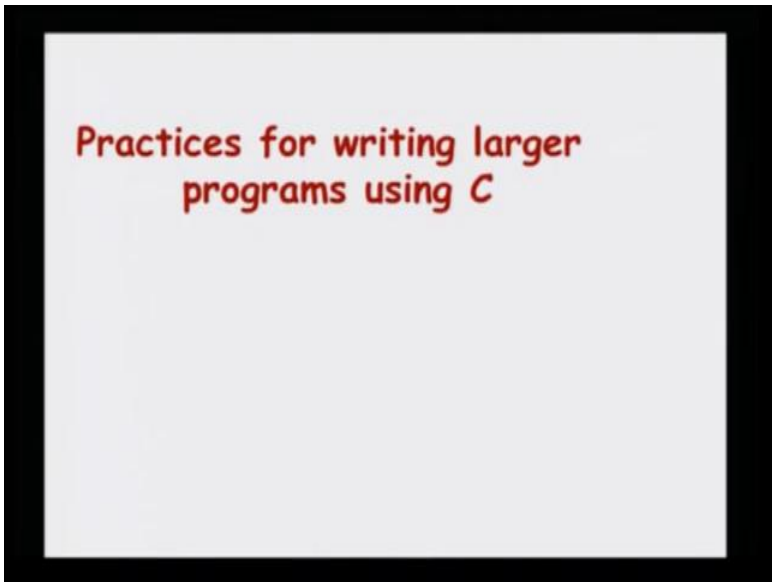
Introduction to Programming in C

Prof. Satyadev Nandakumar
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 53

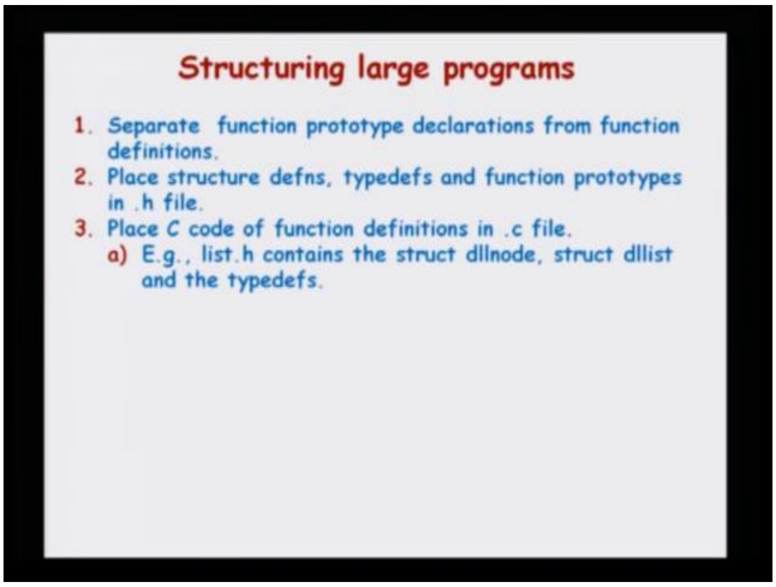
(Refer Slide Time: 00:07)



Practices for writing larger programs using C

In this lecture we will see some practices for writing larger programs using c. As far as we have seen so far, we always wrote our code in a single file; and this is not practical for very large programs running into C 1000 of lines or millions of lines. So, we will see what is the usual practice for organizing a code when we have larger programs.

(Refer Slide Time: 00:32)



Structuring large programs

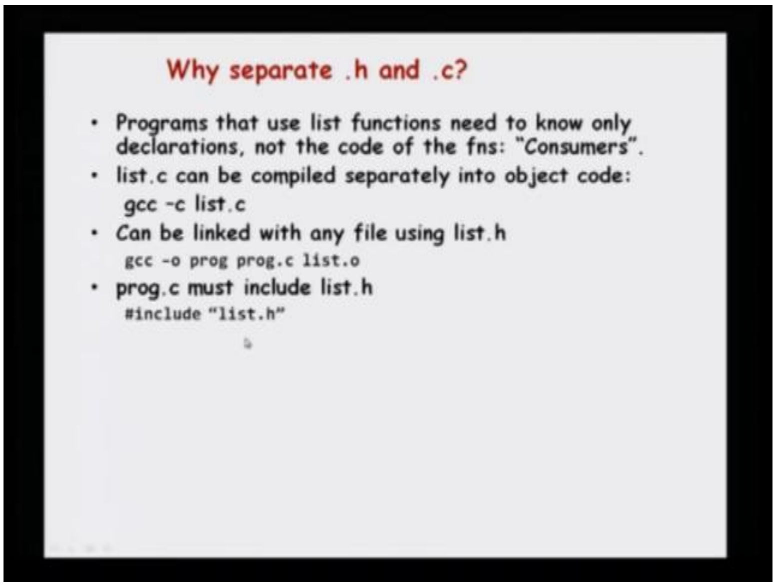
1. Separate function prototype declarations from function definitions.
2. Place structure defns, typedefs and function prototypes in .h file.
3. Place C code of function definitions in .c file.
   a) E.g., list.h contains the struct dllnode, struct dllist and the typedefs.

So, one of the basic principles is to separate the function prototype declarations from the function definitions. We have seen that when we have a function we have 2 things to do - one is to declare a function which is just the types involved in the function, and then the definition of the function which is actually the code of the function. So, one way to structure it, one principle in structuring is that we will separate out the function prototype from the function definition.

Now, place all prototype definitions, structure definitions, typedefs, so just the declarations, you will place it in a file with suffix dot h. So, right now we have been coding in a file call dot c. So, right now what we are proposing is that the declarations alone we will place it in a separate file with suffix dot h. You have already seen such an example which is stdir dot h, we never bothered about what is inside a stdir dot h. Now, we are talking about how to write these header files.

Now, decelerations are only half the function, right. I mean we have to write the definition of the function, the code of the function; the actual code of the function you place it in a dot c file separately. So, for example, list dot h contains the definitions of the struct dllnode for doubly link list and so on, and list dot c would contain the bodies of the function.

(Refer Slide Time: 02:05)



Why separate .h and .c?

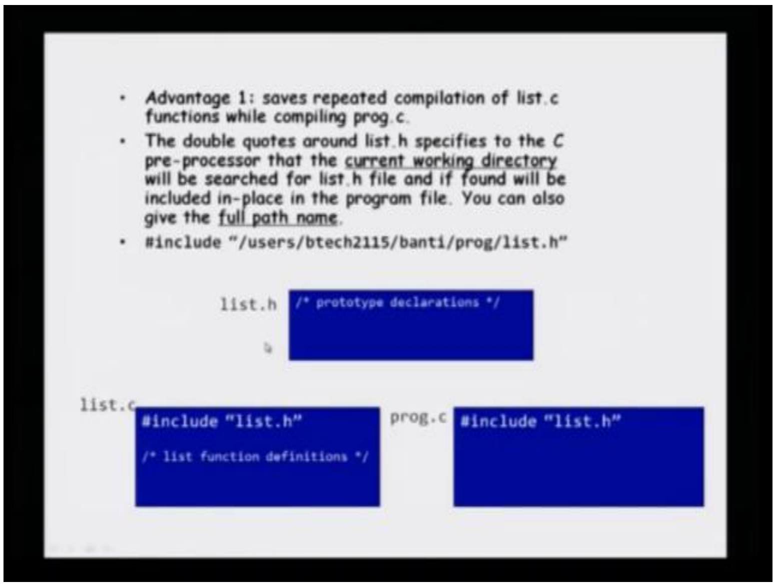
- Programs that use list functions need to know only declarations, not the code of the fns: "Consumers".
- list.c can be compiled separately into object code:
  `gcc -c list.c`
- Can be linked with any file using list.h
  `gcc -o prog prog.c list.o`
- prog.c must include list.h
  `#include "list.h"`

Now, we will see why separate dot h and dot c? Programs that use the doubly link list need to know only the declarations actually. These are, think of these programs as the consumers of this code. Now, it does not need to know how the code is implemented, just what to call and what is the declaration of the function. Now, if we do this, then list dot c can be compiled separately into object code. So, we can say gcc minus c list dot c, this will produce just file call list dot o. List dot o is not executable, but it can be used in other programs to create executables.

So, how can we do this? This is the procedure known as linking. So, we can link the list dot o. So, notice the difference here. When we see gcc minus c list dot c, what it could produce is a dot o function, dot o file; and this dot o file can be included to produce output. So, this says that we are compiling prog dot c file, with list dot o object file, and the output we will produce is called prog. So, gcc minus o prog means, the output file we will produce will be called prog. So, if we omit minus o prog, and the simply say gcc prog dot c list dot o, then the file that we will get is, a dot out. If you specify an output file we will get that output file.

Now, inside prog dot c, let say that we need to use list functions. So, prog dot c will include list dot h; this is the important thing. It will not say include list dot c, it will just say include list dot h. This is similar to what we have seen with stdio; we did not bother about whether there was an stdio dot c file. We said we will include stdio dot h. Also, notice the difference that we are using double codes instead of angular brackets. So, when we wrote stdio dot h, what we had was angular brackets, open angular bracket and close angular bracket, here we have codes; why use that? We will see this.

(Refer Slide Time: 05:04)



- Advantage 1: saves repeated compilation of list.c functions while compiling prog.c.
- The double quotes around list.h specifies to the C pre-processor that the current working directory will be searched for list.h file and if found will be included in-place in the program file. You can also give the full path name.
- #include "/users/btech2115/banti/prog/list.h"

list.h
```
/* prototype declarations */
```

list.c
```
#include "list.h"
/* list function definitions */
```

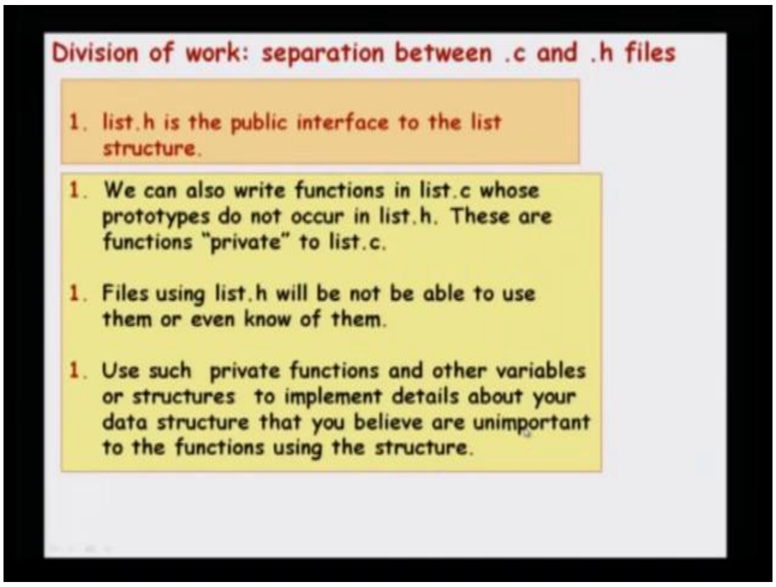
prog.c
```
#include "list.h"
```

So, what is the advantage of separating list dot c from prog dot c, why break it up into multiple files? One advantage is that it saves a repeated compilation of list dot c functions while compiling prog dot c. So, the double codes around list dot h specifies to the c pre-processor; we will see this is in a subsequent lecture, that the current working directly will be searched for list dot h file.

So, since we are saying that include list dot h within double codes, what it means is that, where is list dot h found? It will be found in the current directory. If it is not found in the current directory it will search for some standard library parts, some standard header file parts. So, in the case of stdio dot h we put angular brackets around stdio dot h; that means, that stdio dot h will be found not in the current directory, but in some standard header directories.

So, when you use double codes you can also use some full path names. Suppose, your full path name in a Linux system is slash users, slash btech 2 1 1 5 slash banti slash prog slash list dot h, you can specify the hole path as well. This is the more general notation. So, currently the structure is as follows: you have a list dot h file, it has just the prototype declarations; list dot c will define those functions. So, first inside list dot c you would say, include within codes list dot h, and then have all the function definitions. Prog dot c needs list dot c functions, but instead of saying include list dot c it will say include list dot h within quotations. Now, we will see how to compile such a setup.

(Refer Slide Time: 07:00)



Division of work: separation between .c and .h files

1. list.h is the public interface to the list structure.

1. We can also write functions in list.c whose prototypes do not occur in list.h. These are functions "private" to list.c.

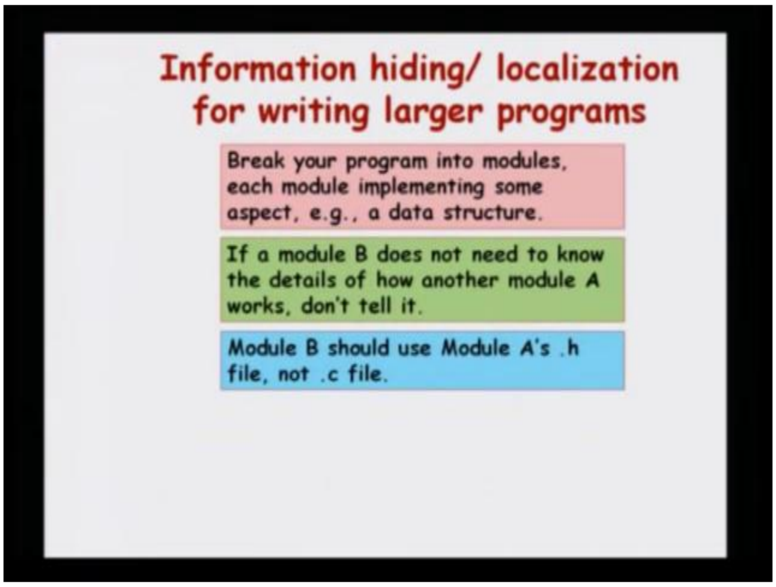
1. Files using list.h will be not be able to use them or even know of them.

1. Use such private functions and other variables or structures to implement details about your data structure that you believe are unimportant to the functions using the structure.

So, what is the division of work? What is the separation between the dot c and the dot h files? Dot h file is the public interface; that, if anybody else, any other program wants to use the list functions what you would do is, include the list dot h functions, include the list dot h file. Now, list dot c implements, defines all the functions that list dot h has declared. In addition, it can also define other functions, but these functions will not be available to other programs that are using the list dot h file.

So, files using list dot h will not be able to use these extra functions or even know about these functions. These are thought of as private functions. So, this can be used to implement certain details of your code that other uses of this program need not know about.

(Refer Slide Time: 08:03)



Information hiding/ localization for writing larger programs

Break your program into modules, each module implementing some aspect, e.g., a data structure.

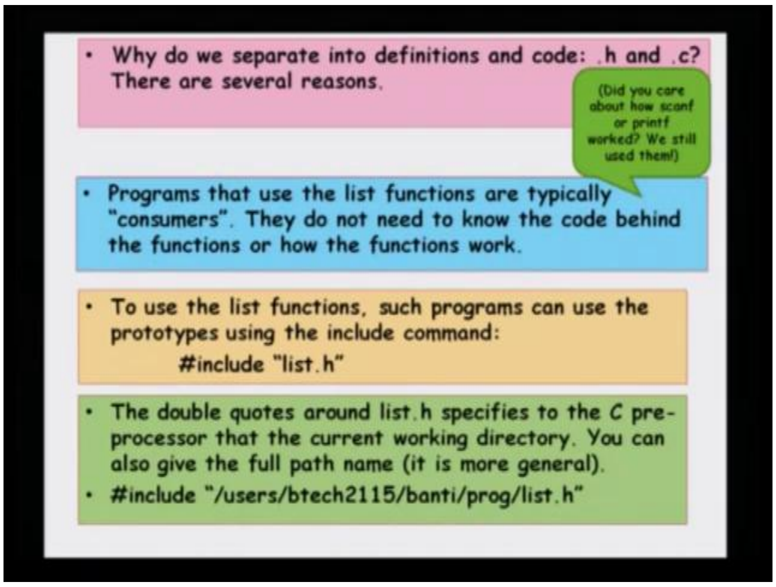
If a module B does not need to know the details of how another module A works, don't tell it.

Module B should use Module A's .h file, not .c file.

So, the general principle is what is known as information hiding or localization. So, break your programs into modules. We have already seen one way to break a program into modules, which is by writing functions. Now, this is another way to, this is another level of module array station where you say that take a collection of functions and put them in a file and have multiple files.

Now, each module implementing some aspect; for example, data structures like a link list. Now, if a module B does not need to know the details of how another module A works, then we do not need to tell B about how it is done. But, module B should use module A’s dot h file, not the dot c file.

(Refer Slide Time: 08:54)



- Why do we separate into definitions and code: .h and .c? There are several reasons.

(Did you care about how scanf or printf worked? We still used them!)

- Programs that use the list functions are typically "consumers". They do not need to know the code behind the functions or how the functions work.

- To use the list functions, such programs can use the prototypes using the include command:

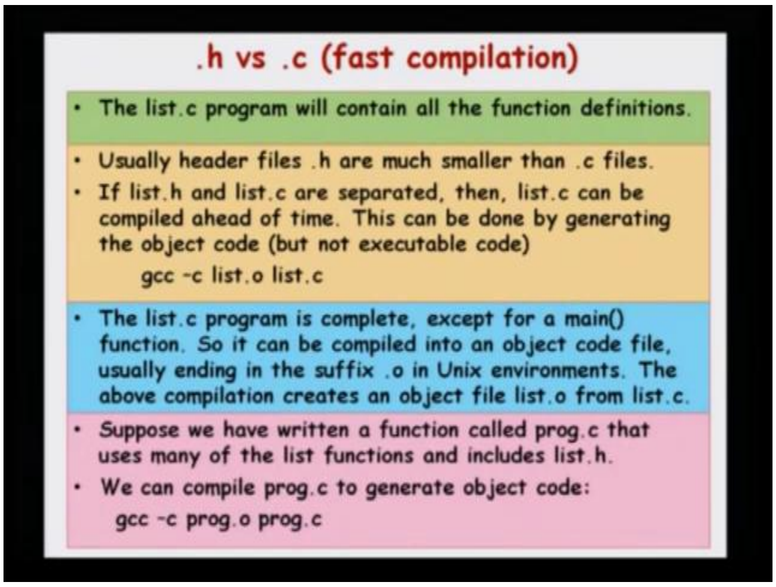
```
#include "list.h"
```

- The double quotes around list.h specifies to the C pre-processor that the current working directory. You can also give the full path name (it is more general).
- #include "/users/btech2115/banti/prog/list.h"

Why do we separate definitions into dot h and dot c? There are some reasons. Programs that use the list functions, for example, are typically consumers, and they do not need to know the exact details behind how these functions work. And we have already done this in other, we do not know about how scan f or print f worked. We just know that scan f needs these 2 arguments, for example, it needs a format string and it needs the variable to be printed; the print f needs the format string and the variable to be printed. Similarly, scan f needs the format string and the variables to be assigned.

So, we just knew that, we do not know anything about how scan f or print f is actually defined or implemented; we just know that it needs these arguments, and therefore we can call them. So, this is the kind of separation of detail that we are hoping to achieve. Now, so if some program wants to use the list functions, such programs can use the prototypes using the include command, slash include within codes list dot h. So, again to remind the double codes specify that it is the current working directory that list dot h is present in; you can also give full paths.

(Refer Slide Time: 10:17)



.h vs .c (fast compilation)

- The list.c program will contain all the function definitions.
- Usually header files .h are much smaller than .c files.
- If list.h and list.c are separated, then, list.c can be compiled ahead of time. This can be done by generating the object code (but not executable code)

```
gcc -c list.o list.c
```

- The list.c program is complete, except for a main() function. So it can be compiled into an object code file, usually ending in the suffix .o in Unix environments. The above compilation creates an object file list.o from list.c.
- Suppose we have written a function called prog.c that uses many of the list functions and includes list.h.
- We can compile prog.c to generate object code:

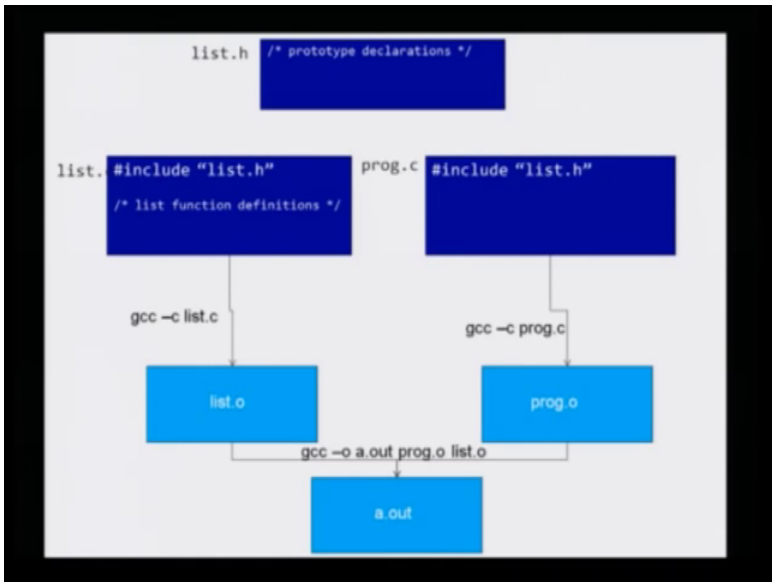
```
gcc -c prog.o prog.c
```

List dot c program will contain all the actual function definitions. Now, usually header files are much smaller than the c files. If list dot h and list dot c are separated, then list dot c can be compiled ahead of time, and you can generate the object file. Now, notice that list dot o in this case will be not executable; it is just an object file that can be used to build executables.

Now, list dot c programs is complete, except for a main function. So, it has a lot of functions; it defines all the functions there list dot h has declared, plus optionally some more functions. And it can be compiled to produce an object code, but it cannot be done, it cannot be compiled into an executable code because it does not have a main function.

Now, suppose we have written function called prog dot c that uses many of the list functions, that uses list dot h. We can compile prog dot c to generate an object code: gcc minus c prog dot o prog dot c. So, now, we have 2 object files, list dot o and prog dot o; and then we can use these 2 object files to create the executable file.

(Refer Slide Time: 11:47)

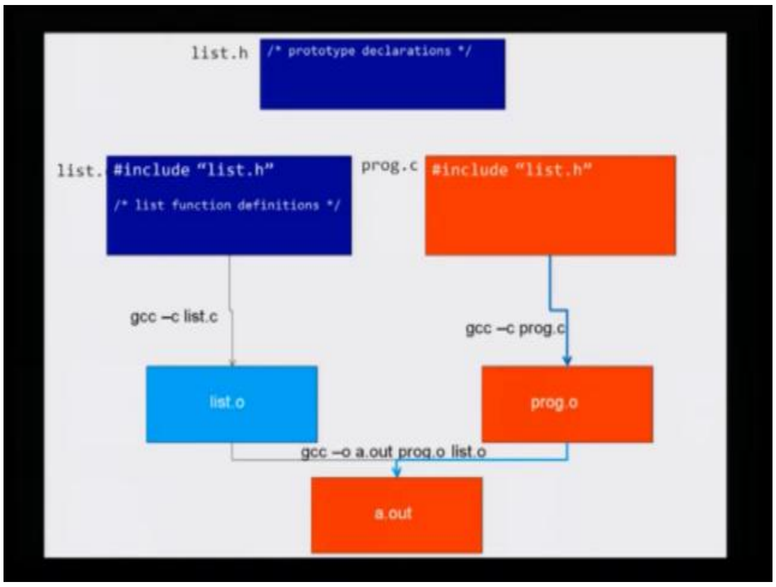


Let us look at a diagram which will hopefully be helpful. So, list dot h includes a prototype declarations, but not the function definitions; list dot c defines these functions. So, in order to define these functions, first it says, include list dot h, therefore, it will get all the declarations, and then it has this code which implements the list function definitions. Prog dot c is a consumer which needs these functions. So, how does it do it? It does not say list dot c, it says include list dot h. So, the declarations of all the functions are available to prog dot c.

Now, I separately compile list dot c into list dot o using gcc minus c, and prog dot c into prog dot o using gcc minus c. So, now, I have 2 object files, list dot o and prog dot o. And these will be combined using gcc minus o. So, this says that the output file will be called, a dot out; the compilation units that I need are prog dot o and list dot o. So, use these 2 files, in order to create the output file, a dot out.

And what is the big advantage here? Let us consider a scene where the prog dot c file changes. I need some changes to be made into prog dot c; maybe I add some more functions, modify some functions and all. So, now I need to recompile and produce the output file. I have changed prog dot c, but not list dot c.

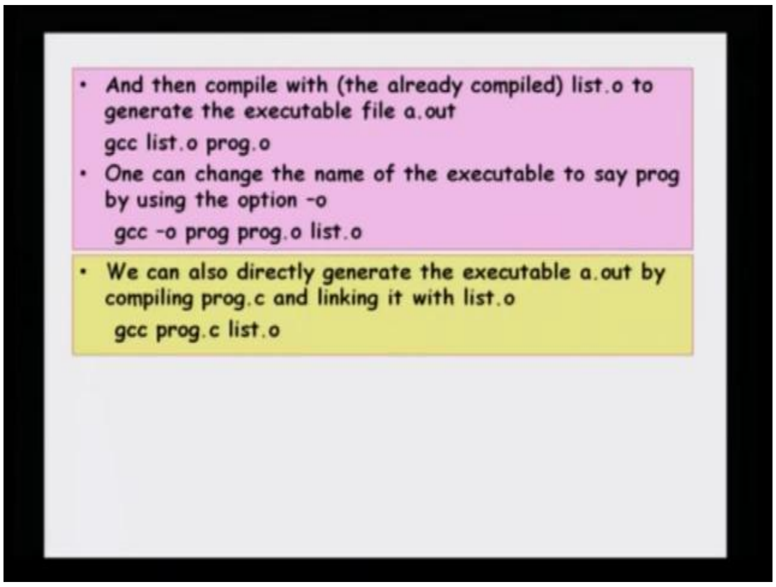
(Refer Slide Time: 13:30)



So, let us say prog dot c has changed. What should I do now? I should compile only the prog dot c. I can say gcc minus c prog dot c; now, I will produce a new prog dot o file. Notice, list dot c has not changed. So, we do not need to recompile list dot c. So, we can just say, gcc minus c prog dot c; list dot o is same as before. And then I can use the new prog dot o, the old list dot o, in order to produce the new a dot o. So, notice the, a dot out depends on prog dot o and prog dot c has changed.

So, only this path gets recompiled which is saving a lot of effort. And in large programs, when one particular file changes and you recompile the project, only the necessary files get recompiled. It does not recompile the whole project which will take a lot of time, instead it will compile only those files which are necessary. So, this is the huge advantage.

(Refer Slide Time: 14:37)



- And then compile with (the already compiled) list.o to generate the executable file a.out

```
gcc list.o prog.o
```

- One can change the name of the executable to say prog by using the option -o

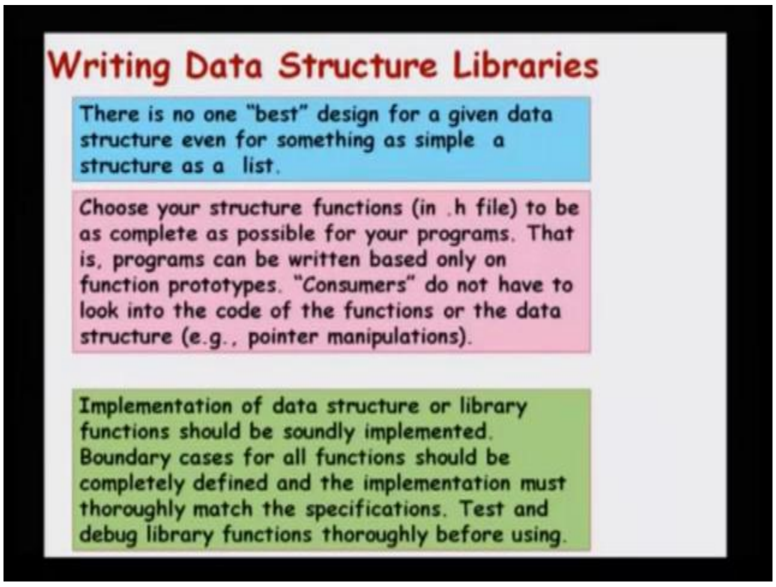
```
gcc -o prog prog.o list.o
```

- We can also directly generate the executable a.out by compiling prog.c and linking it with list.o

```
gcc prog.c list.o
```

So, this just repeats what was said in the last slide.

(Refer Slide Time: 14:49)



Writing Data Structure Libraries

There is no one "best" design for a given data structure even for something as simple a structure as a list.

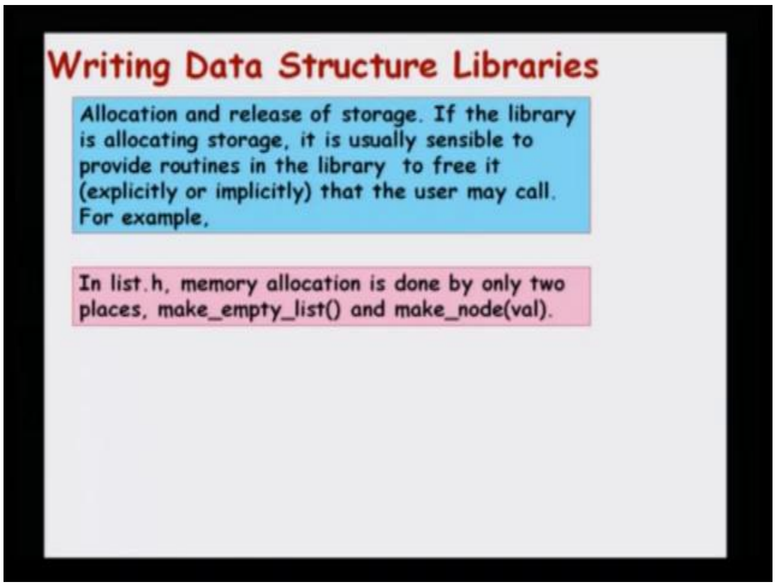
Choose your structure functions (in .h file) to be as complete as possible for your programs. That is, programs can be written based only on function prototypes. "Consumers" do not have to look into the code of the functions or the data structure (e.g., pointer manipulations).

Implementation of data structure or library functions should be soundly implemented. Boundary cases for all functions should be completely defined and the implementation must thoroughly match the specifications. Test and debug library functions thoroughly before using.

Now, couple of thoughts about a writing data structure libraries. There is no one design which is best for a data structure library. Choose your structure functions to be as complete as possible for your programs. Now, programs can be based only on the function prototypes. Suppose I write a program which needs a list function, I can just look at the function prototypes in the dot h file and then write by program. Consumers do not need to know how the program is implemented; just what the functions are, what are its arguments, not the details about how it is implemented.

Now, implementation of libraries should be very sound. All boundary cases should be completely defined and the implementation should thoroughly match the specifications. So, libraries need to be tested and debuged thoroughly before other users can use it.

(Refer Slide Time: 15:50)



Writing Data Structure Libraries

Allocation and release of storage. If the library is allocating storage, it is usually sensible to provide routines in the library to free it (explicitly or implicitly) that the user may call. For example,

In list.h, memory allocation is done by only two places, make_empty_list() and make_node(val).

Also one more thing, allocation and release of storage. If the library is allocating storage, it is only sensible to provide routines in the library itself which can free those storage. So, it cannot be that libraries allocating a storage and the freeing of storage has to be done outside the library, that is not a sensible design. So, if the library itself is allocating storage, you give library functions to free the storing as well. For example, in list dot h, memory allocation is done only in 2 places, make empty list and make node. So, to deallocate that you should provide a free functions for these functions, corresponding to these functions.

Thanks.