**SEPARATE COMPILATION**

Program parts

* Kept in separate files
* Compiled separately to .o files
* Linked together before program runs
* include h files to your program

Class definitions

* Separate from “using” programs
* Build library of classes
  + Re-used by many different programs
  + Just like predefined libraries

**CLASS SEPARATION**

Class independence

* Separate class definition/specification
  + Called “interface”
* Separate class implementation
* Place in two files

If implementation changes 🡪 only that file need be changed

* Class specification need not change
* “User programs need not change

CHECK 11.01 (header/interface), 11.02 (implementation) and 11.03 (driver code)

You tell to compiler that where the header files and where the libraries.

**ENCAPSULATION REVIEWED**

* Encapsulation principle:
  + Separate how class is used by programmer from details of class’s implementation
* “Complete” separation
  + Change to implementation 🡪 NO impact on any other programs
* Basic OOP principle

**Encapsulation Rules**

Rules to ensure separation:

1. All member variables should be private
   1. EXCEPTION: You may use “const doube PI” kind of member variables in your public area of your class.
   2. Determine if the variables need setters and getters.
2. Basic class operations should be:
   1. Public member functions
   2. Friend or ordinary functions
   3. Overloaded operators
   4. Group class definition and prototypes together called “interface” for class
      1. Interface will be available to your customers
      2. All the implementations and rest of the stuff will be private to you
3. Make class implementation unavailable to users of class

When you make interface part different from the implementation part and put in to different files, then it is better encapsulation.

**More Class Separation**

Interface File

* Contains class definition with function and operator declarations/prototypes
* Users “see” this
* Separate compilation unit

Implementation File

* Contains member function definitions
* Separate compilation unit

include

* Class interface always in header file, use .h naming convention
  + #include “myclass.h”
  + class name is also myclass (just for convention)
  + implementation: myclass.cpp (must include class’s header file)
  + interface/header: myclass.h
* Quotes indicate you wrote header
  + Find it in your working directory
* Recall library includes, e.g., <iostream>
  + <> indicate predefined library header file
  + Find it in library directory

In general .cpp files contain executable code (e.g., function definitions, including main())

Extensions (.cpp, .h, etc.) is for you. There is no problem without them.

**Class Files**

Class header file #included by

* Implementation file
* Program file
  + Often called “application file” or “driver file”

Organization of files is system dependent

* Typical IDE has “project” or “workspace”
  + Implementation files “combined” here
  + Header files still “#included”

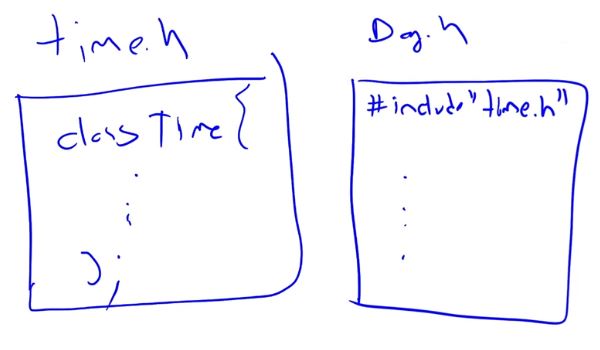
**Multiple Compiles of Header Files**

Header files

* Typically included multiple times
  + e.g., class interface included by class implementation and program file
* Must only be compiled once!
* No guarantee “which #include” in which file, compiler might see first

Use preprocessor

* Tell compiler to include header only once



Your customer shouldn’t include time.h more than once to their program. Compiler says class time is defined 2 times in that situation.

If your customer include time.h and Day.h to their program, he/she end up including time.h 2 times because Day.h already includes time.h.

We get help from the preprocessor. We asked to preprocessor “Did you compile time.h before?”. If it says yes, then I say I am not going to compile it anymore.

CHECK 11.04, 11-05dtime.h, 11-06dtime.cpp, 11-07timedemo.cpp to learn how to do it

Since I have 2 cpp files, you need to call the compiler 2 times. Both of them (dtime.cpp and timedemo.cpp) include dtime.h. That doesn’t mean that dtime.h will be compiled once bc when I compile them separately (for dtime.cpp and timedemo.cpp), I am running 2 compilation units. Each time I am using the compiler, I am including dtime.h.

**Compilation Unit**

Every time you make a compile call, you are creating a new compilation unit.

Every time you make a new compilation unit, preprocessor runs again and it lets the dtime.h to be compiled again. In that case, with 2 different compilation units, ifndef mechanism doesn’t kick in bc defines are only valid for single compilation unit.

dtime.h could be in both compilation units(for dtime.cpp and timedemo.cpp).

**Using #ifndef**

* Header file structure:

#ifndef FNAME\_H

#define FNAME\_H

…//Contents of header file

…

#endif

* FNAME typically name of file for consistency, readability
* This syntax avoids multiple definitions of header file

**Other Library Files**

Libraries not just for classes

Related functions

* Prototypes 🡪 header file
* Definitions 🡪 implementation file

Other type definitions

* structs, simple typedefs 🡪 header file
* Constant declarations 🡪 header file

**NAMESPACES**

* Namespace defined:
  + A collection of name definitions
    - Class definitions
    - Variable declarations
* Programs use many classes, functions
  + Commonly have same names
  + Namespaces deal with this
  + Can be “on” or “off”
    - If names might conflict 🡪 turn off

You say that “I have a namespace, its name is std and I am gonna put these classes into this namespace: string class, vector class, linklist class, etc.”. Otherwise compiler won’t know what you are using.

So you make a new namespace and you put class definitions, function declarations, variable declarations in that namespace.

When the time comes to use those classes or functions, you say that “I am gonna use this function inside this namespace.“.

Everything has to be part of a namespace in C++.

std namespace is std namespace, you should not change it.

We have never said that I am gonna add this class to this namespace. You should have your own namespace but where is my namespace? We will talk about that.

namespaces are kind of ways of arranging your classes and functions in a nice and tidy way.

**using Directive**

* using namespace std;
  + Makes all definitions in std namespace available
* Why might you NOT want this?
  + Can make cout, cin have non-standard meaning (You shouldn’t do this)
    - Perhaps a need to redefine cout, cin
  + Can redefine any others.

**Namespace std**

* Contains all names defined in many standard library
* Example:
  + #include <iostream>
    - Places all name definitions (cin, cout, etc.) into std namespace
    - Program doesn’t know names
    - Must specify this namespace for program to access names

When you say “using namespace std”, you are saying “From now on, if I talk about cin, cout, string, etc., I am talking about the cout, cin object etc. that is defined inside this namespace std.

**Global Namespace**

In C++ all the definitions go to some kind of a namespace.

* If you define your namespace and tell that “This is my namespace and I put this class definition into this namespace” this is fine.
* If you don’t do that, if you don’t specify where you are putting your classes etc., everything you declare will go to this global namespace.
* All code goes in some namespace
* Unless specified 🡪 global namespace
  + Global namespace doesn’t have a name
  + No need for using directive
  + Global namespace always available
  + Implied “automatic” using directive (you don’t say “using namespace global;”)

**Multiple Names**

Multiple namespaces (e.g., global, and std typically used)

What if name defined in both? What if you do “int cout;” while you are using namespace std (If you are not using namespace std there is no problem.)?

* Error
* Can still use both namespaces
* Must specify which namespace used at what time (You can do std::cout without using namespace std and define “int cout;” at the same time)

If you didn’t use “using namespace std;”, you can do :

int cout = 3;

std::cout << cout;

Of course you shouldn’t do this kind of stuffs.

Namespaces are part of the encapsulation.

Your related classes and definitions are part of the same structure: namespace.

Recall: we didn’t like putting global functions inside our classes. Then we put them in our class but we made them static so that they can’t do anything with non-static members or functions. Now we can take these functions out and put them in special namespace. With this they will be kind of global but not exactly because they will be valid only within that namespace.

**Specifying Namespaces**

Given namespaces NS1, NS2

* Both have void function myFunction() defined differently

{

using namespace NS1;

myFunction();

//myFunction will come either from NS1 or global namespace

}

{

using namespace NS2;

myFunction();

//myFunction will come either from NS2 or global namespace

}

* “using” directive has block-scope, it doesn’t have to be at the top (at the global scope)
* Inside first block I am using NS1 namespace, inside second block I am using NS2 namespace.
* You can have functions with same signature in 2 different namespaces.

*NAMESPACES ARE GENERALIZATION OF ENCAPSULATION IDEA.*

We said that these classes, these functions, these variables are related. Then we put them in the same capsule and we say them namespace.

**CREATING A NAMESPACE**

* Use namespace grouping:

namespace Name\_Space\_Name

{

Some\_Code

}

* Places all names defined in Some\_Code into namespace Name\_Space\_Name
* Can then be made available:

using namespace Name\_Space\_Name;

Try to give your namespace a unique name (your lastname, company name, etc.).

CHECK 11-08application.cpp, 11-09dtime.cpp, 11-10dtime.h

Why don’t we like the global functions? 🡪 We don’t know who is going to call them.

Why don’t we like the global variables? 🡪 We don’t know who is going to modify them.

If you make your global functions and global variables part of the namespaces, you are kind of restricting our customers saying that “If you are going to use these, then you have to use this namespace.”. This is kind of a solution to global functions and global variables.

All kind of C++ libraries are all defined in the namespace std.

std namespace is not supposed to be modified.

**Separate interface from the implementation**

Function declaration (in header file):

namespace Space1

{

void greeting();

}

Function definition (in implementation file):

namespace Space1

{

void greeting()

{

cout << “Hello from namespace Space1.\n”;

}

}

You can define part of the namespace in some place and other part of the namespace somewhere else.

**using Declarations**

Can specify individual names from namespace

Consider NS1 and NS2 namespaces both have functions fun1() and fun2().

* Declaration syntax: using NAMESPACE::ONENAME;
  + Makes ONE name in NAMESPACE available
  + Introduces names so no other uses of name are allowed
* Specify which name from each:

using NS1::fun1;

using NS2::fun2;

* This means I don’t want to use all the definitions inside NS1 or NS2. I just need to use fun1 or fun2.

using std::cout; 🡪 using DECLARATION

Recall: using DIRECTIVE makes all names in namespace available (using namespace std;).

**Qualifying Names**

std::cout << “Hello”; //without any “using” things

* Use only intend one use (or few)
* Especially useful for parameters:

int getInput(std::istream inputStream);

* + - Parameter found in istream’s std namespace
    - Eliminates need for using directive or declaration

**Naming namespaces**

* Must be unique to reduce chance of other namespaces with same name.
* Without distinct names, multiple definitions of same name in same scope results in error.

CHECK 11.11application.cpp, 11.12dtime.cpp, 11.13dtime.h, 11.14

If you use main function like this:

namespace Gursimsir{

int main{

…

}

}

This will compile but not link bc linker needs a main function that lives in the global namespace.

Don’t put your main inside any namespaces other than the global namespace.

**Unnamed Namespaces**

Special as global namespace

Compilation unit: Compilation unit is a file along with all files included in file. So your file + all the files you included to your file form the compilation unit. Whatever the compiler compiles for your code is called compilation unit.

Every compilation unit has exactly 1 unnamed namespace.

* Written same way, but with no name
* All names are then local to compilation unit. Since it doesn’t have a name, you cannot use that namespace outside of that compilation unit.

Use unnamed namespace to keep things “local”. Whatever you defined in unnamed namespace, stays inside that compilation unit. Nobody other than the compilation unit elements can use those definitions. Scope of unnamed namespace is compilation unit.

You can use all the functions in unnamed namespace in the current compilation unit.

**GLOBAL vs. UNNAMED NAMESPACES**

* Not same
* Global namespace:
  + No namespace grouping at all
  + Available for all the compilation units
  + Global scope
  + There is only 1 global namespace for your all program (doesn’t matter how many cpp files or compilation units you have)
* Unnamed namespace:
  + Has namespace grouping, just no name
  + Available for only THAT compilation unit
  + Local scope
  + Number of unnamed namespaces is the same as number of compilation units.

Other than the main, we don’t want anything inside global namespace. So always make your classes, functions, etc. part of something.

CHECK 11.15dtime.cpp, 11.16dtime.h, 11.17timedemo.cpp

**NESTED NAMESPACES**

namespace S1

{

namespace S2

{

void sample()

{

…

}

}

Now you have to qualify names twice if you want to use sample function without any using directives or using declarations:

S1::S2::sample();

**HIDING HELPING FUNCTIONS**

* Recall helping function:
  + Low-level utility
  + Not for public use, we should hide them
* 2 ways to hide:
  1. Make private (static) member function
     + If function naturally takes calling object
  2. Place in class implementation’s (compilation unit) unnamed namespace
     + If function needs no calling object
     + Makes cleaner code (no qualifiers)

You can give any name to functions inside your unnamed namespace bc they cannot be confused by somebody else’s function.

If I say g++ compiler “Compile a.txt (text file)” it will compile it. So doesn’t matter if a file has .h or .cpp extension.