More on classes and Makefile

Tutorial 6b

Katja Mankinen

19 October 2017

Aim

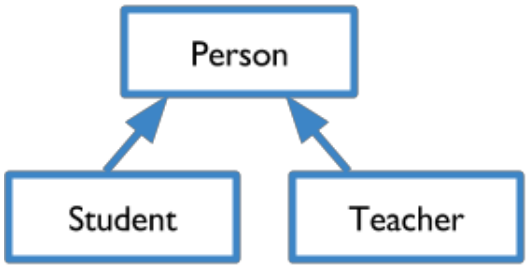
After this tutorial, you're able to decrease your programming workload by

- using inheritance in your classes
- creating Makefiles.

You should preferably finish all the steps today. It may look like a lot of work, but it is actually very doable! You can immediately start doing the exercises at your own pace. If you have any problems or questions, just ask!

1. Adding inheritance

Last week you created a class `Student`. Today you will implement two new classes: `Person` and `Teacher`. A `Person` class will act as a base class (parent class), and both `Student` and `Teacher` are derived classes (child classes, sub classes) of `Person`. A child class inherits both member functions member variables from the parent. These variables and functions become members of the derived class.



1.1 Creating a Person

Similarly to what you did last week, create a new class in a new header file: `class Person` in `Person.h`. Add data members that are common to any type of person, regardless profession or other properties: `name` and `personNumber`. You can make all of the variables and functions public. This is only for the sake of simplicity in our example: normally we should of course make the variables private.

Test your implementation in a new source code file: `main.cpp`.

1.2 Add inheritance

Last week you created a class `Student` with data members of age, GPA, and student ID. Our students should have names and person numbers as well. This information is a part of Person class, and we have basically two options:

1. Add name and person numbers to the Student class. This would duplicate the code that already is written in Person class, so this is a bad but simple choice.

2. Ask Student to inherit name and person number from Person, because Student is-a Person. (it wouldn't work other way around, would it?)

Now, of course, the second option is way to go. Thus, change your existing Student class to inherit (with public access) from a class Person.

(note: we could have "age" in Person class as well, but there's no need to change it now unless you want to).

Hint!: Remember to include `Person.h` in your `Student.h`! In case you get errors about previous definitions, make sure that you do not actually define your Person class twice. In addition, you are probably missing *include guardians*. You need to make sure that the code is not duplicated, i.e. you're not defining things twice when including other header files. You can use include guards as follows in your header files, for example in Parent.h:

```
#ifndef PARENT_H
#define PARENT_H

//your class comes here

#endif /* PARENT_H */
```

Here, the first inclusion of `Parent.h` causes the macro `PARENT_H` to be defined. When `Student.h` includes Student.h the second time, the `#ifndef` test returns false, and the preprocessor skips down to the #endif, which avoids the second definition. The program compiles correctly.

1.3 No fun without teachers!

Create another class that also inherits from Person: `Teacher`. Again, create a new header file for your class declaration. Teacher class should have two new member variables: employeeID and salary. It also inherits name and person number from Person.

Note that now Teacher and Student do not have any direct relationship, even though they both inherit from Person.

1.4 Extra: adding polymorphism

This is an optional exercise. You can directly jump to Makefiles!

As you noticed, both Student and Teacher classes have information on IDs. Students have student ID and teachers employee ID. This is a bit of code duplication and could be avoided by using polymorphism. Modify your code so that all three classes have member functions of a same name to set and get IDs. Make them virtual in Person class, and override that function in your Student and Teacher classes. To make sure your code implementation works, add different printouts to your functions.

2. Makefile

As you've already learned, compilation process is basically three steps long:

1. the preprocessor copies the contents of the header file into the source code file
2. the source code file is compiled into .o object modules
3. object modules are linked together to create a single executable file

Today we will use our newly made class files to illustrate how to write a Makefile to compile a program. Instead of writing all the code in one file and compiling that one file, you can write multiple .cpp files and compile them separately. Usually `.cpp` files consist of the implementations of all methods in a class and standalone functions (i.e. functions that are not part of any classes). The corresponding header `.h` files consist of the class declarations and function prototypes.

We already have compiled several .cpp files during our exercises with g++. What if you want to compile many .cpp files that are some way connected to each other? And what if your project has tens of .cpp files?

Solution is **Makefile**! Makefile is a set of targets and rules to build them. It saves you from long and complex g++ commands. Instead, it will be enough to type `make` and it will do all the hard work for you.

When you have many .cpp files, it is not necessary to recompile them all when you make change to one of them. You only need to recompile a small subset of all the files.

In our current case, we have many files:

- Student.h, header file for the Student class
- Student.cpp, implementation file for the Student class
- Teacher.h, header file for the Teacher class
- Teacher.cpp, implementation file for the Teacher class
- Person.h, header file for the Person class
- Person.cpp, implementation file for the Person class
- main.cpp, a main program

Before moving on, let's define some words in a short glossary:

- `target`: the name of an executable or object file that is generated by g++
- `prerequisites` or `dependencies`: a list of files that are needed to create the target
- `command`: an action that make carries out; usually compilation or linking

In addition, you should know some compiler options:

```
g++ -Wall -g -o student student.cpp -I ./
```

- `-Wall`: prints "all" Warnings
- `-g`: enables extra debugging information
- `-o`: specifies the output filename
- `-I`: adds the directory to the list of directories to be searched for header files
- `-c`: compile or assemble the source code files, but do not link

You can find many many more options from g++ man page.

Let's now get familiar with Makefiles: how to write and use them.

3. Anatomy of a Makefile

Create a file called `Makefile`. Now when you call `make` in the terminal window, it will search for `Makefile`. Makefile is Makefiles can look very complex and be long, but when it's working, you just need to type in "make" and everything else is done automatically for you.

As a pseudo-code a Makefile looks something like

```
<the file> : <needs these files>
[TAB] <the file is created by this command>
```

which is essentially the same as

```
target : dependency1 dependency2 ...
[TAB] command1
[TAB] command2
```

Here's a very simple example of a Makefile:

```
# Makefile

all: main.o function.o
        g++ -o MyProgram main.o function.o

main.o: main.cpp
        g++ -c main.cpp -I ./

function.o: function.cpp
        g++ -c function.cpp -I ./

clean:
        rm -rf *.o
        rm -rf MyProgram
```

- Lines starting with # are comments
- `all` is a special target which depends on main.o and function.o, and has the command to make g++ link the two object files into the executable MyProgram
- `main.o` is a file name target that depends on main.cpp. It has the command to compile main.cpp to produce main.o.
- `function.o` is a target that depends on function.cpp and function.h. It has the command to compile the function.cpp file to produce function.o.
- `clean` is a special target that has no dependencies. It specifies the commands to clean the compilation outputs.

To use this makefile to create the executable file called MyProgram, type:

```
make
```

Now you can use your newly created executable as usual, by typing

```
./MyProgram
```

To delete the executable file and all the object files from the directory, type:

```
make clean
```

Now your task is to create a Makefile for your current project consisting of students, teachers, persons and main file.

4. More on Makefiles

You can use shell commands in your Makefile to decrease your workload even more. For example, you could add new variables so that you do not need to change everything by hand:

```
# Makefile

# Add new variables to control Makefile operation
# defining compiler and options for more debug information

CXX = g++
```

```
CXXFLAGS = -Wall -g

# Targets

main: main.o function.o function2.o
  $(CXX) $(CXXFLAGS) -o main main.o function.o function2.o

# The main.o target can be written more simply:

main.o: main.cpp function.h function2.h
  $(CXX) $(CXXFLAGS) -c main.cpp

function.o: function.h

# the target filename : (colon) files it depends on
function2.o: function2.h function.h
```

A variable begins with a $. There are also automatic variables:

- `$@`: the target filename
- `$*`: the target filename without the file extension
- `$<`: the first prerequisite filename
- `$^`: the filenames of all the prerequisites, separated by spaces, discard duplicates.
- `$+`: similar to $^, but include duplicates
- `$?`: names of all prerequisites that are newer than the target, separated by spaces.

Using them, we can switch from easily human-readable part of the makefile

```
main: main.o
   g++ -o MyProgram main.o
```

to

```
# $@ matches the target; $< matches the first dependent
main: main.o
  g++ -o $@ $<
```

And so on. How would you change your Makefile to use some of the automatic variables?

Finally, go to Live@Lund, navigate to Lesson plans and read through a file "Makefile". It has an example implementation of a Makefile and many comments.

Makefiles are a large topic. There are even more capabilities in them! However, the aim of this exercise was to introduce Makefiles and their basic syntax to you. Even if you never write a Makefile again, you probably need to read and even modify Makefiles written by other great scientists. But then, Google is again your friend!