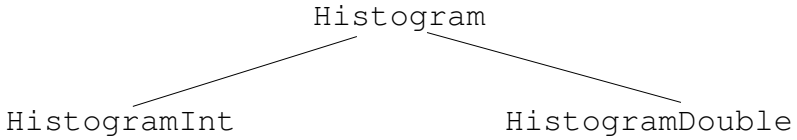
JP, Wtorek, 18.12.2012, 16:15-17:45

Zaczynamy powoli „przygodę” z jednym z najważniejszych elementów języka C++ - wirtualnością. Wirtualność oraz polimorfizm (tu rozumiany jako możliwość wyboru postaci funkcji w trakcie działania programu) stanowią zupełnie inne podejście do programowania niż programowanie proceduralne, które do tej pory wykorzystywaliśmy. Są najważniejszą cechą programowania orientowanego obiektowo (OOP – *Object Oriented Programming*) i w języku C++ mają ogromne możliwości, które objawiają się w pełni przy dużych projektach. To z tego powodu język ten jest wykorzystywany w pisaniu wielkich aplikacji, od systemów operacyjnych po gry komputerowe. Będziemy dalej rozwijać i modyfikować klasę `Histogram` wykorzystując dziedziczenie oraz wirtualność.
Uwaga! Zadanie ma charakter projektowy.

Przychodzi do nas klient i prosi o stworzenie programu do tworzenia histogramów i przeprowadzania na nich prostych operacji.

Załóżmy, że chcemy mieć dwa rodzaje histogramów. Jeden typ byłby taki jak nasz napisany na 6 zajęciach (12.11.2012) - posiadałby w każdym binie „liczbę zliczeń”, czyli wartość całkowitą oznaczającą krotność wystąpienia liczb z danego przedziału. Z kolei w drugim typie histogramu chcielibyśmy przechowywać nie tylko całkowite zliczenia, ale również niecałkowite (może się tak zdarzyć na przykład w wyniku podzielenia dwóch histogramów przez siebie). Najlepiej więc jest stworzyć klasę nadrzędną zawierajacą najwięcej wspólnych pól i metod obu typów histogramów oraz klasy podrzędne: dla każdego typu histogramu.
W tym celu stworzymy sobie pewną prostą hierarchię dziedziczenia klas. Ilustruje to schemat:



Klasa nadrzędna będzie się nazywać `Histogram`, zaś klasy z poszczególnymi typami histogramów to `HistogramInt` (gdzie „liczba zliczeń” jest zawsze całkowita) i `HistogramDouble` (gdzie „liczba zliczeń” jest zawsze rzeczywista). Powinna zawierać wszystkie pola i metody, które będą takie same w obu klasach. Część z tych metod będzie tak samo się nazywać w obu klasach, ale będzie robić trochę inne rzeczy (na przykład operować na innych typach danych), wobec tego trzeba użyć wirtualności (metod wirtualnych i czystko wirtualnych).

W ogólności powinny być zaimpelemntowane takie metody:
```
double GetBinWidth(); //zwraca szerokosc binu
void Fill(double x); //wypelnia odpowiedni bin histogramu
void PrintHistogram(); //wypisuje histogram na ekran
void Divide(Histogram* h1); //dzieli histogram przez histogram h1
void Add(Histogram* h1, double c=1); //dodaje do histogramu histogram h1 z waga c
```
Zadanie projektowe polega na optymalnym zaimplementowaniu powyższych metod. Przykładowo rozważmy dzielenie dla histogramu z typem liczb całkowitych. Jeśli rezultatem dzielenia jest liczba niecałkowita, to jako wartość w danym binie powinna zostać wzięta jej część całkowita (np. jeśli mamy w wyniku dzielenia 8.78, to zostawiamy wartość 8 ostatecznie).

Z uwagi na możliwe niecałkowite wartości w binach, metoda `void PrintHistogram()` niech wypisuje histogram na ekran w formacie:

```
Printing histogram HistTest
underflow: 2
overflow: 5
0: 1
1: 5.45
2: 15.23
3: 27.43
4: 45.27
5: 56.78
6: 39.57
```

Po stworzeniu klas należy stworzyć program, który przetestuje działanie całego projektu (taki, który moglibyśmy na przykład pokazać potencjalnemu klientowi). W tym celu można wykorzystać napisany na wspomnianych szóstych zajęciach generator liczb pseudolosowych.

Kilka słów na temat Makefile'i

Do tej pory korzystaliśmy z bardzo prostego pliku **Makefile**, który miał jedną komendę i której wykonanie uruchamiało nam kompilator g++ (zamiast wpisywanie tego „z palca” w linii poleceń), czyli np.:

```
all:
	g++ -Wall Klasa.cpp main.cpp -o main
```

W ogólności narzędzie `make` ma dużo więcej możliwości i jest wykorzystywane w dużych projektach informatycznych składających się z ogromnej liczby plików źródłowych. Zrobimy sobie dzisiaj bardziej rozbudowany **Makefile**.

Struktura plików **Makefile** składa się z tak zwanych reguł i jest następująca:

```
CEL: SKŁADNIKI
	KOMENDA
```

gdzie `CEL` jest nazwą pliku wynikowego, który jest tworzony z plików wymienionych jako `SKŁADNIKI`, zaś `KOMENDA` jest komendą (przed nią musi stać tabulacja), która tworzy `CEL` z plików w `SKŁADNIKI`.

Przykład takiej reguły:

```
main: main.cpp klasa.cpp
	g++ -Wall klasa.cpp main.cpp -o main
```

Pole `SKŁADNIKI` nie jest wymagane, jeśli pliki składowe wymienione są w komendzie *explicite* (w tym przypadku nie muszą zatem tam być podane – podobnie jak w naszym prostym Makefile'u).

W dużych projektach nasz ostateczny wynikowy plik tworzony jest z wielu reguł pomocniczych. Na przykład

```
main: klasa.o
	g++ -Wall klasa.o main.cpp -o main
klasa.o: klasa.cpp
	g++ -Wall klasa.cpp -c -o klasa.o
```

Załóżmy, że mamy klasę składającą się z plików **klasa.cpp** oraz **klasa.h**. Druga komenda (wywołanie g++ z flagą `-c`) tworzy nam plik **klasa.o**, który jest tzw. „object file” - jest to skompilowany fragment kodu (nie program), zawierający wszystkie zmienne i metody klasy. Wiele „object files” można łączyć w biblioteki. Ponadto tego typu organizacja projektu pozwala na kompilowanie wybranych fragmentów i ewentualne poprawianie błędów tylko w danej części bez kompilowania całego programu.

W pliku **Makefile** możemy też używać zmiennych, podobnie jak w skryptach bash'owych. Na przykład (dla przykładu tym razem z dwoma klasami):

```
OBJS=klasa1.o klasa2.o

main: $(OBJS)
     g++ -Wall &(OBJS) main.cpp -o main
klasa1.o: klasa1.cpp
     g++ -Wall klasa1.cpp -c -o klasa1.o
klasa2.o: klasa2.cpp
     g++ -Wall klasa2.cpp -c -o klasa2.o
```

Często tworzymy również zmienne standardowe, które ułatwiają nam modyfikację parametrów zarządzających kompilacją projektu. Stwórzmy więc zestaw takich zmiennych. Zazwyczaj używamy nazw takich, jak podane poniżej:

`CXX` – kompilator C++

`CXXFLAGS` – flagi kompilatora C++ (np. `-Wall`)

`LFLAGS` – flagi linkera (gdy dołączamy jakąś zewnętrzną bibliotekę, jeszcze tego nie robiliśmy, więc zostawiamy tą zmienną pustą)

Możemy więc teraz napisać nasz **Makefile** jako:

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

main: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) &(OBJS) main.cpp -o main
klasa1.o: klasa1.cpp
     $(CXX) $(CXXFLAGS) klasa1.cpp -c -o klasa1.o
klasa2.o: klasa2.cpp
     $(CXX) $(CXXFLAGS) klasa2.cpp -c -o klasa2.o
```

Oprócz tego, możemy jeszcze korzystać z tak zwanych zmiennych automatycznych. Zmienne automatyczne to zmienne dynamiczne, które zmieniają się w zależności od przetwarzanego pliku:

`<` - aktualnie przetwarzany plik z listy składników

`@` - nazwa pliku docelowego (czyli `CEL`)

`^` - składniki (wkleja w to miejsce to co znajduje się w polu `SKŁADNIKI`)

Przy użyciu zmiennych automatycznych nasz **Makefile** będzie zatem miał postać:

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

main: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main.cpp $^ -o $@
klasa1.o: klasa1.cpp
     $(CXX) $(CXXFLAGS) $< -c -o $@
klasa2.o: klasa2.cpp
     $(CXX) $(CXXFLAGS) $< -c -o $@
```

Widzimy, że pliki z pola `SKŁADNIKI` wybierane są teraz automatycznie przez zmienne automatyczne. Kolejnym uproszczeniem jest wykorzystanie wzorca reguły. Pozwoli to na skrócenie pliku **Makefile**. Widzimy w powyższym przykładzie, że musimy *explicite* stworzyć pliki .o dla obu klas. Można sobie wyobrazić, że w przypadku wielkich projektów takich plików (zatem i linijek w pliku **Makefile**) byłoby tysiące. Z wykorzystaniem wzorca nasz **Makefile** skracamy do postaci:

```
CXX=g++
```

```
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

main: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main.cpp $^ -o $@
$(OBJS): %.o: %.cpp
     $(CXX) $(CXXFLAGS) $< -c -o $@
```

Co więcej, reguła tworząca pliki .o jest regułą domyślną i nie trzeba jej *explicite* pisać. Poniższy **Makefile** zrobi więc dokładnie to samo (stworzy pliki .o i z nich program wynikowy):

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

main: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main.cpp $^ -o $@
```

Oprócz reguł do tworzenia możemy stworzyć regułę do usuwania skompilowanego programu, regułę `clean`.

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

main: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main.cpp $^ -o $@
clean:
     rm -rf *.o main
```

Jak widzimy, reguła ta usuwa nam wszystkie pliki powstałe przy kompilacji (pliki .o oraz plik z programem). Co jednak zrobić, gdy mamy 2 programy wynikowe i chcemy używać jednego pliku **Makefile**? Nic trudnego, piszemy sobie dwie reguły i możemy też dopisać regułę `all`, która od razu skompiluje nam oba (wywoła polecenia `main1` oraz `main2`), np.:

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

all: main1 main2
main1: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main1.cpp $^ -o $@
main2: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main2.cpp $^ -o $@
clean:
     rm -rf *.o main
```

Reguły `clean` i `all` są regułami specjalnymi w tym sensie, że `clean` i `all` nie są nazwami plików. Gdyby jednak zdarzyło się, że w katalogu bieżącym istnieje plik o nazwie `all` lub `clean` to reguły te mogłyby nie działać! Aby temu zapobiec trzeba by poinstruować make'a, że `all` i `clean` nie są nazwami plików. Do tego celu służy specjalna reguła o nazwie `.PHONY.`:

```
CXX=g++
CXXFLAGS=-Wall
LFLAGS=

OBJS=klasa1.o klasa2.o

all: main1 main2
main1: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main1.cpp $^ -o $@
main2: $(OBJS)
     $(CXX) $(LFLAGS) $(CXXFLAGS) main2.cpp $^ -o $@
clean:
     rm -rf *.o main
.PHONY: all clean
```

Tak przygotowany **Makefile** możemy sobie następnie łatwo modyfikować i dostosowywać w celu zarządzania naszą kompilacją (zmieniając flagi kompilatora, pliki składowe projektu itp.). Z reguły więc, w przypadku niedużych projektów, mamy szablon takiego Makefile'a, który tylko dostosowujemy sobie.

Do zrobienia:

- Stworzyć Makefile do prostego programu, który wczutuje z pliku liczby i wypełnia histogramy `HistogramInt` oraz `HistogramDouble`.