Let's work through the drills of chapter 8, starting with this one:

1. `my.h, my.cpp,` and `use.cpp`. The header file `my.h` contains

   ```
   extern int foo;
   void print_foo();
   void print(int);
   ```

   The source code file `my.cpp #includes my.h` and `std_lib_facilities.h`, defines `print_foo()` to print the value of `foo` using `cout`, and `print(int i)` to print the value of `i` using `cout`.

   The source code file `use.cpp #includes my.h`, defines `main()` to set the value of `foo` to 7 and print it using `print_foo()`, and to print the value of 99 using `print()`. Note that `use.cpp` does not `#include std_lib_facilities.h` as it doesn't directly use any of those facilities.

   Get these files compiled and run. On Windows, you need to have both `use.cpp` and `my.cpp` in a project and use `{ char cc; cin>>cc; }` in `use.cpp` to be able to see your output.

   Hint: You need to `#include <iostream>` to use `cin`.

```
// my.h
#pragma once
#include <iostream>

extern int foo;
void print_foo();
void print(int);
```

```
//my.cpp
#include "my.h"
#include "std_lib_facilities.h"

void print_foo() {
    std::cout << foo;
}

void print(int i) {
    std::cout << i;
}
```

```
//use.cpp
#include "my.h"

int main() {
    int foo = 7;
    print_foo();
    print(99);
    //char cc{ '1' };
    std::cin.get();
}
```

The preceding three program listings were my first (perhaps, naive) attempt at this. However, it doesn't compile and leads to this linker error: `1>my.obj : error LNK2001: unresolved external symbol "int foo" (?foo@@3HA)`

Searching `stackexchange.com` we find the following (paraphrased):

> If you want to have one global variable available from different `cpp`, you should make two things: definition in one `cpp` and `extern` declaration in `my.h`.
>
> For example:
>
> `// my.h extern int foo;`
>
> and
>
> `// my.cpp int foo;`
>
> Then in any file where `foo` is needed:
>
> `#include "my.h"` And, of course, `my.cpp` have to be part of project (compiled with other files)

> "Every definition is (by definition;-)) also a declaration, but only some declarations are also definitions. Here are some examples of declarations that are not definitions; if the entity it refers to is used, each must be matched by a definition elsewhere in the code:
>
> ```
> double sqrt(double); // no function body here
> extern int a; // "extern plus no initializer" means "not definition"
> ```

Ok, so when we declare `foo` on line 5 of `my.h`, using the `extern` modifier, it's to emphasize that it's not defined: it's emphasizing that it's just a declaration and not a definition. It's not initialized there, so even without the `extern` it would not be defined, but if we drop the `extern` modifier, we get another linker error: `error LNK2005: "int foo" (?foo@@3HA) already defined in my.obj`.

But...it's **not** defined in `my.h`! Well, error messages are not always error-free. Let's read some stack overflow: https://stackoverflow.com/questions/31925442/defining-an-extern-variable-in-the-same-header-file

> If you want to have one global variable available from different `.cpp`s, you should make two things: definition in one `cpp` and `extern` declaration in the header.
>
> For example:
>
> ```
> // my.h
> extern int foo;
> ```
>
> and
>
> ```
> // my.cpp
> int foo;
> ```
>
> Then in any file where foo is needed:
>
> `#include "my.h"` And, of course, `use.cpp` have to be part of project (compiled with other files)

With this in mind, I modified `my.cpp`

```
#include "my.h"

int foo; // This foo has a default value of 0.
         // Use int foo = 17; to define it as 17, say.

void print_foo() {
    std::cout << foo;
}

void print(int i) {
    std::cout << i;
}
```

That gets rid of the errors, but produces the output `099`, meaning that `foo` is 0, not 7. If you want it to have a value, you must define it in `my.cpp` (`int foo = 17;`, not just `int foo;` and that will supersede any definition given in `main()`, so, even though we have `int foo = 7;` in `main()`, the output of `printfoo()` is 17, not 7. Any function that `#includes` `my.h` will reference the same `foo`.

Note that if we add the statement `print(foo)` to `main()` then the output is 17997. So you've got two `foos` in play here without colliding?! Generally, this is to be avoided!

Here's the final three files:

```
//-------my.h------------
#pragma once
#include <iostream>

extern int foo;
void print_foo();
void print(int);

//-----------my.cpp---------
#include "my.h"

int foo = 17;

void print_foo() {
    std::cout << foo << std::endl;
}

void print(int i) {
    std::cout << i << std::endl;
}

//---------use.cpp---------
#include "my.h"

int main() {
    int foo = 7;
    print_foo();
    print(99);
    print(foo);
    std::cin.get();
}
```

2. Write three functions `swap_v(int,int)`, `swap_r(int&,int&)`, and `swap_cr(const int&,const int&)`. Each should have the body `{ int temp; temp = a, a=b; b=temp; }`

   where `a` and `b` are the names of the arguments.

   Try calling each swap like this

```
int x = 7;
int y =9;
swap_?(x,y); // replace ? by v, r, or cr
swap_?(7,9);
```

```
const int cx = 7;
const int cy = 9;
swap_?(cx,cy);
swap_?(7.7,9.9);
double dx = 7.7;
double dy = 9.9;
swap_?(dx,dy);
swap_?(7.7,9.9);
```

Which functions and calls compiled, and why? After each swap that compiled, print the value of the arguments after the call to see if they were actually swapped. If you are surprised by a result, consult §8.6.

Ok let's start with this program:

```
// G. Hagopian:  PPP 8 Drill #2
#include <iostream>

void swap_v(int a, int b) { int temp; temp = a; a = b; b = temp; }
void swap_r(int& a, int& b) { int temp; temp = a; a = b; b = temp; }
//void swap_cr(const int& a, const int& b) { int temp; temp = a; a = b; b =
   temp; }

int main() {
    int x = 7;
    int y = 9;
    swap_v(x, y); // When pass by value is used only the copies of x and y
                  // local to swap_v() are swapped.  After returning, x is
                  // still 7 and y is still 9
    std::cout << "\nx = " << x << ", and y = " << y;
    std::cin.get();
}
```

If you want to swap the values, pass by reference: `swap_r(x,y)` will swap the values in the actual memory locations of `x` and `y`. `swap_v(x,y)` will swap the local variables `a` and `b`, which are copies of `x` and `y`, but the local variables are destroyed when control is returned to `main()`, so the swap doesn't "stick."

If integer literals are passed to the `swap_v()` function by, say, `swap_v(7,9)`, then there is no compiler error since 7 and 9 are then assigned to the local variables of the swap function and a pointless swap is performed without a hitch...except that, again, with pass-by-value, these variables are destroyed when control returns to `main()`.

If you call `swap_r(7,9)` you get the error `error C2664: 'void swap_r(int &,int &)': cannot convert argument 1 from 'int' to 'int &'`. Here the function `swap_r()` wants the address of a variable (a reference to the variable) and you give that address the literal value 7. That just doesn't work. It's true that addresses take the form of integers, but they are not of the type `int`.

The function `swap_cr()` can't work because `a` is passed as a reference to a `const int` and so can't be assigned some other value, as is attempted on the line `a = b;`

If you create `const int` values `cx = 7;` and `cy = 9` then the call to `swap_v(cx,cy)` will not induce a compiler error, but it also won't swap `cx` and `cy`, though it will swap copies of them. Interestingly (is this interesting?...kind of deadly technicalities...) if you call `swap_r(cx,cy)` you get the compiler error `error C2664: 'void swap_r(int &,int &)' : cannot convert argument 1 from 'const int' to 'int &'` with the `note: Conversion loses qualifiers`

Similarly, the call to swap floating point literals 7.7 and 9.9 will only compile with `swap_v()`, and then it will only swap copies...and those copies will narrow the `float` type to `int` values 7 and 9.

You might expect the function `swap_r(dx,dy)` to behave the same as `swap_r(x,y)` but, no. You get the error `C2664: 'void swap_r(int &,int &)': cannot convert argument 1 from 'double' to 'int &'`

By the way, in class we got an "unresolved external" linker error when wrote

```
void swap_dr(double& a, double& b) {
    double temp;
    temp = a;
    a = b;
    b = temp;
}
```

and used it to try to swap a couple of doubles. I can't seem to replicate that error...it works just fine.

3. Write a program using a single file containing three `namespaces X, Y,` and `Z` so that the following `main()` works correctly:

```
int main() {
    X::var = 7;
    X::print(); // print X's var
    using namespace Y;
    var = 9;
    print(); // print Y's var
    {
        using Z::var;
        using Z::print;
        var = 11;
        print(); // print Z's var
    }
    print(); // print Y's var
    X::print(); // print X's var
}
```

Each namespace needs to define a variable called `var` and a function called `print()` that outputs the appropriate var using `cout`.

SOLN:

```
// Chapter 08, drill

///#include "../lib_files/std_lib_facilities.h"
#include <iostream>
using namespace std;

namespace X {
    int var;
    void print() {
        cout << var << endl;
    }
}

```

```
namespace Y {
    int var;
    void print() {
        cout << var << endl;
    }
}

namespace Z {
    int var;
    void print()    {
        cout << var << endl;
    }
}

int main() {
    X::var = 7;
    X::print();     // print X's var
    using namespace Y;
    var = 9;
    print();        // print y's var
    {
        using Z::var;
        using Z::print;
        var = 11;
        print();    // print Z's var
    }
    print();        // print Y's var
    X::print();     // print X's var
    cin.get();
}
```

In class we experimented with creating a variable `var` in the scope of `main()` and saw how that interfered with with the namespace `Y`.