Multiple File Programs

Pre-processor

CIS 15 : Spring 2007

Functionalia

Today:

- Multiple File Compilation
- Pre-processor
-

Multiple File Programs

By now your programs have and will grow to be unwieldy in one `.cpp` file.

You can split your program up into multiple files.

In C++ - it is customary to separate your Classes so you have one class per file.

Multiple File Programs

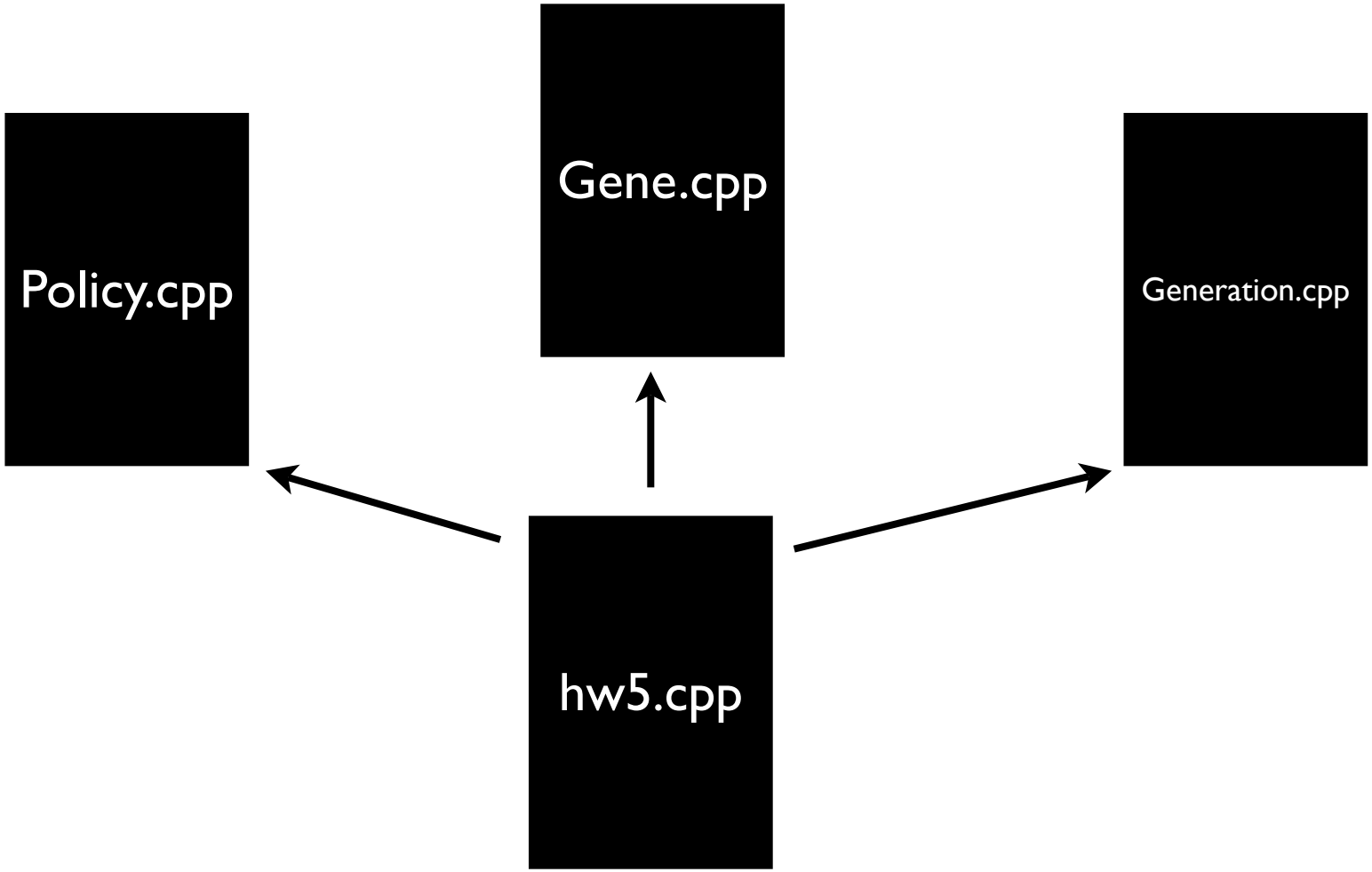
Homework 5 is a file that contains 3 classes:

`Policy, Gene, Generation`

Multiple File Programs

Extract the 3 classes and put them into their own file.

`Policy.cpp, Gene.cpp, Generation.cpp`

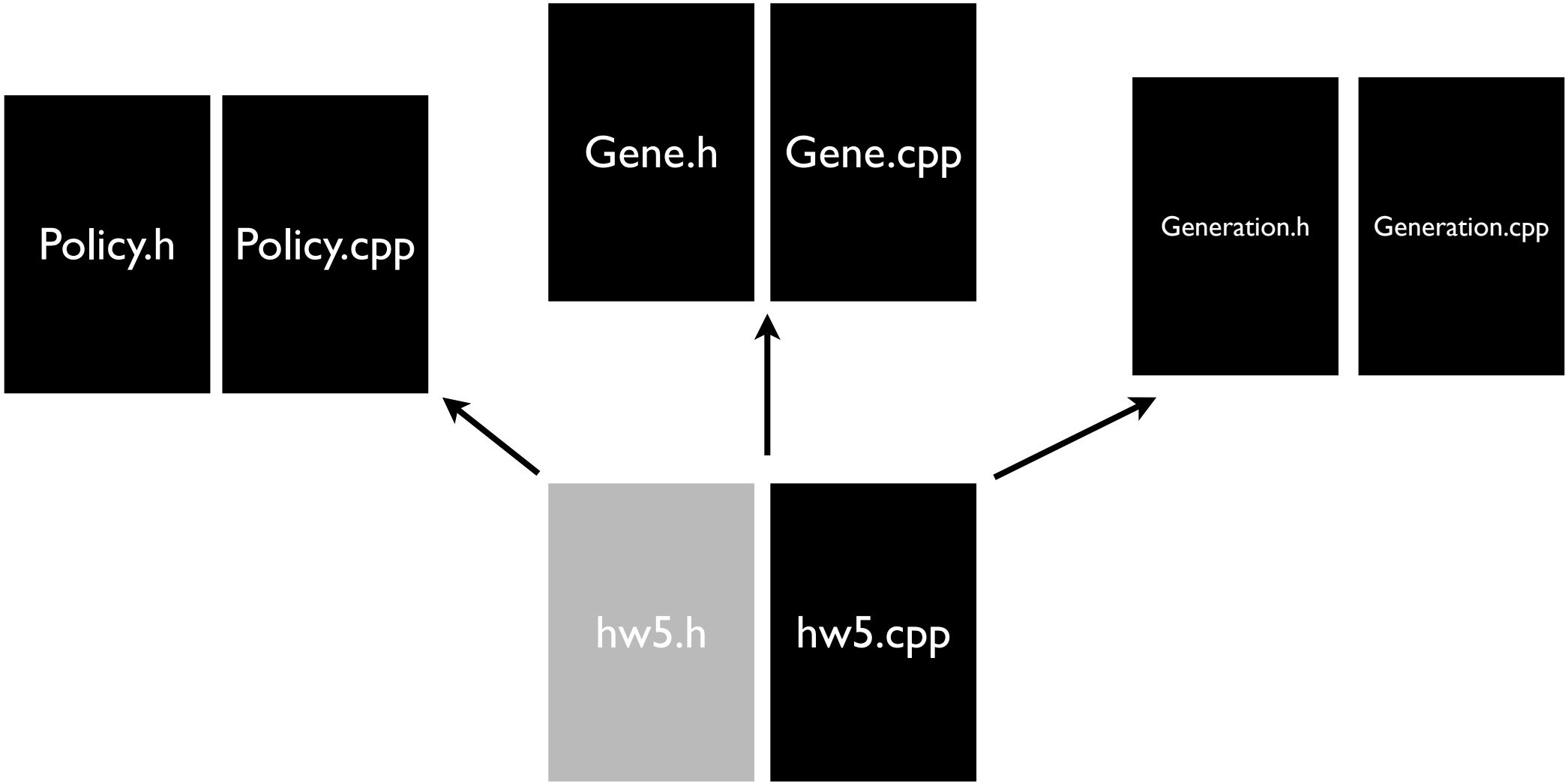


Multiple File Programs

In fact, we will split each on of our .cpp files into two files.

Header File *(.h)* – contains Class Definitions, Prototypes, Constants

Code File (*.cpp*) - contains External function definitions, (i.e. **`main()`**)



Gene.h - holds the Class

```
////////////////////////////////////////////////////////////////////
//    Gene Class:
//        Holds information about one Gene.  Information includes the
//        genotype, it's fitness rating (according to the current Policy),
//      the generation it comes from, and whether this Gene has been mutated.

class Gene {
      private:
          static const int SIZE = 10;
          string genotype;
          double fitness;
          int generation;
          bool mutant;

      public:
          Gene(Policy & policy, int gen_id);
          Gene(Gene * parent1, Gene * parent2, int gen_id);

          double calcFitness(Policy & policy);
          double getFitness() const { return fitness; }
          void mutate(Policy & policy);
};
```

Gene.cpp holds the external member functions

```
////////////////////////////////////////////////////////////////////
// Gene Class

// Constructor: Randomly generates a Gene from the provided Policy.
// The Gene's generation is set to gen_id.
// (Used in this simulation to generate the initial Generation of Genes)

Gene::Gene(Policy & policy, int gen_id = 0)
{

}

// Constructor: This generates a Gene from two parent Genes (provided by pointers),
// by choosing a split point randomly.  The first half of parent1 becomes the beginning
// of the new Gene's genotype, and the seocond half of parent2 is appended to the new
// genotype.
// The Gene's generation is set to gen_id.

Gene::Gene(Gene * parent1, Gene * parent2, int gen_id)
{

}

// Calculates the Gene's fitness according to the Policy
// Fitness is calculated by summing the scores of all of the traits that
// are present in the Gene's genotype string.
double Gene::calcFitness(Policy & policy)
{
```

Use `#include` to link files together

```
#include "Gene.h"

////////////////////////////////////////////////////////////////////
// Gene Class

// Constructor: Randomly generates a Gene from the provided Policy.
// The Gene's generation is set to gen_id.
// (Used in this simulation to generate the initial Generation of Genes)

Gene::Gene(Policy & policy, int gen_id = 0)
{

}

// Constructor: This generates a Gene from two parent Genes (provided by pointers),
// by choosing a split point randomly.  The first half of parent1 becomes the beginning
// of the new Gene's genotype, and the seocond half of parent2 is appended to the new
// genotype.
// The Gene's generation is set to gen_id.

Gene::Gene(Gene * parent1, Gene * parent2, int gen_id)
{

}

// Calculates the Gene's fitness according to the Policy
// Fitness is calculated by summing the scores of all of the traits that
```

Use quotes to indicate a local *header* file

```
#include "Gene.h"

//////////////////////////////////////////////////////////////////
// Gene Class

// Constructor: Randomly generates a Gene from the provided Policy.
// The Gene's generation is set to gen_id.
// (Used in this simulation to generate the initial Generation of Genes)

Gene::Gene(Policy & policy, int gen_id = 0)
{

}

// Constructor: This generates a Gene from two parent Genes (provided by pointers),
// by choosing a split point randomly.  The first half of parent1 becomes the beginning
// of the new Gene's genotype, and the seocond half of parent2 is appended to the new
// genotype.
// The Gene's generation is set to gen_id.

Gene::Gene(Gene * parent1, Gene * parent2, int gen_id)
{

}

// Calculates the Gene's fitness according to the Policy
// Fitness is calculated by summing the scores of all of the traits that
```

hw5.cpp would #include all of the Header Files

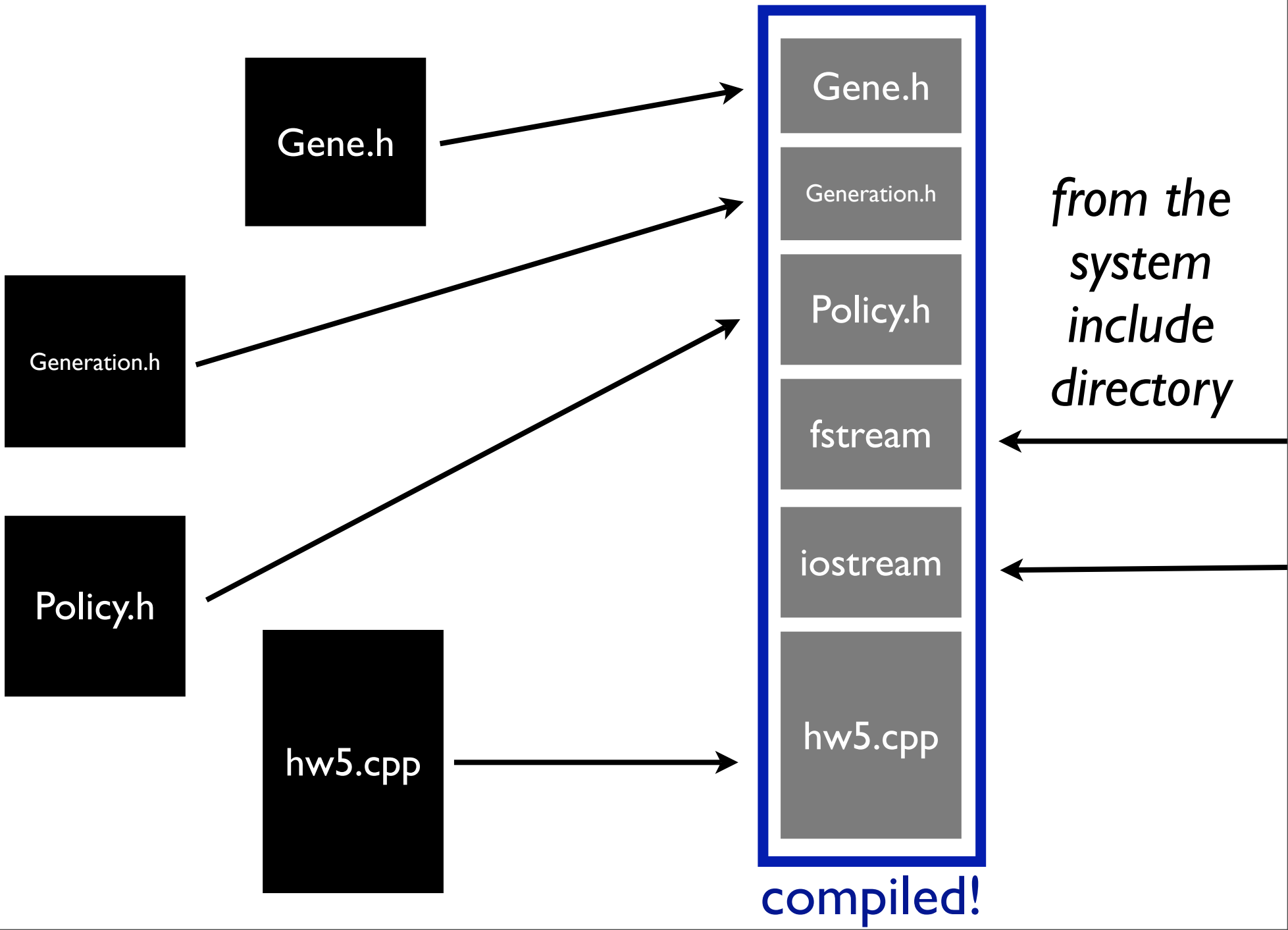
```
#include "Gene.h"
#include "Generation.h"
#include "Policy.h"

#include <fstream>
#include <iostream>

int main()
{
    ...
    ...
    ...
}
```

any order

`#include` glues files together



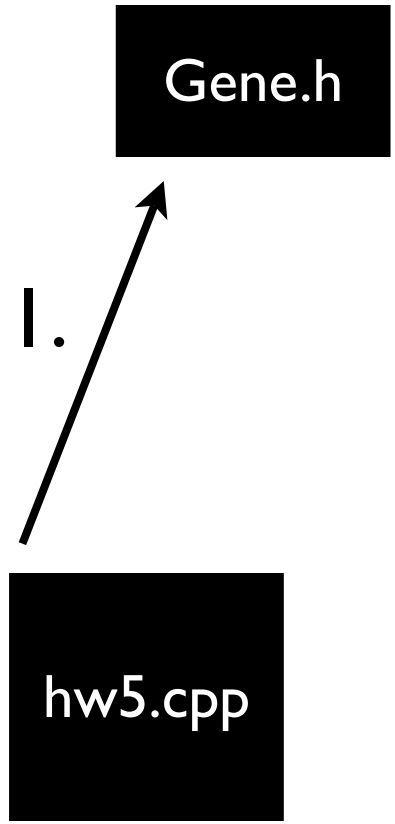
Problem! Circular Definition

hw5.cpp

```
#include "Gene.h"
#include "Generation.h"
#include "Policy.h"
...
```

Generation.h

```
#include "Gene.h"
...
```



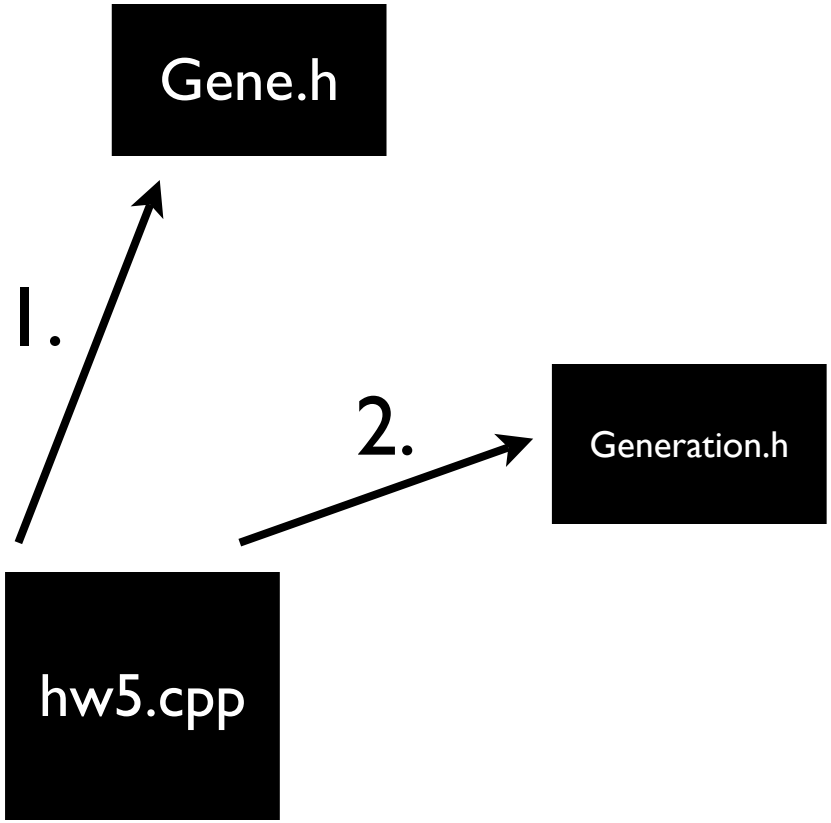
Problem! Circular Definition

hw5.cpp

```
#include "Gene.h"
#include "Generation.h"
#include "Policy.h"
...
```

Generation.h

```
#include "Gene.h"
...
```



Problem! Circular Definitions

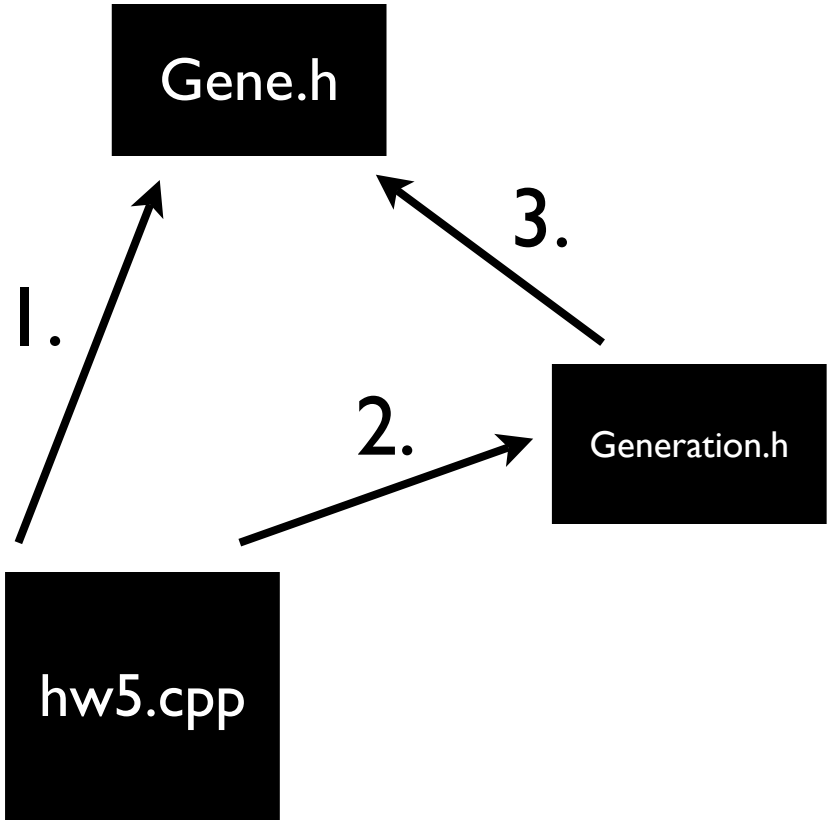
hw5.cpp

```
#include "Gene.h"
#include "Generation.h"
#include "Policy.h"
...
```

Generation.h

```
#include "Gene.h"
...
```

Multiple Definitions of the Gene Class!!!



Solution ! Use *include guards*

hw5.cpp

```
#include "Gene.h"
#include "Generation.h"
#include "Policy.h"
...
```

Leave your .cpp files alone.

Generation.h

```
#include "Gene.h"
...
```

Solution ! Use *include guards*

Generation.h

```
#ifndef GENERATION_H
#define GENERATION_H
#include "Generation.h"
...

#endif
```

Gene.h

```
#ifndef GENE_H
#define GENE_H

...

#endif
```

1. Create a `#define`'d constant (i.e. a Pre-processor constant) for the first time the pre-processor passes over it.

Solution ! Use *include guards*

Generation.h

```
#ifndef GENERATION_H
#define GENERATION_H
#include "Gene.h"
...

#endif
```

1. Create a `#define`'d constant (i.e. a Pre-processor constant) for the first time the pre-processor passes over it.

Gene.h

```
#ifndef GENE_H
#define GENE_H

...

#endif
```

2. Skip Code for the rest of the pre-processor passes.

Now what about those `.cpp` files?

Every file gets compiled separately into ***object*** files.

At the last step they get ***linked*** together into the final binary

```
$ ls
Gene.h Gene.cpp Generation.h Generation.cpp Policy.h Policy.cpp
hw5.cpp

$ g++ Gene.cpp -c -o Gene.o
$ g++ Generation.cpp -c -o Generation.o
$ g++ Policy.cpp -c -o Policy.o
$ g++ hw5.cpp -c -o hw5.o
$ g++ Gene.o Generation.o Policy.o hw5.o -o hw5.exe

$ hw5.exe policy.txt gen 10
```

Now what about those .cpp files?

-c flag compiles but does not ***link***

```
$ ls
Gene.h Gene.cpp Generation.h Generation.cpp Policy.h Policy.cpp
hw5.cpp

$ g++ Gene.cpp -c -o Gene.o
$ g++ Generation.cpp -c -o Generation.o
$ g++ Policy.cpp -c -o Policy.o
$ g++ hw5.cpp -c -o hw5.o
$ g++ Gene.o Generation.o Policy.o hw5.o -o hw5.exe

$ hw5.exe policy.txt gen 10
```

Use a wildcard (*) to compile faster

```
$ ls
Gene.h Gene.cpp Generation.h Generation.cpp Policy.h Policy.cpp
hw5.cpp
```

```
$ g++ *.cpp -c
```

All of the `.cpp` files are compiled and the outputs become `.o` files.

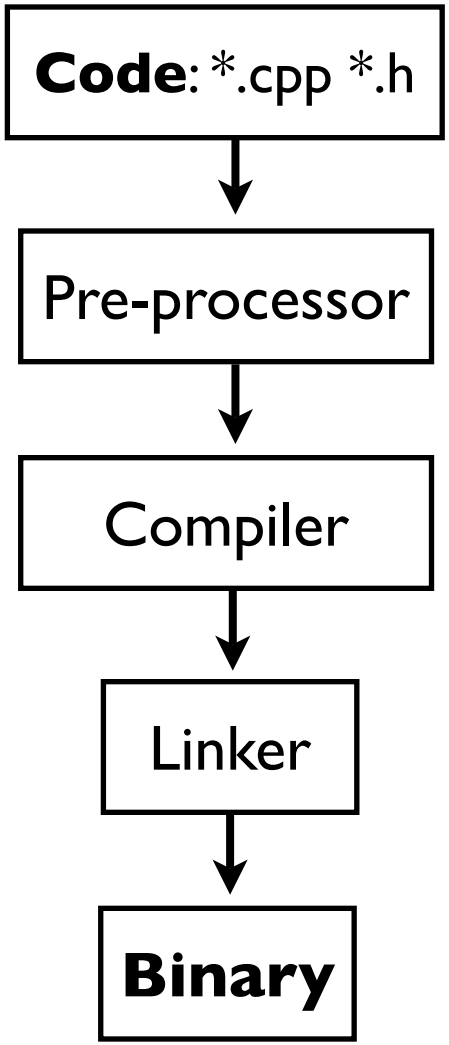
So, `Generation.cpp` becomes `Generation.o`

```
$ g++ *.o -o hw5.exe
```

```
$ hw5.exe policy.txt gen 10
```

Pre-processor directives

Lines included in the code that direct the pre-processor in it's interpretation of the code.



Directives begin with a # and use no semi-colon.

```
#include

#define

#ifdef, #ifndef, #endif

#if, #elif, #else, #endif

#error
```

Important to note: pre-processor **does not** understand C++.

#include

Combines files (we just covered this):

System

```
#include <file>
```

Local Directory

```
#include "file.h"
```

Local Directory can include file paths

```
#include "include/file.h"
```

#define

Does a replacement identifier:

Constants

```
#define CONSTANT 1
```

```
int small[CONSTANT];  ⟶  int small[1];
```

Macros

```
#include getmax(a,b) ((a)>(b)?(a):(b))
```

```
int value = getmax(1,x+1);  ⟶  int value = ((1)>(x+1)?(1):(x+1));
```

#define Macros

Macros have these operators

String Operator

```
#define qw(a) # a
```

```
cout << qw(hi how are you);  ⟶  cout << "hi how are you";
```

Concatenate Instruction

```
#define glue(a,b) a ## b
```

```
glue(c,out) << "test";  ⟶  cout << "test";
```

#ifdef, #ifndef, #endif

Conditionals on Definitions

Include Guard

```
#ifndef GENE_H
#define GENE_H

... code ...

#endif
```

#if, #elif, #else, #endif

Conditionals on Defined Values

Compile Time Defines

```
#if GEN_SIZE < 20
   ... code ...
#elif GEN_SIZE < 100
   ... code ...
#else
   ... code ...
#endif
```

#error

Prints a compile-time error

Compile Time Defines

```
#ifndef __cplusplus
#error A C++ compiler is required!
#endif
```

Error prints the file name, the line number, and the error message.

Predefined Macros

__LINE__	Integer value representing the current line in the source code file being compiled.
__FILE__	A string literal containing the presumed name of the source file being compiled.
__DATE__	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began.
__TIME__	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.
__cplusplus	An integer value. All C++ compilers have this constant defined to some value.

```
// standard macro names
#include <iostream>
using namespace std;

int main()
{
  cout << "This is the line number " << __LINE__;
  cout << " of file " << __FILE__ << ".\n";
  cout << "Its compilation began " << __DATE__;
  cout << " at " << __TIME__ << ".\n";
  cout << "The compiler gives a __cplusplus value of "
       << endl << __cplusplus << endl;
  return 0;
}
```

```
This is the line number 7 of file macro.cpp.
Its compilation began Apr 26 2007 at 09:50:04.
The compiler gives a __cplusplus value of 1
Chipp:~/Teaching/Programming_CIS15/Le
```