

# Software Design and Analysis for Engineers

by

Dr. Lesley Shannon

Email: [lshannon@ensc.sfu.ca](mailto:lshannon@ensc.sfu.ca)

Course Website: <http://www.ensc.sfu.ca/~lshannon/courses/ensc251>



*Simon Fraser University*

Slide Set: 5

Date: September 28, 2015

# What we're learning in this Slide Set:

---

- Separate Compilation
- NameSpaces

# Textbook Chapters:

---

Relevant to this slide set:

- Chapter 12

Coming soon:

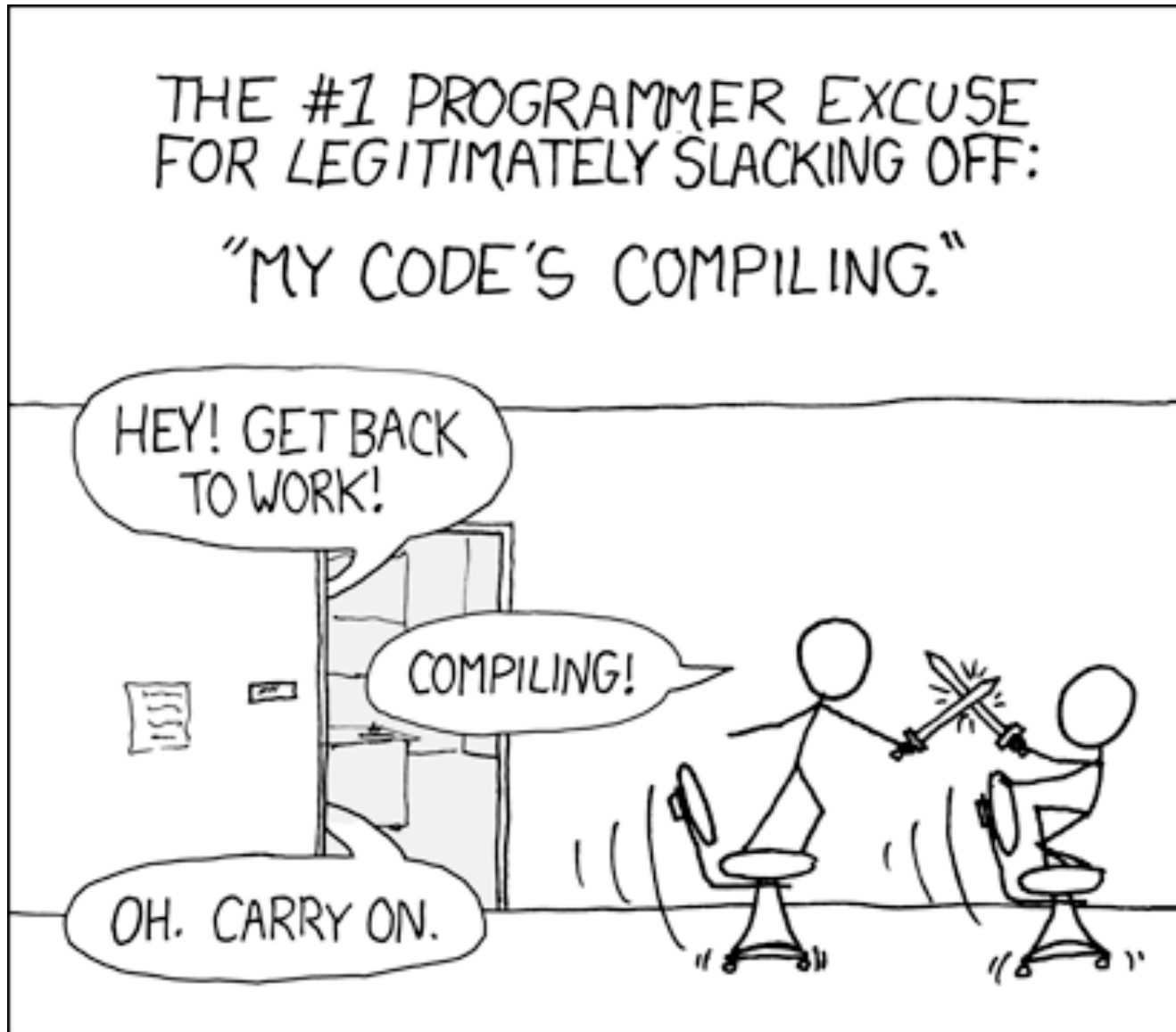
- Chapter 13

This lecture set will cover two topics that relate to the organization of your C++ program.

Learning to distribute your program across multiple files is extremely valuable for:

- readability,
- reuse and
- reducing the recompilation effort when you make changes.

A funny, yet true, statement on the state of affairs:



\*WARNING: Synthesis is *much* worse

<http://xkcd.com/303/>

## Separate Compilation:

When you divide your code across multiple files, each of these files is compiled separately and then linked together to create the final executable.

Ideally you would want to place the definition of a class and its associated function definitions in a single file that is separate from the program file that uses the class

This lets you build a library of classes that can be included in multiple different programs.

Furthermore, you can precompile each class library just once and then link it into each program that needs to use it:

Hint: this is just like the predefined libraries you use (iostream, cstdlib, etc).

## Separate Compilation:

You can even define the class in two separate files so that the standard interface (specification of what the class does) is separated from the implementation.

- We talked about using this to improve the encapsulation of your programs.
- It will also let you recompile only the implementation file.

This is good for ADTs:

- member variables are private,
- basic operations are public/friends/overloaded operators,
- their implementation is unavailable to the user

## Separate Compilation:

Note: The rules for how you set up your files for generic compilation is generic, however, the commands you use to compile and link files may vary amongst compilers



```

//Header file dtime.h: This is the INTERFACE for the class DigitalTime.
//Values of this type are times of day. The values are input and output in
//24-hour notation, as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.
#include <iostream>
using namespace std;
class DigitalTime
{
public:
    friend bool operator ==(const DigitalTime& time1, const DigitalTime& time2);
    //Returns true if time1 and time2 represent the same time;
    //otherwise, returns false.

    DigitalTime(int the_hour, int the_minute);
    //Precondition: 0 <= the_hour <= 23 and 0 <= the_minute <= 59.
    //Initializes the time value to the_hour and the_minute.

    DigitalTime( );
    //Initializes the time value to 0:00 (which is midnight).

    void advance(int minutes_added);
    //Precondition: The object has a time value.
    //Postcondition: The time has been changed to minutes_added minutes later.

    void advance(int hours_added, int minutes_added);
    //Precondition: The object has a time value.
    //Postcondition: The time value has been advanced
    //hours_added hours plus minutes_added minutes.

```

For the definition of the types `istream` and `ostream`, which are used as parameter types

## DigitalTime Interface Continued:

....

```
friend istream& operator >>(istream& ins, DigitalTime& the_object);  
//Overloads the >> operator for input values of type DigitalTime.  
//Precondition: If ins is a file input stream, then ins has already been  
//connected to a file.
```

```
friend ostream& operator <<(ostream& outs, const DigitalTime& the_object);  
//Overloads the << operator for output values of type DigitalTime.  
//Precondition: If outs is a file output stream, then outs has already been  
//connected to a file.
```

*private:*

```
int hour; }  
int minute; }  
};
```

*This is part of the implementation.*  
*It is not part of the interface.*  
*The word **private** indicates that*  
*this is not part of the public interface.*

Note: The Interface indicated it's name: `dttime.h`

The suffix `.h` indicates this is a header file (the interface file is always a header file).

Any program that wants to use the class `DigitalTime` must use the following include directive:

```
#include "dttime.h"
```

Any program that wants to use the class `DigitalTime` must use the following include directive:

```
#include "dtime.h"
```

Predefined header files are included using `<>` (angular brackets).

The `""` (double quotes) indicate that header file is provided/written by you.

This distinction tells the compiler where to search for the file:

- in a library of predefined libraries, or

- in your work space/current directory

Note: you may set up a repository of programmer-defined libraries as well, but that is outside of the scope of this class)

Warning: using the following include directive:

```
#include "ctime.h"
```

\*\*This will only let you compile the program, it won't let you run it.

Why do you think that is?

To run the program, you also need to compile and link in the implementation file.

Although it is not always required, proper code styles says that the implementation and header file have the same name with different suffixes.

As such, your `dtime.h` header file's implementation file is called `dtime.cpp`

Notes: -some compilers like "CPP" as a suffix;

-this style is extensible for C-programming and commonly use (interface in the header file and implementation in a ".c" file)

Let's look at the Implementation of DigitalAllTime

```

//Implementation file dtime.cpp (Your system may require some
//suffix other than .cpp): This is the IMPLEMENTATION of the ADT DigitalTime.
//The interface for the class DigitalTime is in the header file dtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "dtime.h"
using namespace std;

//These FUNCTION DECLARATIONS are for use in the definition of
//the overloaded input operator >>:

void read_hour(istream& ins, int& the_hour);
//Precondition: Next input in the stream ins is a time in 24-hour notation,
//like 9:45 or 14:45.
//Postcondition: the_hour has been set to the hour part of the time.
//The colon has been discarded and the next input to be read is the minute.

void read_minute(istream& ins, int& the_minute);
//Reads the minute from the stream ins after read_hour has read the hour.

int digit_to_int(char c);
//Precondition: c is one of the digits '0' through '9'.
//Returns the integer for the digit; for example, digit_to_int('3') returns 3.

bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
{
    return (time1.hour == time2.hour && time1.minute == time2.minute);
}

```

# Constructors:

```
//Uses iostream and cstdlib:
DigitalTime::DigitalTime(int the_hour, int the_minute)
{
    if (the_hour < 0 || the_hour > 23 || the_minute < 0 || the_minute > 59)
    {
        cout << "Illegal argument to DigitalTime constructor.";
        exit(1);
    }

    else
    {
        hour = the_hour;
        minute = the_minute;
    }
}
DigitalTime::DigitalTime( ) : hour(0), minute(0)
{
    //Body intentionally empty.
}
```

## Overloaded advance function:

```
void DigitalTime::advance(int minutes_added)
{
    int gross_minutes = minute + minutes_added;
    minute = gross_minutes % 60;

    int hour_adjustment = gross_minutes / 60;
    hour = (hour + hour_adjustment) % 24;
}
```

```
void DigitalTime::advance(int hours_added, int minutes_added)
{
    hour = (hour + hours_added) % 24;
    advance(minutes_added);
}
```

-one advances by minutes only

-the other advances by hours plus minutes (and calls the advance function that only advances by minutes)



## iostream and digit\_to\_int (helping function) functions:

```
//Uses iostream:  
ostream& operator <<(ostream& outs, const DigitalTime& the_object)  
{  
    outs << the_object.hour<< ':';  
    if (the_object.minute< 10)  
        outs << '0';  
    outs << the_object.minute;  
    return outs;  
}
```

```
//Uses iostream:  
istream& operator >>(istream& ins, DigitalTime& the_object)  
{  
    read_hour(ins, the_object.hour);  
    read_minute(ins, the_object.minute);  
    return ins;  
}
```

```
int digit_to_int(char c)  
{  
    return (static_cast <int>(c) - static_cast<int>('0'));  
}
```

read\_minute function (used in overloaded “>>” operator):

```
//Uses iostream, ctype, and cstdlib:
void read_minute(istream& ins, int& the_minute)
{
    char c1, c2;
    ins >> c1 >> c2;

    if (!(isdigit(c1) && isdigit(c2)))
    {
        cout<< "Error illegal input to read_minute\n";
        exit(1);
    }

    the_minute = (digit_to_int(c1) * 10) + digit_to_int(c2);

    if (the_minute < 0 || the_minute > 59)
    {
        cout<< "Error illegal input to read_minute\n";
        exit(1);
    }
}
```

## read\_hour function (used in overloaded ">>" operator):

```
//Uses iostream, ctype, and cstdlib:
void read_hour(istream& ins, int& the_hour)
{
    char c1, c2;
    ins >> c1 >> c2;
    if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )
    {
        cout<< "Error illegal input to read_hour\n";
        exit(1);
    }

    if (isdigit(c1) && c2 == ':')
    {
        the_hour = digit_to_int(c1);
    }
    else//(isdigit(c1) && isdigit(c2))
    {
        the_hour = (digit_to_int(c1) * 10) + digit_to_int(c2);
        ins >> c2;//discard ':'
        if (c2 != ':')
        {
            cout<< "Error illegal input to read_hour\n";
            exit(1);
        }
    }
    if (the_hour < 0 || the_hour > 23)
    {
        cout<< "Error illegal input to read_hour\n";
        exit(1);
    }
}
```

# Application File using DigitalTime

*//Application file timedemo.cpp (your system may require some suffix  
//other than .cpp): This program demonstrates use of the class DigitalTime.*

```
#include <iostream>
```

```
#include "dtime.h"
```

```
using namespace std;
```

```
int main( )
```

```
{
```

```
    DigitalTime clock, old_clock;
```

```
    cout<< "Enter the time in 24-hour notation: ";
```

```
    cin>> clock;
```

```
    old_clock = clock;
```

```
    clock.advance(15);
```

```
    if (clock == old_clock)
```

```
        cout << "Something is wrong.";
```

```
    cout << "You entered " << old_clock << endl;
```

```
    cout << "15 minutes later the time will be "
```

```
        << clock << endl;
```

```
    clock.advance(2, 15);
```

```
    cout << "2 hours and 15 minutes after that\n"
```

```
        << "the time will be "
```

```
        << clock << endl;
```

```
    return 0;
```

```
}
```

## Compilation process:

Both the implementation and program file (or application/driver file) must include the header file.

Step 1: Compile the Implementation file

Step 2: Compile the Program file

Step 3: Link the two together (using the linker)

Note: When you generate an executable, the linker will be called automatically. However, it is a separate executable and can be given separate commands from the compiler

Note: You don't need to compile the header file. Why? (Hint: What does the preprocessor do with the include command?)

## Summary:

Previously, you would have put all of your code in one file (and not needed the additional include statement for your class).

However, by separating the files:

- To reuse your DigitalTime class, you only need to include the header file in different programs (reducing the code you need to write).
- You don't need to recompile the program if you change the implementation file (you just need to relink the files).
- You don't need to recompile the implementation file, if you change the program (you just need to relink the files).

Additional hint: 1) For now, keep your header and implementation files in the same directory (you don't have to, but we won't cover it here).

2) You can put them both outside your current working directory, but be sure to include the directory path in the quotes.

## Final thoughts on code reuse:

While reuse saves you effort (designing code) and time (writing and compiling), it is also more reliable.

Why would this be?

You may also want to make libraries of commonly used function (sorted by type/operation) so that you can simply reuse them in all of your programs.

- This is extremely common in C programming as well.

## Another use of #ifndef:

Since you should now be looking to distribute your program across multiple files (separate header & implementation files for each class and potentially multiple program files), you may now find that more than one file has an include directive for your ADTs (i.e. multiple files have an `#include "dtime.h"`)

What if you include `dtime` directly and include another ADT that also includes `dtime`?

C++ does not allow you to define a class more than once (even if all the definitions are the same)

In larger code bases (10K+ lines), it becomes almost impossible to track whether you've include a class definition more than once (especially with derived classes).



## Another use of #ifndef:

To avoid this problem, C++ lets you use the preprocessor command `#ifndef` to indicate that if you have already included this section of code before, do not do it again.

The following directive “defines” `DTIME_H`:

```
#define DTIME_H
```

The preprocessor uses this command to add `DTIME_H` to its list of “defined constants” (typically it should have a value) as the preprocessor uses this command in a “search and replace” format in your file).

You can use this with the `#ifndef` processor as follows:

```
#ifndef DTIME_H  
#define DTIME_H  
<a class definition>  
#endif
```

## Another use of #ifndef:

```
#ifndef DTIME_H
#define DTIME_H
<a class definition>
#endif
```

If you use this in your header file, then it will ensure that even if your header file is included multiple time, the class will only be defined once since `DTIME_H` will be defined the first time the header file is included.

The next slide has an example.

Remember: there is an `#ifdef` preprocessor command (“if defined”) that complements this `#ifndef` command (“if NOT defined”).

- In this case we are using `#ifndef` because we do not want to include extra code; we use `#ifdef` to enable us to conditionally add extra code (e.g. debugging)

## New header file to avoid multiple class definitions

```
//Header file dtime.h: This is the INTERFACE for the class DigitalTime.  
//Values of this type are times of day. The values are input and output in  
//24-hour notation, as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.
```

```
#ifndef DTIME_H  
#define DTIME_H  
  
#include <iostream>  
using namespace std;  
class DigitalTime  
{
```

<The definition of the class DigitalTime is the same as in Display 12.1.>

```
};
```

```
#endif//DTIME_H
```

Note the identifier name (DTIME\_H) doesn't matter, but for readability, this is the style that many people use. Establish/Follow the code conventions of your team.

Going forward, I expect to see this in **every** header file you write.

## Namespaces:

Let you reuse the names of classes, functions, etc. by qualifying the names.

They divide your code into separate components (“spaces”) that let you reuse the same names with different meanings

-so that the names are “local” to that namespace

Why is this good? When working on a team with 100 programmers, there is a good possibility that two people may choose to name something the same way.

A namespace is a collection of name definitions (e.g. class definitions and variable declarations).

## Namespaces:

To date, we have been using the “standard” namespace (`std`).

This namespace includes all the names in the standard library files (e.g. `cstdlib`'s `iostream`).

All the code that you write is included in the ***global namespace*** by default (if you don't specify anything)

This means that you *could* define your own `cstdlib` in the global namespace (instead of using the predefined library if you wanted to).

## Namespaces:

You are also allowed to use more than one namespace at the same time.

However, since each name space might have the same functions (let's say that there are two namespaces `ns1` and `ns2` that have the same function, `my_function`), you will get an error if you simply try to call the function.

To clarify to the compiler which call is which, you can do the following:

```
{
    using namespace ns1;
    my_function( );
}
{
    using namespace ns2;
    my_function( );
}
```

Remember, the braces define a block of code and any definitions inside that block, are local to that block. As such each block has it's own scope and you can use these function calls in the same program.

## Namespaces:

If you place a using directive at the start of a file, then it applies to the entire file.

If you place it within enclosed braces, it only applies to the block.

The using directive should be placed at the start of a file or the start of a block (as it applies from the location at which it appears until the end of the block).

## Creating a Namespace:

To place code in a namespace, you simply:

```
namespace Name_Space_Name  
{  
    Some_Code  
}
```

Once you place some code definitions in a namespace, you make them available by using the the “using” directive:

```
using namespace Name_Space_Name;
```

You can have as many of these namespace grouping as you want for a single namespace distributed throughout your code.

The key is that every name defined in a name space is available in that namespace, but it can also be made available to code outside the namespace (let’s see an example...)



# Function Declarations:

```
#include <iostream>
using namespace std;

namespace savitch1
{
    void greeting( );
}

namespace savitch2
{
    void greeting( );
}

void big_greeting( );
```

# main:

```
int main( )
```

```
{
```

```
{
```

```
using namespace savitch2;
```

```
greeting( );
```

```
}
```

```
{
```

```
using namespace savitch1;
```

```
greeting( );
```

```
}
```

```
big_greeting( );
```

```
return 0;
```

```
}
```

*Names in this block use definitions in namespaces **savitch2**, **std**, and the global namespace.*

*Names in this block use definitions in namespaces **savitch1**, **std**, and the global namespace.*

*Names out here use only definitions in namespace **std** and the global namespace.*

# Function Definitions:

```
namespace savitch1
{
    void greeting( )
    {
        cout << "Hello from namespace savitch1.\n";
    }
}
```

```
namespace savitch2
{
    void greeting( )
    {
        cout<< "Greetings from namespace savitch2.\n";
    }
}
```

```
void big_greeting( )
{
    cout<< "A Big Global Hello!\n";
}
```

## Qualifying Names:

What if you want to use `fun1 ( )` from `ns1` and `fun2 ( )` from `ns2`, but both `ns1` and `ns2` define a function `my_function` (to keep it simple assume no arguments for any of the functions).

The following would provide conflicting definitions for `my_function`

```
using namespace ns1;  
using namespace ns2;
```

Instead you want to indicate the specific functions you want to use from each namespace with ***using declarations***:

```
using ns1::fun1;  
using ns2::fun2;
```

Generic form: `using Name_Space::One_Name`

This will only let you use `One_Name` from that namespace and nothing else.

## Qualifying Names:

What if you want to use `fun1 ( )` from `ns1` or some other function/variable once or a few times:

You can then name the function or other item (e.g. variable/parameter type) using the scope operator as follows:

```
ns1::fun1( );  
int get_number(std::istream input_stream)
```

In the function `get_number`, the `input_stream` is of type `istream` as defined in the `std` namespace.

If this use of the type name `istream` is the only name you need from the namespace (or if you qualify all of the names you need from the namespace using `std::`), then you do not need:

```
using namespace std;
```

## Subtle differences:

There are two differences between a using declaration and a using directive:

`using std::cout;`      versus      `using namespace std;`

1. A using declaration makes only one name in the namespace available while a using directive makes all of the names in the namespace available
2. ***A using declaring introduces a name (like cout) into your code so that no other use of the name can be made.*** However, a using directive only ***potentially*** introduces the names in that namespace.

## Subtle differences:

Recall point 2:

***A using declaring introduces a name (like cout) into your code so that no other use of the name can be made.*** However, a using directive only ***potentially*** introduces the names in that namespace.

Assuming both namespaces have the function `my_function`, here's an example:

This is fine: `using namespace ns1;`  
`using namespace ns2;`

This is illegal: `using ns1::my_function;`  
`using ns2::my_function;`

Why?

## Compilation Unit:

Is an implementation file (such as a class implementation file) along with all of the necessary header files that need to be included.

Every compilation file has an unnamed namespace

A namespace grouping for the unnamed namespace is written like any other, except without a name

```
namespace
{
    void sample_function( )
    .
    .
    .
} //unnamed namespace
```

All names defined in the unnamed namespace are local to the compilation unit.

- This means they can be reused in other compilation units.



## Compilation Unit:

When you create an ADT any helper functions (i.e. functions that are not members or friend functions of the class) should go in the unnamed namespace.

- Why?

Think of `digit_to_int`, `read_hour`, and `read_minute`.

- We want these functions to be local to the class `DigitalTime`, but your application program cannot reuse these names
- This does not really hide these functions from the user (or hides them poorly depending on your perspective).

Putting them in the unnamed namespace, localizes their definition to the compilation unit and makes their names reusable in other programs.

Here's an example


## dtime.h:

```
//Header file dtime.h: This is the interface for the class DigitalTime.  
//Values of this type are times of day. The values are input and output in  
//24-hour notation, as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.
```

```
#ifndef DTIME_H  
#define DTIME_H
```

```
#include <iostream>  
using namespace std;
```

*One grouping for the namespace dtimesavitch.  
Another grouping for the namespace dtimesavitch  
is in the implementation file dtime.cpp.*



```
namespace dtimesavitch  
{
```

```
    class DigitalTime  
    {
```

```
        <The definition of the class DigitalTime is the same as in Display 12.1.>  
        };
```

```
}//end dtimesavitch
```

```
#endif //DTIME_H
```

## mtime.cpp:

```
//Implementation file mtime.cpp (your system may require some
//suffix other than .cpp): This is the IMPLEMENTATION of the ADT DigitalTime.
//The interface for the class DigitalTime is in the header file mtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "mtime.h"
using namespace std;

namespace
{
    //These function declarations are for use in the definition of
    //the overloaded input operator >>:
    void read_hour(istream& ins, int& the_hour);
    //Precondition: Next input in the stream ins is a time in 24-hour notation,
    //like 9:45 or 14:45.
    //Postcondition: the_hour has been set to the hour part of the time.
    //The colon has been discarded and the next input to be read is the minute.
    void read_minute(istream& ins, int& the_minute);
    //Reads the minute from the stream ins after read_hour has read the hour.
    int digit_to_int(char c);
    //Precondition: c is one of the digits '0' through '9'.
    //Returns the integer for the digit; for example, digit_to_int('3')
    //returns 3.
} //unnamed namespace
```

One grouping for the unnamed namespace



599564 2014/10/05 173.180.246.121

dttime.cpp (part 2):

*One grouping for the namespace `dtimesavitch`.  
Another grouping is in the file `dtime.h`.*

```
namespace dtimesavitch  
{
```

```
    bool operator ==(const DigitalTime& time1, const DigitalTime& time2)  
<The rest of the definition of == is the same as in Display 12.2.>
```

```
    DigitalTime::DigitalTime( )  
<The rest of the definition of this constructor is the same as in Display 12.2.>
```

```
    DigitalTime::DigitalTime(int the_hour, int the_minute)  
<The rest of the definition of this constructor is the same as in Display 12.2.>
```

```
    void DigitalTime::advance(int minutes_added)  
<The rest of the definition of this advance function is the same as in Display 12.2.>
```

```
    void DigitalTime::advance(int hours_added, int minutes_added)  
<The rest of the definition of this advance function is the same as in Display 12.2.>
```

## dttime.cpp (part 3):

```
ostream& operator <<(ostream& outs, const DigitalTime& the_object)  
<The rest of the definition of << is the same as in Display 12.2.>
```

```
//Uses iostream and functions in the unnamed namespace:  
istream& operator >>(istream& ins, DigitalTime& the_object)  
{  
    read_hour(ins, the_object.hour);  
    read_minute(ins, the_object.minute);  
    return ins;  
}  
} //dtimesavitch
```

*Functions defined in the unnamed namespace are local to this compilation unit (this file and included files). They can be used anywhere in this file, but have no meaning outside this compilation unit.*

*namespace* ← *Another grouping for the unnamed namespace.*

```
{  
    int digit_to_int(char c)  
<The rest of the definition of digit_to_int is the same as in Display 12.2.>  
  
    void read_minute(istream& ins, int& the_minute)  
<The rest of the definition of read_minute is the same as in Display 12.2.>  
  
    void read_hour(istream& ins, int& the_hour)  
<The rest of the definition of read_hour is the same as in Display 12.2.>  
  
} //unnamed namespace
```

## Application program:

```
//This is the application file: timedemo.cpp. This program  
//demonstrates hiding the helping functions in an unnamed namespace.
```

```
#include <iostream>  
#include "dtime.h"
```

```
void read_hour(int& the_hour);
```

*If you place the using directives here, then the program behavior will be the same.*

```
int main( )  
{
```

```
    using namespace std;
```

```
    using namespace dtimesavitch;
```

```
    int the_hour;
```

```
    read_hour(the_hour);
```

*This is a different function read\_hour than the one in the implementation file dtime.cpp (shown in Display 12.7).*

```
    DigitalTime clock(the_hour, 0), old_clock;
```

```
    old_clock = clock;
```

```
    clock.advance(15);
```

```
    if (clock == old_clock)
```

```
        cout << "Something is wrong.";
```

```
    cout << "You entered " << old_clock << endl;
```

```
    cout << "15 minutes later, the time will be "
```

```
        << clock; << endl;
```

## Application program (part 2):

```
clock.advance(2, 15);
cout << "2 hours and 15 minutes after that\n"
      << "the time will be "
      << clock; << endl;

return 0;
}
void read_hour(int& the_hour)
{
    using namespace std;

    cout << "Let's play a time game.\n"
          << "Let's pretend the hour has just changed.\n"
          << "You may write midnight as either 0 or 24,\n"
          << "but I will always write it as 0.\n"
          << "Enter the hour as a number (0 to 24): ";
    cin >> the_hour;
    if (the_hour == 24)
        the_hour = 0;
}
```

NOTE: This function `read_hour` has the same name as the `DigitalTime` function, but since that helper function is part of the ADT's compilation unit, there is no overlap in the declaration.

## Last thoughts on namespaces:

-It's a good idea to create namespaces that you can be reasonably confident will be unique.

- Use your last name or some other unique string to identify all namespaces that you create (and minimize the chances that somebody else will create the same namespace as you).

## Note:

The global and unnamed namespace are not the same.

- Both can be accessed without a qualifier
- However, names in the global namespace have global scope (all program files)
- Names in the unnamed namespace are local to a compilation unit



# Review Questions for Slide Set 5

- What is the compilation process when you divide your code across multiple files?
- How do you include a custom library (like the LinkedList one you created in Assignment 2) in an application program?
- What's the difference in syntax between including predefined header files versus programmer defined header file in a program? Why is this distinction important to the compiler?
- Why does adding the include statement let your program compile but not let it run?
- What is the difference between the linker and compiler?
- What does the preprocessor do with the include command?

# Review Questions for Slide Set 5

- What are the advantages of having separate compilations for library files from application files?
- Do header and cpp files have to be in the same library?
- What are `#ifndef` and `#ifdef`? Are these preprocessor, compiler or linker commands?
- What is the syntax for `#ifndef` and `#ifdef`?
- How would you use `#ifdef`? Give an example.
- How would you use `#ifndef`? Give an example.
- What is a namespace?
- What is the global namespace?
- Can you use more than one namespace at once?

# Review Questions for Slide Set 5

- Can you have the same function name in more than one namespace? Why or why not?
- How should you use the “using” directive?
- How do you make something declared in a namespace available to code outside the namespace?
- What does the phrase “qualifying names” refer to?
- What’s the difference between a “using declaration” and a “using directive”?
- What is the unnamed namespace?
- Why would you put helper functions in the unnamed namespace?
- How are the global and unnamed namespace the same? How are they different?