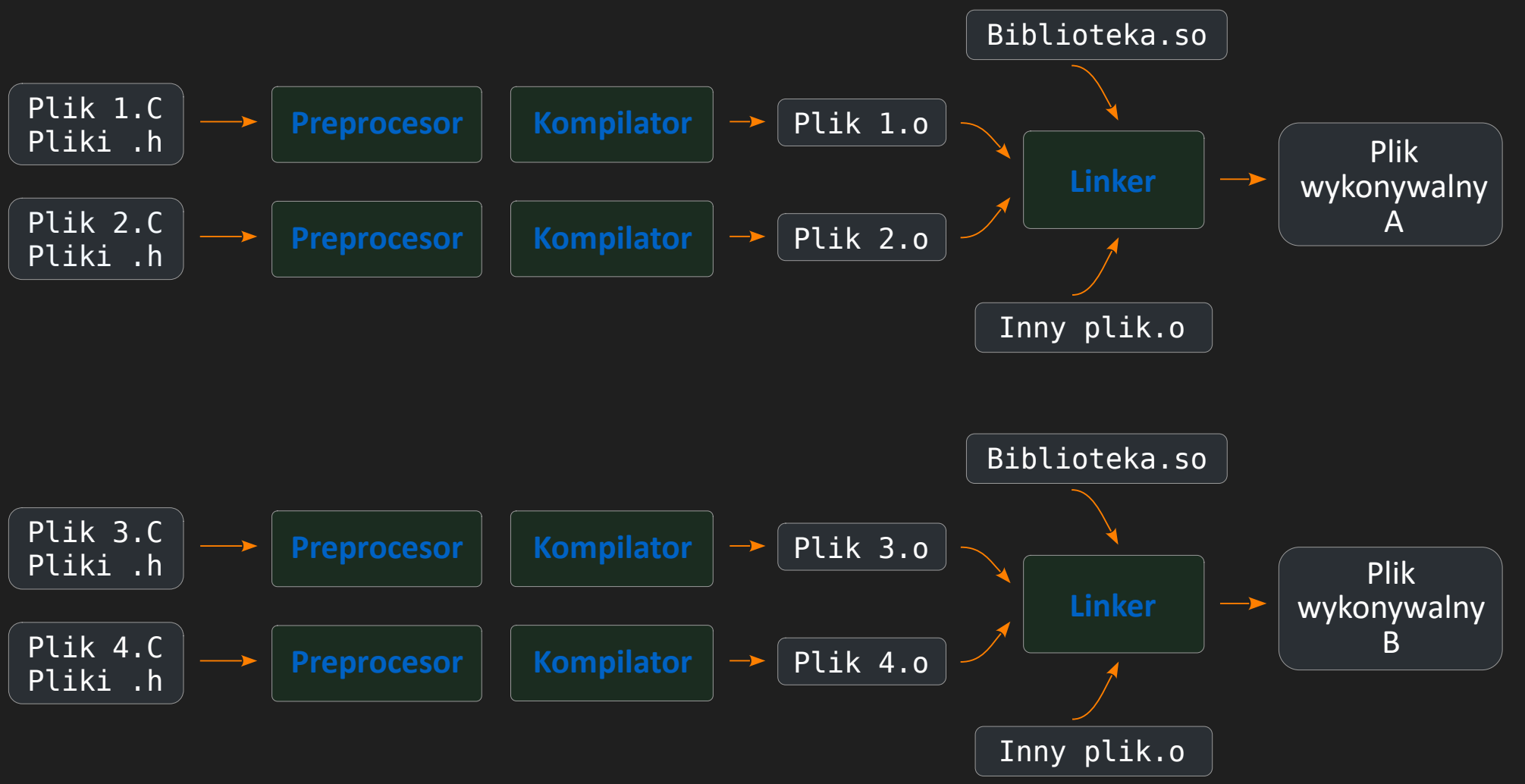
Narzędzia Wspierające Programowanie

Make – automatyzacja budowania aplikacji

- Nieco ogólniejszy schemat budowania aplikacji z kodu kompilowalnego:



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

- make : program do automatyzacji budowania aplikacji.

  Instrukcja odautorska jest [tutaj]. Przykładowe samouczki na stronach FUW: [tutaj] i [tutaj].

- Program ten czyta plik sterujący `makefile`, w którym zawieramy „reguły" budowania.
  Każda reguła – to zasada wykonania porcji zadania
  (np.: kompilacja kodu z kilku plików .C i .h do pliku .o)

- Składnia prostej reguły wygląda tak:

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

`Target` (cel reguły) – to string, stanowiący nazwę reguły.
Można wywołać `make`, wskazując dany `target` – i `make` wykona tę regułę.
W środku procedowania `makefile`, odwoływanie się do reguł jest też po nazwie (`targecie`).

- Przykładowa reguła w środku pliku `makefile`:

```
Funkcja: src/Funkcja.C includes/Funkcja.h
{tabulator}  g++ -I./includes -c src/Funkcja.C -o obj/Funkcja.o
```

- Wystarczy teraz wpisać polecenie: make
  Program wczyta `makefile` i zajmie się powyższą regułą:

  ① sprawdzi, czy na obecnej ścieżce są wymagane składniki:
  pliki `includes/Funkcja.h` i `src/Funkcja.C`

  ② Jeśli tak, to zawoła `g++`, który skompiluje `Funkcja.C` do pliku obiektowego `Funkcja.o`

Uwagi odnośnie składni:

- Linijka z komendą (np. do kompilacji) musi zaczynać się od tabulatora.

- W regule, linii z komendami może być więcej. Ale linie są niezależne od siebie:
  np. zmienna utworzona w takiej linii – nie istnieje w następnej.
  Podobnie, włączenie skryptu konfiguracyjnego – ma zasięg tylko do tej linii.
  Ale jeśli w jednej linii połączymy polecenia (A ; B) lub (A && B), to B będzie pamiętać o A.

- Ostatnia komenda pliku sterującego musi zawierać [Enter] .

- Plik sterujący możemy nazwać inaczej. Np. dla pliku `MyMakeFile2`, wywołujemy:

```
make -f MyMakeFile2
```

- `#` na początku linijki oznacza komentarz .

Ważne uwagi odnośnie reguł:

- Reguł może być więcej.
  Można nakazać `make`, aby procedował wybraną regułę. Np. dla nazwy `target1` piszemy:

```
make target1
```

- Jeżeli `make`, czytając regułę A, widzi brak składnika (w nagłówku po prawej stronie) ,
  to szuka innej reguły (B), o targecie będącym tym składnikiem.
  Reguła A jest wtedy “zawieszana”, a wykonywana jest reguła B.
  Następnie `make` powraca do reguły A.

- Przykład. Zbierzmy te uwagi i rozszerzmy nieco nasz `makefile`:

```
All: obj/Funkcja.o obj/Wymierna.o
    @echo Two classes compiled to object files in obj folder.

obj/Funkcja.o: src/Funkcja.C includes/Funkcja.h
    g++ -I./includes -c src/Funkcja.C -o obj/Funkcja.o

obj/Wymierna.o: src/Wymierna.C includes/Wymierna.h
    g++ -I./includes -c src/Wymierna.C -o obj/Wymierna.o
```

- `make` zacznie od pierwszej reguły (`All`) i zobaczy, że plików `Funkcja.o` i `Wymierna.o` nie ma na ścieżce `obj`.
  - ↳ Poszuka więc reguł o tych nazwach (są takie!)
    - ↳ Wstrzyma `All` i wykona reguły o targetach: `obj/Funkcja.o` i `obj/Wymierna.o`.
      - ↳ Powróci do reguły `All`.

- Uwagi :

- Przedrostek `@` (tu: użyty przed `echo`) – bez niego, `make` wypisze polecenie. Znak `@` to wyciszy.

- Pisząc `make -n`, możemy dla testu zobaczyć komendy do wykonania, ale bez wykonywania.

- Zmienne

- `make` rozpoznaje zmienne środowiskowe (np. PATH).

- My też możemy zdefiniować nowe zmienne. Dodajmy na początek naszego `makefile`:

```
Includes=-I./includes
CXXFlags=-std=c++17
```

Użycie zmiennej wygląda tak: `$(zmienna)`. Zmodyfikujmy więc nasze reguły w ten sposób:

```
obj/Funkcja.o: includes/Funkcja.h src/Funkcja.C
    g++ $(CXXFlags) $(Includes) -c src/Funkcja.C -o obj/Funkcja.o
```

- Często używane nazwy zmiennych:

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.

Zmienne automatyczne

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

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

  Zobaczmy, jak można skrócić regułę `obj/Funkcja.o`, używając `$@` i `$<` :

```
obj/Funkcja.o: src/Funkcja.C includes/Funkcja.h
    g++ $(CXXFlags) $(Includes) -c $< -o $@
```

- Konwencja: ładnie napisany `makefile` zawiera też regułę `clean`. Gdy użytkownik wpisze: `make clean`, reguła ta ma wyczyścić skutki po `make`. U nas reguła ta powinna wyglądać tak:

```
clean:
    rm -f obj/*.o
```

Zobaczmy cały `makefile`, kompilujący też resztę klas i budujący całość:

```
Includes=-I/usr/include -I./includes
CXXFlags=-std=c++17
GSL_Libs=$(shell gsl-config --cflags)
MYEXE=./bin/myCalcApp

myCalcApp: obj/Funkcja.o obj/Wymierna.o obj/Wielomian.o \
           obj/Gauss.o obj/IntGauss.o obj/MathTools.o
    g++ $(CXXFlags) $(Includes) main.C -o $(MYEXE) $^ $(GSL_Libs)

obj/Funkcja.o: src/Funkcja.C includes/Funkcja.h
    g++ $(CXXFlags) $(Includes) -c $< -o $@

obj/Wymierna.o: src/Wymierna.C includes/Wymierna.h
    g++ $(CXXFlags) $(Includes) -c $< -o $@

obj/Wielomian.o: src/Wielomian.C includes/Wielomian.h
    g++ $(CXXFlags) $(Includes) -c $< -o $@

obj/Gauss.o: src/Gauss.C includes/Gauss.h
    g++ $(CXXFlags) $(Includes) -c $< -o $@

obj/IntGauss.o: src/IntGauss.C includes/IntGauss.h
    g++ $(CXXFlags) $(Includes) -c $< -o $@

obj/MathTools.o: src/MathTools.C includes/MathTools.h
    g++ $(CXXFlags) $(Includes) -c $< -o $@

clean:
    rm -f obj/*.o $(MYEXE)
```

\ = przedłużenie linii

Dalsza serializacja

- Nasz `makefile` wygląda długo. I właściwie, wiele reguł jest takich samych.
  Nauczymy się to serializować. Najpierw utwórzmy zmienną z listą plików `.C` oraz drugą, odpowiadających im przewidywanych plików `.o`. Później użyjemy tych list do serializacji.

  ① Pliki .C istnieją w katalogu src . Do utworzenia listy z nimi, użyjemy funkcji `wildcard`:

  ```
  SRCS = $(wildcard src/*.C)
  ```

  Zwraca ona listę plików zgodnych ze wzorcem (elementy oddzielone są spacją).

  ② Utwórzmy teraz nową listę, na bazie SRCS. Będzie mieć podmienione wystąpienia .C na .o
  Możemy tu użyć funkcji `subst` albo `patsubst`:

  ```
  OBJS = (subst .C,.o, $(SRCS) )          albo:
  OBJS = (patsubst %.C,%.o, $(SRCS) )
  ```

  Obie funkcje czytają string SRCS i generują z niego string zmieniony.
  Funkcja `subst` dowolne wystąpienie znaków `.C` podmieni na `.o`
  Funkcja `patsubst` rozważy słowa (oddzielone spacją). Używając % jako wildcard, sprawimy, że `patsubst` każdemu słowu kończącemu się na `.C`, zamieni końcówkę na `.o`

  ③ Mamy teraz dwie listy plików: dla `.C` w zmienej SRCS i .o w zmiennej OBJS.
  Teraz zserializujemy reguły. Wpierw pod regułę główną podstawimy pliki obiektowe:

  ```
  myCalcApp: $(OBJS)
      g++ $(CXXFlags) $(Includes) main.C -o $(MYEXE) $^ $(GSL_Libs)
  ```

  Widzimy, że w komendzie `g++` grupa plików `.o` została podstawiona dzięki zmiennej `$^` .

  ④ Użyjemy teraz pattern rule (reguły dla wzorca) , która będzie wspólną receptą na wykonanie każdego pliku `.o` z plików `.C` i `.h` :

  ```
  ./obj/%.o: src/Funkcja.C includes/Funkcja.h
      g++ $(CXXFlags) $(Includes) -c $< -o $@
  ```

  Upewnijcie się, czy rozumiecie, co ta reguła dokładnie robi.
  Czyż to nie skraca ogromnie naszego `makefile`?

- Ostatecznie, nasz `makefile` nabrał takiej postaci:

```
Includes=$(shell gsl-config --cflags) -I./includes
GSL_Libs=$(shell gsl-config --libs)
CXXFlags=-std=c++17

SRCS=$(wildcard ./src/*.C)
OBJS=$(patsubst ./src/%.C, ./obj/%.o, $(SRCS))

MYEXE=./bin/myCalcApp

myCalcApp: $(OBJS)
    g++ $(CXXFlags) $(Includes) main.C -o $(MYEXE) $^ $(GSL_Libs)

./obj/%.o: src/%.C includes/%.h
    g++ $(CXXFlags) $(Includes) -c $< -o $@

clean:
    rm -f $(OBJS) $(TARGET)
```

- Tu kończymy omawianie make.
  Narzędzie `make` posiada wiele innych funkcji i reguł.
  Zainteresowanych odsyłamy do tutoriali.