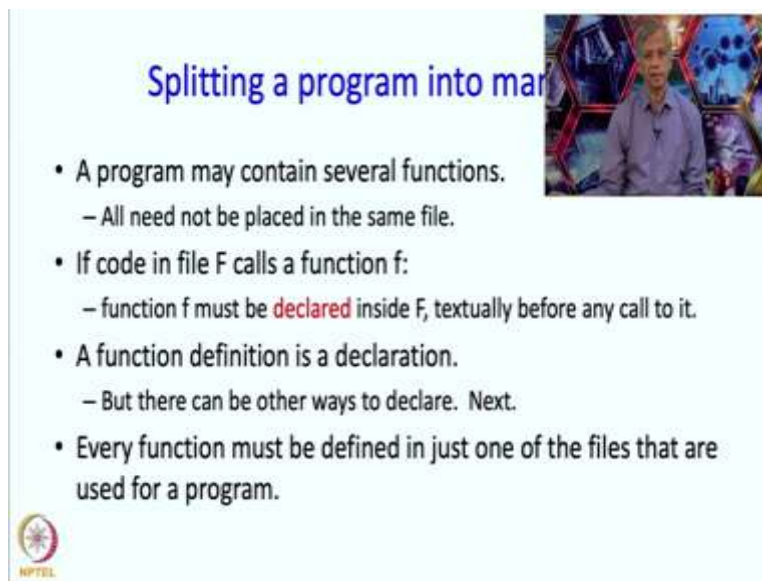


An Introduction to Programming through C++
Prof. Abhiram G. Ranade
Department of computer Science and Engineering, IIT Bombay
Lecture 13 Part 2
Program Organization and Function
Splitting into files

Welcome back, in the previous segment we discussed that the main program is a function and we motivated the need for splitting the program into function as well as into several files. In the next segment we are going to in the rest of this lecture sequence actually, we are going to see how to actually split programs into files and that is what is going to happen in this segment and then in subsequent segments we will see things like how your program uses code written by others and features that C plus-plus provides for that and also finally we will see how you can use C++ without simple CPP.



Ok so how do you split program split, split a program into many files?

(Refer Slide Time: 01:16)



Splitting a program into many files

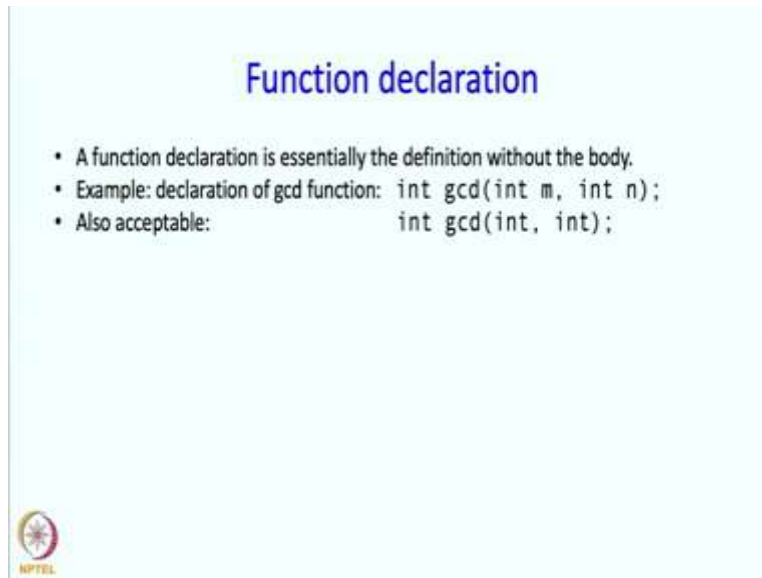
- A program may contain several functions.
 - All need not be placed in the same file.
- If code in file F calls a function f:
 - function f must be **declared** inside F, textually before any call to it.
- A function definition is a declaration.
 - But there can be other ways to declare. Next.
- Every function must be defined in just one of the files that are used for a program.



Okay so a program may contain several functions and all of them need not be placed in the same file, so if a code in file capital F calls a function little f then the function f must me declared inside F, textually before any call to it. Now a function definition that we have been talking is a declaration but that is not the only way to declare a function.

So, the definition could be in another file and so long as there is a declaration in the file capital F which contains calls to this function that is enough okay, so every function must be defined in just one of the files that are used for a program. But it must be declared in all the files that make a call to that function.

(Refer Slide Time: 2:26)




The slide is titled "Function declaration" in blue text. It contains three bullet points: "A function declaration is essentially the definition without the body.", "Example: declaration of gcd function: `int gcd(int m, int n);`", and "Also acceptable: `int gcd(int, int);`". In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

So what is a function declaration? Well it is essentially the definition without the body, so declaration of the GCD function looks like `int gcd(int m, int n)` so after that you could have given the body but that is being omitted in the declaration okay. Also acceptable is `int gcd(int, int)`. Now you may notice that this is really like declaration of a variable, so I might have written `int X` okay so it looks like a declaration of a variable but it is the parentheses are saying it is not a variable but because of those parentheses following the `gcd` the compiler can tell that oh this is a very this is a function that we are talking about. So, basically that is what a declaration tells the compiler. It tells that look `gcd` is a function and should the name `gcd` appears, appear later on in this file, think of it as a function which takes 2 arguments and the arguments had better be of the specified type.

(Refer Slide Time: 03:43)

Function declaration

- A function declaration is essentially the definition without the body.
- Example: declaration of gcd function: `int gcd(int m, int n);`
- Also acceptable: `int gcd(int, int);`
- The declaration tells the compiler that if the name gcd appears later, it will be a function and take 2 arguments of the specified type.
 - Compiler can now check if the name is used correctly in the rest of the file.
- If a file contains a call to a function but does not contain its definition:




So this way the compiler checks that subsequent uses of the name gcd adhere to what you have said about gcd, again the general strategy over here is to say what gcd is and then check whether you are using gcd in the manner that you have just described. This is also because we are worried about making mistakes. So the compiler tries to prevent, tries to help you out and tries to see if you are possibly making a mistake and therefore it needs to know what every name is over whether it is a function, it is a variable and therefore, the declaration has to be given preceding the use of that name.

(Refer Slide Time: 04:42)

Function declaration

- A function declaration is essentially the definition without the body.
- Example: declaration of gcd function: `int gcd(int m, int n);`
- Also acceptable: `int gcd(int, int);`
- The declaration tells the compiler that if the name gcd appears later, it will be a function and take 2 arguments of the specified type.
 - Compiler can now check if the name is used correctly in the rest of the file.
- If a file contains a call to a function but does not contain its definition:
 - It can only be partially compiled into an **object module**.
 - To get an executable program, all the object modules containing all called functions must be **linked** together.
- Other names for "declaration": **Signature, Prototype**
- Example next.



Now if a file contains a call to a function but does not contain its definition, the compiler will assume that it will appear elsewhere. In fact, that is the whole point of putting in a declaration, so putting in that `int gcd(int m, int n)` without the body okay. But the compiler will not be able to compile it as a main program okay as in fact unless the main program is there it will not be able to compile but further more if some function body is missing again it will not be able to compile. What it can compile your file into is what is called an object module and we will see that.


To get an executable program all the object modules containing the call function must be linked together only then you can produce an executable program. We said this long ago when we were talking about how the hardware works but now you will see a little bit a few additional details okay. The function declaration that we have talked about so this line, this portion, this or this is called a declaration but it is also called the signature of the function or the prototype of the function. So some examples, so here is an example of a program which is split over multiple files.

(Refer Slide Time: 6:17)

Example of code split over multiple files

- File gcd.cpp
 - `int gcd(int m, int n){ ... }`
- File lcm.cpp
 - `int gcd(int, int);`
 - `int lcm(int m, int n){`
`return m*n/gcd(m,n);`
`}`
- File main.cpp
 - `int lcm(int, int);`
 - `int main(){`
`cout << lcm(36,24) << endl;`
`}`

- Function definitions, function declarations
- Each file has declarations of called functions.
- To compile and link all files together:
`s++ main.cpp lcm.cpp gcd.cpp`
- To compile each file separately:
`s++ -c lcm.cpp`
- `-c`: produce lcm.o (object module).
- To get executable from Object modules:
`s++ main.o lcm.o gcd.o`
- What you typically do: `s++ pgm.cpp m1.o m2.o`
- Compile your program `pgm.cpp`
- Link it to object modules developed by others



So you can have a program consisting of a function in gcd with the body also given and perhaps you can put it in the file gcd.cpp okay, you can also have a second file lcm.cpp a third file main.cpp okay. So what is in these files? So the green portions are function definitions and the red portions are the function declarations and each file you will see has a declaration, declarations of the called function. So, in this file there is call to gcd and the definition the function is not defined over here, but the function is declared in this file. So this is the declaration.

So the compiler can compile this file at least partially into an object module. Then in this file main.cpp, lcm is being called and it is not defined over here but there is a declaration saying that oh it contains it has 2 arguments and in fact compiler can check that indeed 2 arguments of type integers have been given. Now, if you have these files and want to compile and link them together okay you would issue the command `s++ main.cpp lcm.cpp gcd.cpp` if you are operating under Unix and these files were in your current directory. If you are operating, if you are using the full IDE then you would have defined something called a project in which these files would have to be present. But the contents of the files would have to be exactly these, that if file a uses a function makes a call to a function and if the function is not defined there then there has to be a declaration of it above the use or the call to that function.

If you want to compile each file separately, this is what you do you write `g++` but you add this compiler option `-c` which essentially says compile only or do not link `lcm.cpp`. So this produces `lcm.o` which is an object module which you can think of as something which is partially compiled. And if you have all this object modules then you can compile them together by writing `g++` again, `g++` does everything as you can see the compiler does everything. You can write `main.o lcm.o gcd.o`. So this will actually produce your executable.

Now, what you typically do is not quite this, what you typically do is that you write a program say in a file called `pgm.cpp`. On Unix you compile it but the `g++` command itself supplies all the additional modules that you want, so if you are using `sqrt` that module is supplied, if you are using `simple CPP` the modules required for `simple CPP` are supplied okay so this compilation is really quite similar to this compilation or even this compilation even though you are using only a single file.

So yeah, so this `g++` is compiling your program `pgm.cpp` but you are also linking it to object modules which are developed by others. So where are they? Well they are in some places that this `g++` knows and some of them might be present as libraries okay, so its not only object modules that you can compile together you can also compile together libraries. So the libraries might also get listed over here. So we are going to omit that detail because `g++` places those libraries for you, but they are sort of like object modules, they also contain code for functions. Okay so for all this to work correctly your program must have the right declaration.

(Refer Slide Time: 11:12)

Example of code split over multiple files

- File gcd.cpp

```
int gcd(int m, int n){ ... }
```
- File lcm.cpp

```
int gcd(int, int);
int lcm(int m, int n){
    return m*n/gcd(m,n);
}
```
- File main.cpp


```
int lcm(int, int);
int main(){
    cout << lcm(36,24) << endl;
}
```

- Function definitions, function declarations
- Each file has declarations of called functions.
- To compile and link all files together:
s++ main.cpp lcm.cpp gcd.cpp
- To compile each file separately:
s++ -c lcm.cpp
- -c : produce lcm.o (object module).
- To get executable from Object modules:
s++ main.o lcm.o gcd.o

What you typically do: s++ pgm.cpp m1.o m2.o

- Compile your program pgm.cpp
- Link it to object modules developed by others

Your pgm.cpp must have the right declarations...



So if this is your program then it will better have a declaration of the function that you are going to use. Now you may say that look I used square root, I used right, I used turtleSim, I used in it canvas but did not put declarations to all those in my in the program that I wrote okay. So, there is a different mechanism for putting in the declarations which now I am going to tell about.

(Refer Slide Time: 11:53)

Header files

- Tedious to remember what declaration to include in each file.
- Instead, we can put all declarations into a header file, and "include" the header file into every file.
- Header file gcdlcm.h


```
int gcd(int, int);
int lcm(int, int);
```
- The directive "#include filename"
 - gets replaced by the content of the named file.
 - It is acceptable if we declare functions that do not get used.
 - It is acceptable if we have both a declaration and then the definition of a function in the same file.

- File gcd.cpp

```
#include "gcdlcm.h"
int gcd(int m, int n){ ...
}
```
- File lcm.cpp

```
#include "gcdlcm.h"
int lcm(int m, int n){ ...
}
```
- File main.cpp

```
#include <simplecpp>
#include "gcdlcm.h"
int main(){ ...}
```



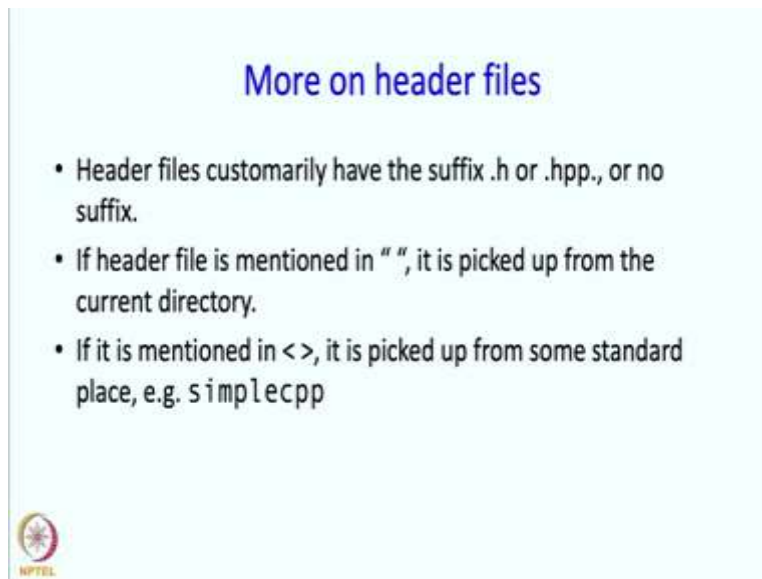
So this mechanism is called header files. So it is tedious to remember what declaration to include in each file, so it is tedious to remember what declaration to include in each file. So what do we do instead? We put all the declaration into a file called the header file and include the header file into every file that we use. So here for example is the header file gcdlcm.h so well I just made up a name okay. So this file contains int GCD int LCM so it contains the declaration of GCD and LCM.

And now the file gcd.cpp does not actually contain the declaration but it says include this file, so all those declarations would go in okay, LCM similarly just includes this file and main also include that file okay, so basically I do not, I am not worrying about oh include the declaration for this function, I am not going to worry about this function, that function. I am just going to say look whatever functions I need those and maybe be more than those are present in this header file and I just going to include my header file and that will automatically bring in these lines.

So the directive include file name just copies the contents of that file into your file okay, so it gets replaced by the contents of the name file. And it is acceptable if we declare functions that do not get used okay, so for example in this if we include gcdlcm.h this file itself directly does not need the declaration for GCD but it is ok to have it okay. It just says that GCD is a function whether I am using it or is not important okay.


It is acceptable if have both declaration and then the definition, so that is going to happen over here. So this include is going to be replaced by `int gcd(int, int)` which is just a declaration but it is also going to contain a definition okay but that is okay we can a declaration and a definition. So long as they are consistent ofcourse. We what we cannot have are 2 definitions, so 2 pieces of code cannot be given. You will also have noted that there is another include over here, include simple CPP. Now it is in this braces but anyway that simple CPP file is included it is picked up from some specific places and that file contains the declarations of all things like `turtleSim`, right, left okay also things yeah so all those things okay and that is why you have not been needing to actually include put in declarations for all the functions that you have been using so far. And that file simple CPP also includes and include file itself called `cmath` okay which contains the declarations of square root sign and all of that. So the include files have been putting in all the declaration that you really needed.

(Refer Slide Time: 15:13)



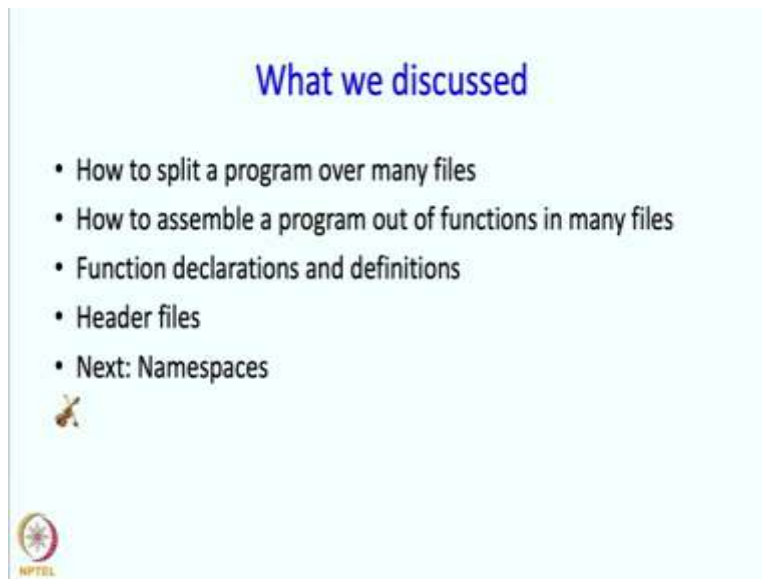
More on header files

- Header files customarily have the suffix `.h` or `.hpp`, or no suffix.
- If header file is mentioned in `" "`, it is picked up from the current directory.
- If it is mentioned in `<>`, it is picked up from some standard place, e.g. `simplecpp`





So a little bit more on header files, header file customarily have the suffix `‘.h’` which is what we just used they could also be `‘.hpp’` or sometimes they do not have any suffix that is also allowed. If the header file is mentioned in quotes it is picked up from the current directory okay and if it is mentioned in angle braces then it is then it is picked up from some standard place, for example simple CPP is present in some in some standard place which the compiler knows about. So you say `include simple CPP s++` will pick it from that standard place.

(Refer Slide Time: 15:49)



What we discussed

- How to split a program over many files
- How to assemble a program out of functions in many files
- Function declarations and definitions
- Header files
- Next: Namespaces



Alright, so what have we discussed? We have discussed how to split a program over many files and effectively how do you assemble a program out of functions in many files and I guess this is really what is most relevant from for you because you are not going split a program but you are going to use functions which are there in several program. So, you want to know how to use those functions. Then we have discussed what are function declarations and definitions and we also talked about header files. Next we are going to talk about namespaces which is a C++ feature which helps in writing program across multiple files. So we will take a break.