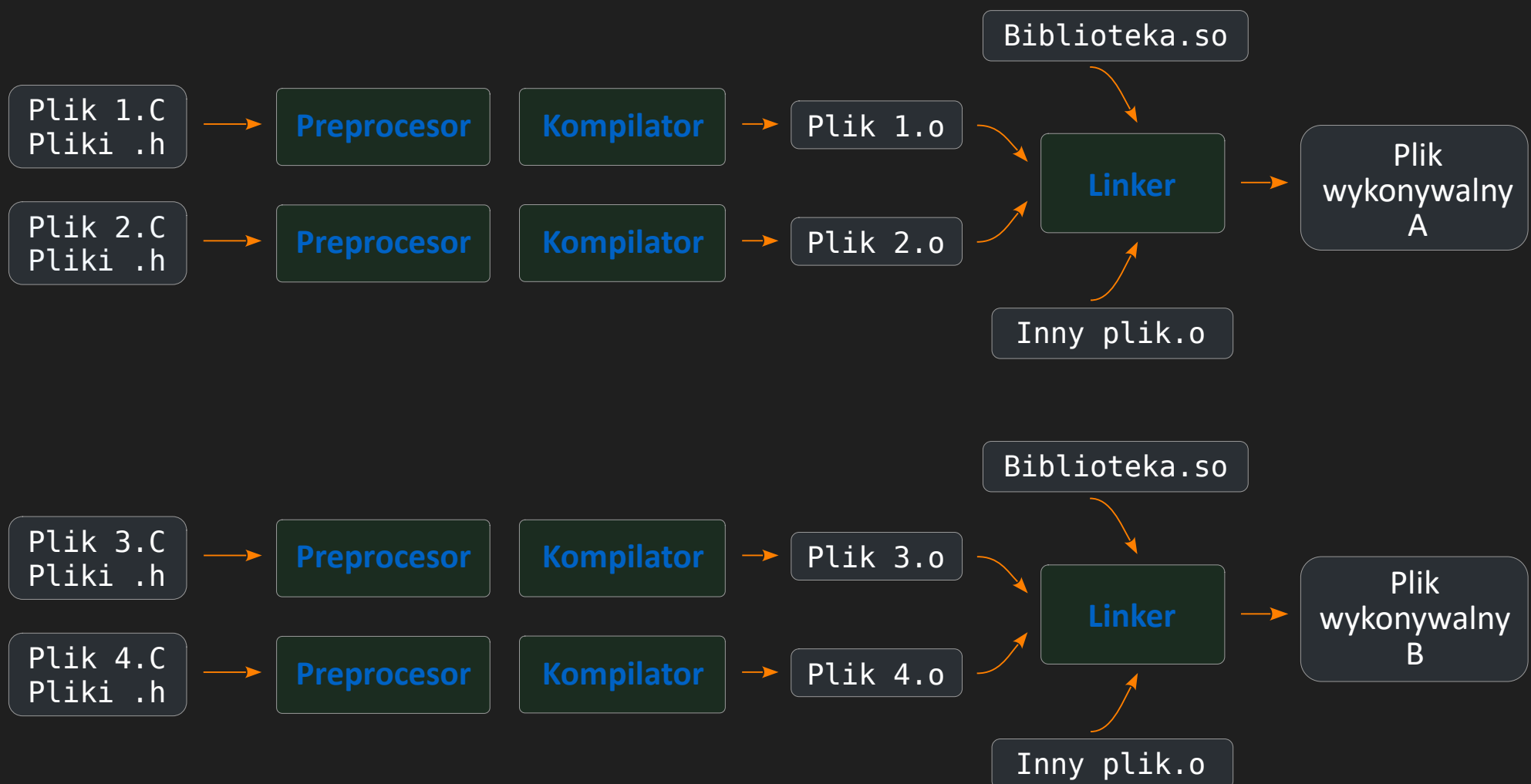


Narzędzia Wspierające Programowanie

Make – automatyzacja budowania aplikacji

🔗 Nieco ogólniejszy schemat budowania aplikacji z kodu kompilowalnego:



- 🕒 **Przykład.** Na ścieżce MathTools mamy projekt, z którego zrobimy aplikację.
- ◇ Na ścieżce głównej jest plik main.C z funkcją main.
- ◇ Każdy plik na ścieżkach src i includes poświęcony jest jednej klasie:
 - W Funkcja.C i .h jest klasa Funkcja. To klasa bazowa, po której dziedziczą inne funkcje. W jej polach kodujemy dziedzinę funkcji (od – do). Zapowiada ona metodę „f”, która ma podawać wartość funkcji dla danego x. Ta klasa i dziedziczące – mają też metodę Status, wypisującą dziedzinę i parametry.
 - Zakodowane są klasy: Wymierna, Wielomian, Gauss i IntGauss ($\int_{-\infty}^x \text{Gauss}$). Każda z nich ma swój konstruktor i metodę f, zwracającą wartość funkcji dla danego x.
 - Klasa IntGauss do działania potrzebuje zewnętrznej biblioteki matematycznej GSL.
 - Klasa MathTools – ma proste narzędzia analityczne, operujące na tych funkcjach: MinXY (punkt o najmniejszej wartości), Integral (całka) i Root (miejsce zerowe)
- ◇ Funkcja main testuje działanie tych narzędzi na kilku rodzajach funkcji.
- ◇ Mamy też puste katalogi: obj i bin. Podczas budowania, do obj wstawimy pliki obiektowe, a do bin – wykonaną aplikację. Takie rozplanowanie katalogów wprowadza porządek i jest często używane.

Narzędzia Wspierające Programowanie

Make – automatyzacja budowania aplikacji

Kompilując (i linkując) tego typu 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 klasy w plikach .C i .h → do pliku .o)
- ▶ Składnia prostej reguły wygląda tak:

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

Target (cel reguły) – to string, stanowiący nazwę reguły.

wymagane składniki – to pliki, które powinny być na dysku. Jeśli są, to make wykona regułę.
Jeśli jakiegoś składnika nie ma, to make poszuka w pliku **makefile** regułę o targecie z taką nazwą – i ją wykona.

Uwaga: Linijka z komendą (np. do kompilacji) musi zaczynać się od **tabulatora**.

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

```
$ make -f MyMakeFile2
```

Dzięki temu np. możemy poćwiczyć **make** na wyimkach z gotowego **makefile**.

- ▶ Przykładowa reguła w środku pliku **makefile_step1**:

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

- ▶ Wystarczy teraz wpisać polecenie: **\$ make -f makefile_step1**
Program wczyta **makefile_step1** 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:

- ▷ 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] .
- ▷ # na początku linijki oznacza komentarz .

⦿ Ważne uwagi odnośnie reguł:

- ▷ Reguła może być więcej.
Można nakazać make, aby procedował wybraną regułą. 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 reguły, do pliku makefile_step2:

```
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)`. Nasze reguły mogą teraz wyglądać tak:

```
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:

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

Zobaczmy `makefile_step3`, zawierający nasze modyfikacje, w tym z użyciem `$@` i `$$<` :

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

```
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++ $(CXXFlags) $(Includes) -c $$< -o $@
```

```
obj/Wymierna.o: src/Wymierna.C includes/Wymierna.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:

```
# Definitions of variables
```

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

```
# Here comes the main (final) rule:
```

```
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)
```

```
# Here come the rules for compilations of class files:
```

```
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 $@
```

```
# Rule for cleaning:
```

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

• Dalsza serializacja

- ▶ Nasz `makefile` wygląda długo. I właściwie, wiele reguł jest takich samych. Nauczmy się to serializować. Najpierw utworzymy 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 zmiennej `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_final` 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
```

```
# Here comes the main (final) rule:
```

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

```
# Here comes the implicit rule for compiling any class to .o file:
```

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

```
# Rule for cleaning
```

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

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