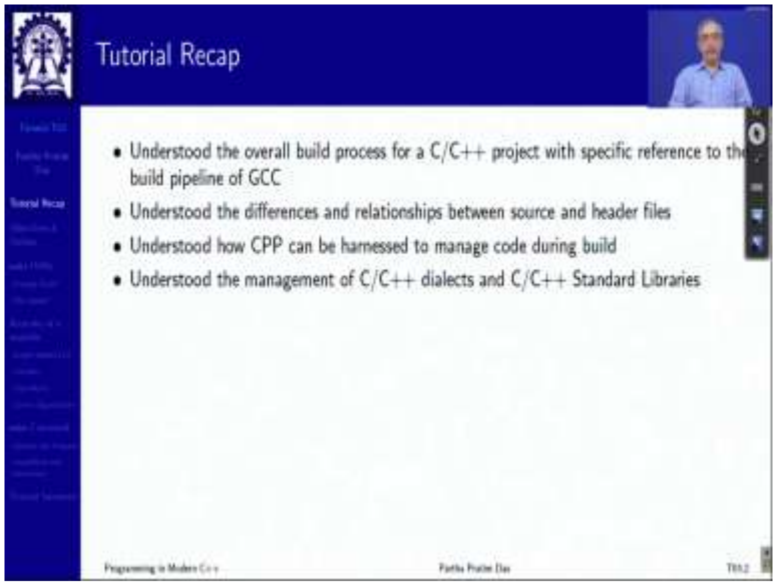
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Tutorial # 03
How to build a C/C++ program?

Welcome to Programming in Modern C++. We have been discussing certain complementary and supplementary content in tutorials.

(Refer Time Slide: 00:49)



Tutorial Recap

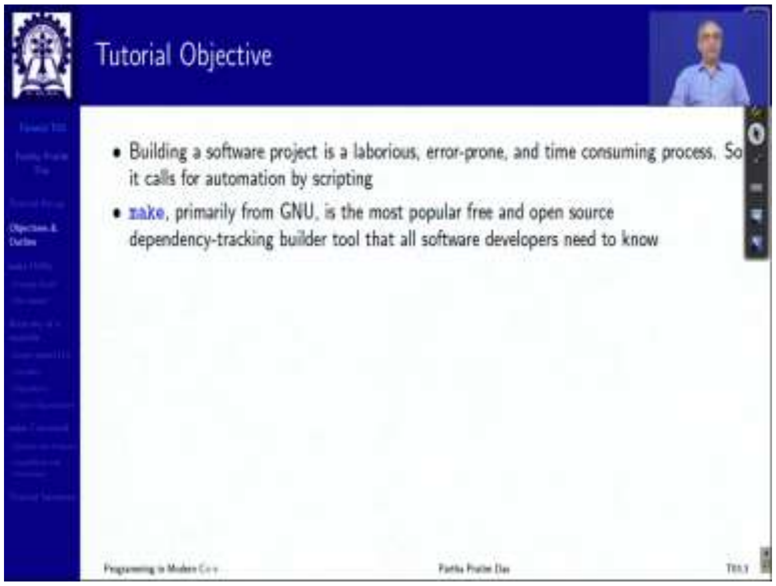
- Understood the overall build process for a C/C++ project with specific reference to the build pipeline of GCC
- Understood the differences and relationships between source and header files
- Understood how CPP can be harnessed to manage code during build
- Understood the management of C/C++ dialects and C/C++ Standard Libraries

Programming in Modern C++ Partha Pratim Das T03.2

Particularly from tutorial one we started discussing about how to efficiently build a C or C++ project. We have made specific reference to the build pipeline of gcc. Though other compilers can also be used in a similar way, we understood the differences between source and header files what to write where and initially, we took a deep look into how C preprocessor CPP can manage the code during the build. We talked about different dialects of C C++ standard libraries.

And we have seen that if we are doing the build manually giving command one by one using gcc then how can we build a project.

(Refer Time Slide: 01:50)



Tutorial Objective

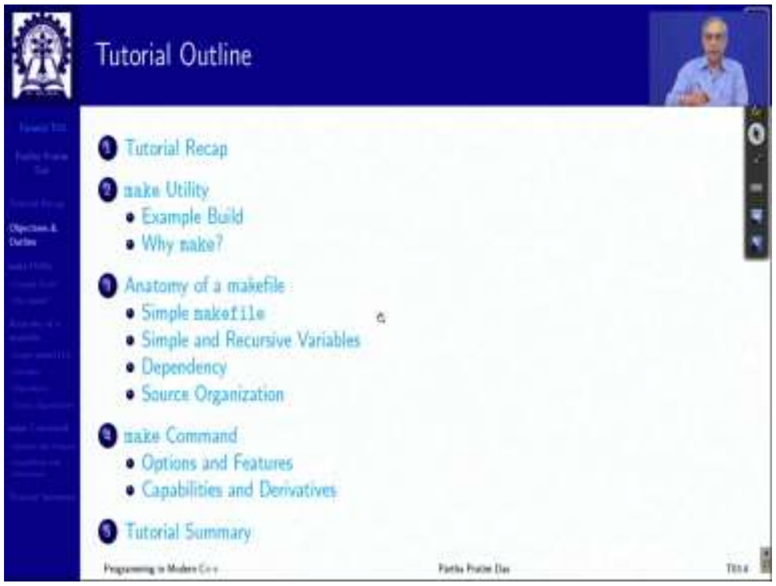
- Building a software project is a laborious, error-prone, and time consuming process. So it calls for automation by scripting
- make, primarily from GNU, is the most popular free and open source dependency-tracking builder tool that all software developers need to know

Programming in Modern C++ Partha Pratim Das T03.3

In this tutorial 03, the third tutorial of the series. We are going to talk about a very interesting utility that GNU provides and kind of it has become all-pervasive in the in the build process everywhere in all kinds of software may not be the specific utility, but utilities on the same principle. The GNU built utilities called make.

How to make a C or C++ build file, it is popular open-source and it performs dependency cracking build and very good to know for any software developer. So, this removes the laborious error prone and time-consuming manual process by introducing a kind of automated scripting.

(Refer Time Slide: 02:54)



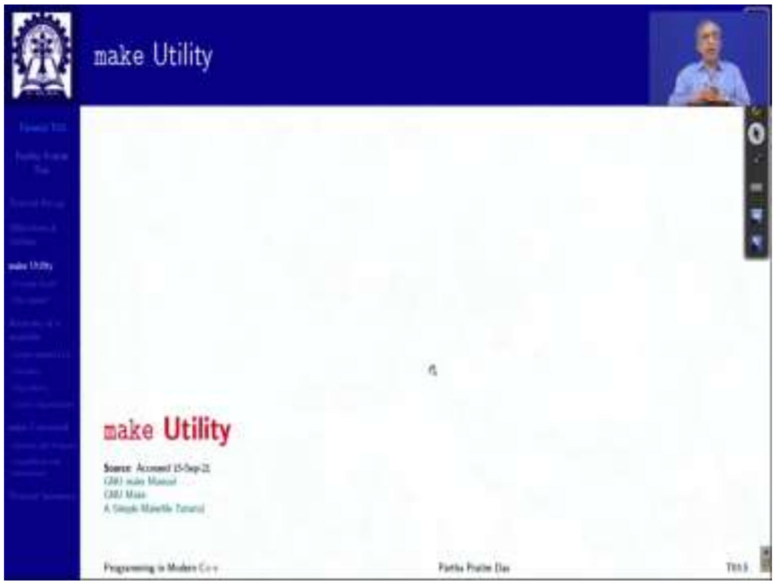
Tutorial Outline

1. Tutorial Recap
2. `make` Utility
   - Example Build
   - Why `make`?
3. Anatomy of a makefile
   - Simple `makefile`
   - Simple and Recursive Variables
   - Dependency
   - Source Organization
4. `make` Command
   - Options and Features
   - Capabilities and Derivatives
5. Tutorial Summary

Programming in Modern C++ Partha Pratim Das T03.4

So, we are going to discuss and understand about this make utility and here is the outline of items which is also on the left bar.

(Refer Time Slide: 03:04)



`make` Utility

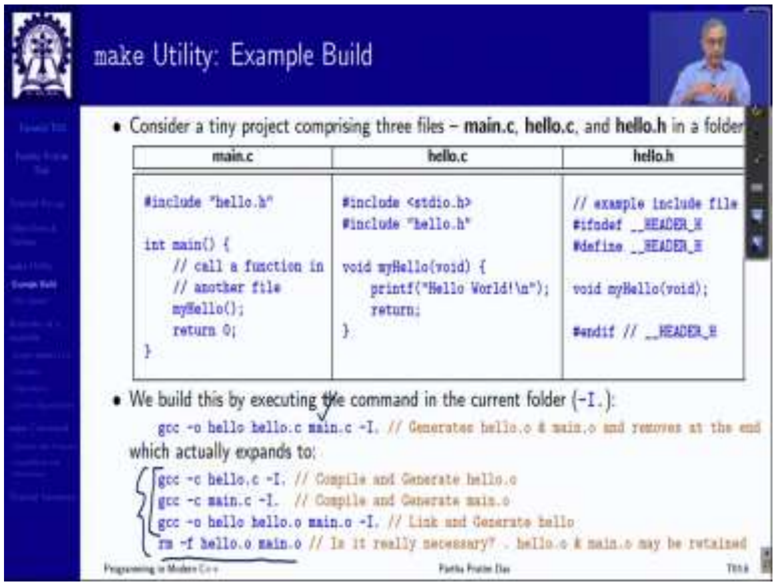
`make` Utility

Source: Accessed 15-Sep-21
GNU make Manual
GNU Make
A Simple Makefile Tutorial

Programming in Modern C++ Partha Pratim Das T03.5

So, make utility if you want to learn more about this, there are certain sources which are I find good they are given here linked.

(Refer Time Slide: 03:17)



make Utility: Example Build

- Consider a tiny project comprising three files – **main.c**, **hello.c**, and **hello.h** in a folder

main.c

```
#include "hello.h"

int main() {
    // call a function in
    // another file
    myHello();
    return 0;
}
```

hello.c

```
#include <stdio.h>
#include "hello.h"

void myHello(void) {
    printf("Hello World!\n");
    return;
}
```

hello.h

```
// example include file
#ifndef __HEADER_H
#define __HEADER_H

void myHello(void);

#endif // __HEADER_H
```

- We build this by executing the command in the current folder (-I.):

```
gcc -o hello hello.c main.c -I. // Generates hello.o & main.o and removes at the end
```

which actually expands to:

```
gcc -c hello.c -I. // Compile and Generate hello.o
gcc -c main.c -I.  // Compile and Generate main.o
gcc -o hello hello.o main.o -I. // Link and Generate hello
rm -f hello.o main.o // Is it really necessary? . hello.o & main.o may be retained
```

Programming in Modern C++ Partha Pratim Das T03.6

So, let us first recapitulate the example build process using a very simple project, which has a it is basically the Hello World project, but written in a little bit more structured manner to introduce the basic concept of make. So, we have a hello.h, which basically defines the header of the myhello function.

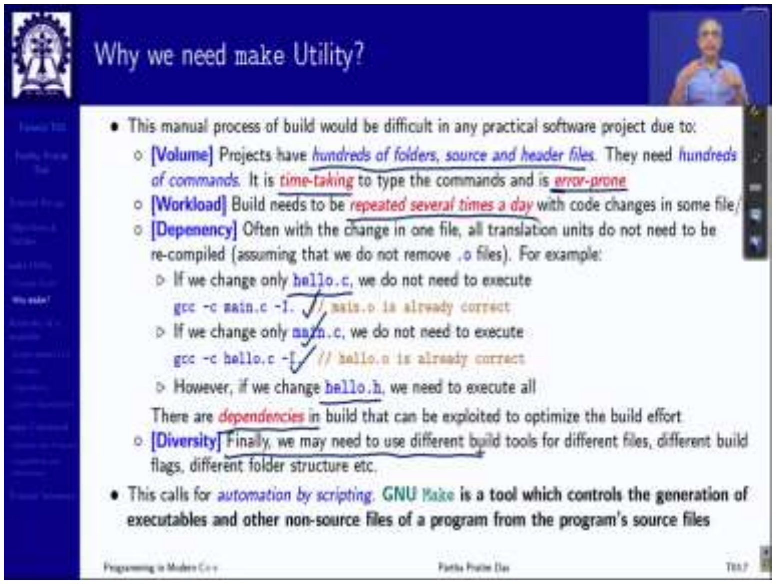
Then we have hello.c which is an implementation of hello.h which for that myhello function provides the body, the implementation, and finally we have the application file main.c which calls myHello as it finds in the hello.h header. So application file source file and header file three components of the project are here. And as you have learned by now, this is we can we can build this in the current folder, the current folder is designated by dot.

We can build this in the current folder by this gcc command, gcc - o which says the target name is Hello. And these are the two source files the translation units to build and the include of the header hello.h will happen will be available in the current folder that is what this - i dot designates gcc command we all have learned and understood this well.

Now, when we do this command, actually what happens are these or rather, I should mark only the first three, that it builds hello.c, creates the hello.o, builds my dot main.c, creates main.o, and then links hello. o and main.o to build the final executable hello.

Optionally, I could also when I actually give this command it also actually what it does is main.o and hello.o that it has created, it will remove them from the folder as if they were temporary files. So this is the basic manual process of build that you have already seen.

(Refer Time Slide: 05:59)



Why we need make Utility?

- This manual process of build would be difficult in any practical software project due to:
  - [Volume] Projects have *hundreds of folders, source and header files*. They need *hundreds of commands*. It is *time-taking* to type the commands and is *error-prone*
  - [Workload] Build needs to be *repeated several times a day* with code changes in some file/
  - [Depenency] Often with the change in one file, all translation units do not need to be re-compiled (assuming that we do not remove .o files). For example:
    - If we change only hello.c, we do not need to execute
      `gcc -c main.c -I. // main.o is already correct`
    - If we change only main.c, we do not need to execute
      `gcc -c hello.c -I. // hello.o is already correct`
    - However, if we change hello.h, we need to execute all

    There are *dependencies* in build that can be exploited to optimize the build effort
  - [Diversity] Finally, we may need to use different build tools for different files, different build flags, different folder structure etc.
- This calls for *automation by scripting*. GNU Make **is a tool which controls the generation of executables and other non-source files of a program from the program's source files**

Programming in Modern C++ Partha Pratim Das T03.7

Now, we are going to show you how to automate this process through scripting. So before we get into that utility, let us try to see why we at all need this kind of utility. There are several reasons and the most important ones I have listed here. The first reason is volume. The typical projects we are now doing just, in course academic projects, 2 3 4 files, but a typical projects have hundreds of folders, source files, header files, and then in hundreds of commands.

So it is time taking to type these commands. And if we have to try to do so much of typing, it is not only the time but it is going to be error-prone. So handling volume is a is a major motivation for going for scripting. Second is the workload. See, it is not that, that I have created a project and I build it only once. I will be building it several times repeatedly even within a day because as I add code and I correct code as I run and see that things are not as I wanted, as I debug.

So repeated builds are a regular requirement. So it is not only the volume of task also, the overall workload is really I mean it might actually be more time taking and laborious than the actual coding and fixing problem.

The third is often there are dependencies between these builds that we do of the translation unit, what do we mean by dependency say for example, in the project we are seeing, suppose we have made some changes only in the hello.c. If we have only changed hello.c, we just need to build hello.c into hello.o, not main.c again assuming that we have not removed the .o files after the last build.

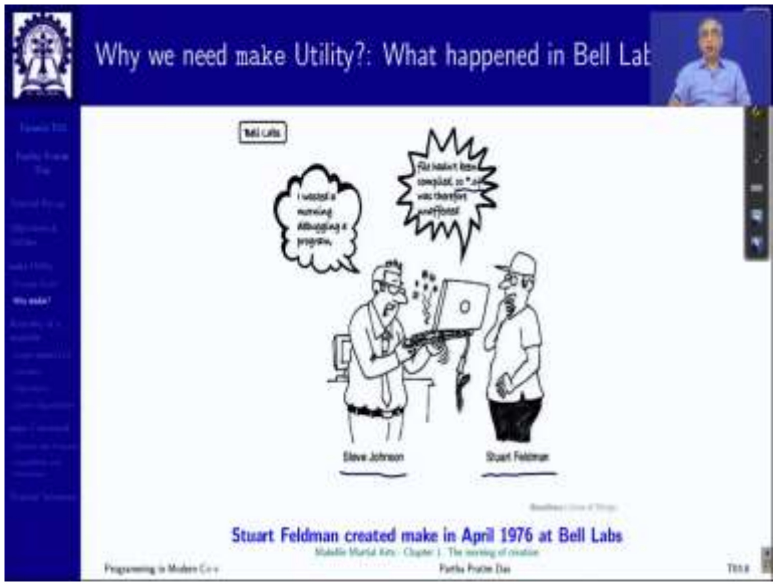
So, we just need to if the change only hello.c then we do not need to execute this command. Similarly if we have changed only main.c, we do not need to build this hello.c into hello.o. However, if this is if changes have happened in the source files, but what if I have made a change on the header file?

Now, this header file as we have seen is included both in hello.c as well as in main.c. So, if I made some change in the hello.h then I will actually need to build both of them. So, there is a decision to be made in terms of what file are which files have been changed and based on that which translation units need to be rebuilt and what linking needs to be done and so on so forth. And that is what is known as the dependency.

Dependency says that if I change a file, then what are the files that have already been built in .o, or say in even in other .i, .s, If we are doing any other files that will get affected by the change in the header or source file that have met. So managing that, remembering that itself would be very, very difficult problem. If the project is large. Then there is a question of diversity. There could be different build tools.

I might use some specific libraries which may require a different set of tools, different flags, different folder structures and so on. So there is a diversity in the project. In a large project it is not as flat as just having three files in the current folder. So, all of these we need to be managed in an automated manner. So that it is efficient it is easy to make it correct it is easy to repeat that and we can always kind of compute what is the optimal amount of tasks that we need to do by analyzing the dependency and just build that much. So, GNU make is such a tool.

(Refer Time Slide: 10:30)



Why we need make Utility?: What happened in Bell Lab

Bell Labs
I wasted a morning debugging a program.
File hadn't been compiled, so *.o was therefore unaffected
Steve Johnson
Stuart Feldman

Stuart Feldman created make in April 1976 at Bell Labs

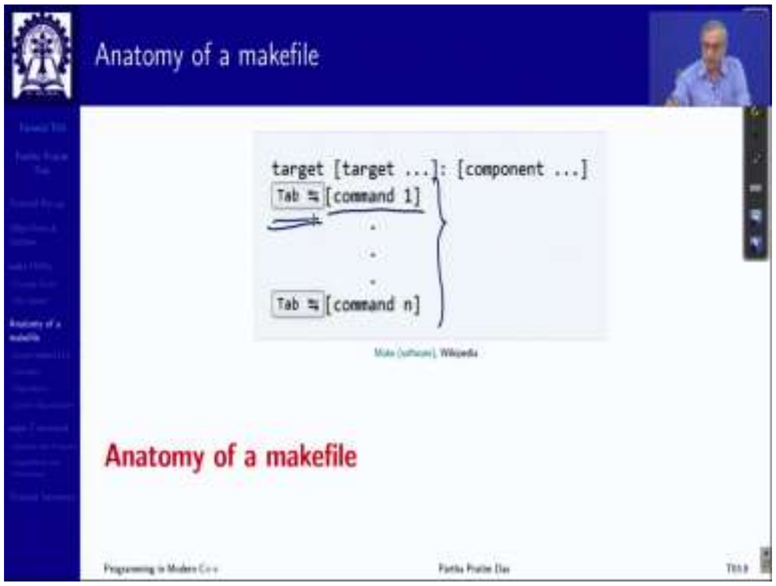
Programming in Modern C++ Partha Pratim Das

Here is an interesting cartoon, which refers to the historical origin of make file or make utility, though exactly the make that we use today is somewhat different. Make was originally created by Stuart Feldman in April 1976. Sothat is nearly 50 years back and it was done in Bell Labs. So what motivated or Stewart Feldman to go for such utility is his friend, Steve Johnson.

So, Steve Johnson was a very gifted programmer and he was working on some complex project and he was exasperated and came to Stuart and said that Stuart had been working since morning trying to debug a program and I wasted my whole morning because the file that I compiled was not a correct one.

So, with that compile, my actual CC * .o, CC is another form of compiler gcc kind compiler dot object files, were not properly updated. And therefore, I wasted all that all that time. So, this is the kind of problem so Stuart felt that why do we have to do this manually? Why cannot we automate this process and that gave birth to the make utility.

(Refer Time Slide: 12:05)



Anatomy of a makefile

```
target [target ...]: [component ...]
Tab ⇆ [command 1]
        .
        .
        .
Tab ⇆ [command n]
```

Make (software), Wikipedia

Anatomy of a makefile

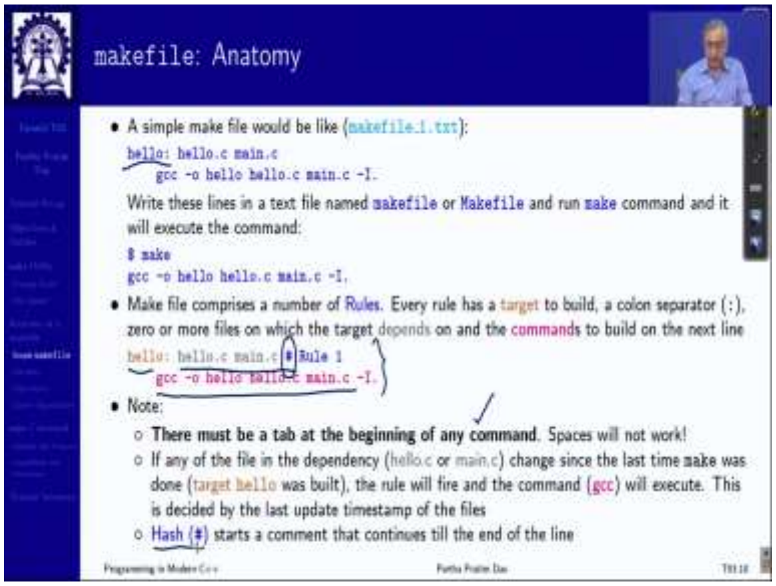
So, how does a make file look like the anatomy of a file, it is somewhat like this that there are what is known as targets. Target is something which you want to create by doing the make and there could be multiple targets. So, here, when we write something within square bracket, it means that it is optional, I may have it once or I may not have it at all.

So the target, so, there has to be at least one target, that is why the first target does not have the bracket, but there could be multiple targets. Then there is a colon separated, which is very important, and then we have zero or more components. So, it says that if I have the target to build this particular file, then what are the components that participate into it? Or in other words, what are the components whose change would trigger me to rebuild, remake this target?

So, that kind of gives you kind of the dependency information or the input output information. Then the question is how do I actually do the make or how do I actually do the build. So, for that, we give commands, again, there could be I mean, in some cases, there may not be a command and there could be multiple, but every command must be within one line.

And it must start with a tab, this is very, very important in the syntax of the make utility, if we just give space, it might look the same, but it will not work, you will get very weird kind of errors. So always make sure that the first line first character of the command must be a tab. So use the tab key to provide that and there could be multiple commands to this. So this is the basic anatomy, you could look at it in the code details.

(Refer Time Slide: 14:08)



makefile: Anatomy

- A simple make file would be like (makefile_1.txt):

```
hello: hello.c main.c
	gcc -o hello hello.c main.c -I.
```

Write these lines in a text file named makefile or Makefile and run make command and it will execute the command:

```
$ make
gcc -o hello hello.c main.c -I.
```

- Make file comprises a number of Rules. Every rule has a target to build, a colon separator (:), zero or more files on which the target depends on and the commands to build on the next line

```
hello: hello.c main.c # Rule 1
	gcc -o hello hello.c main.c -I.
```

- Note:
  - **There must be a tab at the beginning of any command**. Spaces will not work!
  - If any of the file in the dependency (hello.c or main.c) change since the last time make was done (target hello was built), the rule will fire and the command (gcc) will execute. This is decided by the last update timestamp of the files
  - Hash (#) starts a comment that continues till the end of the line

Programming in Modern C++ Partha Pratim Das T03.18

As like this, so here is the target. Here are the components, I have two components here. And here is my command and what you cannot see but has to be actually is a tab not just spaces. And I have made some pointers for the file names. So these text files tiny make files will also be available to you for practice. So, we write this.

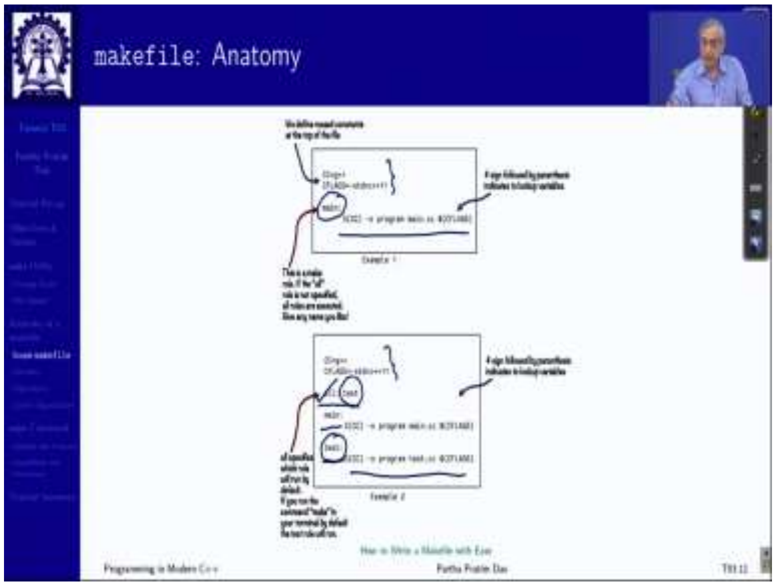
So if we put this command into a file, which name is make file, or make file with a capital N Then run the command make, then it will execute that command. So what it will do? It will take the target hello. And this is the only target given. So it takes that target and executes this command, this is a sample. Now, there are obviously several questions.

First of all, if I write my commands into a text file called make file or capital M make file, then I can just to make, but if I use something like this that I have written here, then how do I tell that make according to that file, so I can do that by putting - f and then giving the name of the file, we will come to those options later on.

The other question certainly is here we are talking about a very simple make file, which has only one target, what if you have if I have multiple such targets in the same make file, which one do I bid. So, there are there are ways of doing that there are ways of specifying that as well. So, make file in general comprises, a number of rules.

So, these are my rules, target, components and command followed by the tab. Hash in a make file means a comment from that point till the end of the line. So, every there has to be a tab we have already saved. And hash is for the command. These details are noted here.

(Refer Time Slide: 16:43)



makefile: Anatomy

Example 1

Example 2

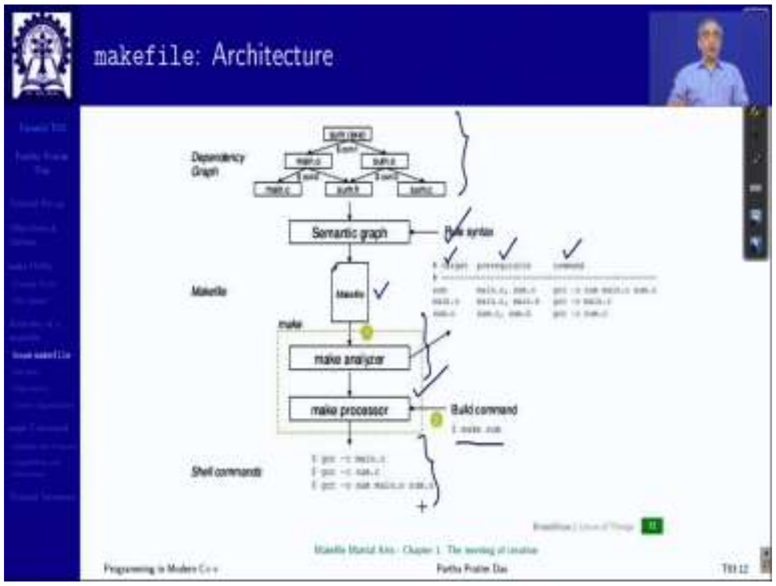
How to Write a Makefile with Ease

Programming in Modern C++ Partha Pratim Das

So, that is the basic structure of make file, we say the anatomy of the make file. So if I look at the anatomy in little bit more detail, we will see that here is a target, here is a command. And we, you have some more stuff here, which is kind of variables, and we will see what variables are and why they are important. And as you can see, here, there are multiple targets.

So you can if you just do make, it always does the target, which is named all that is a given name. For other targets, you can give your own name. So what all says, all says that to make all I need test. Now test here in this case, in general, it could be any file, but in this case, the test is a, is another target. So all will make sure that you do this particular build command. So, there could be several variants of that will slowly come to all that.

(Refer Time Slide: 17:51)

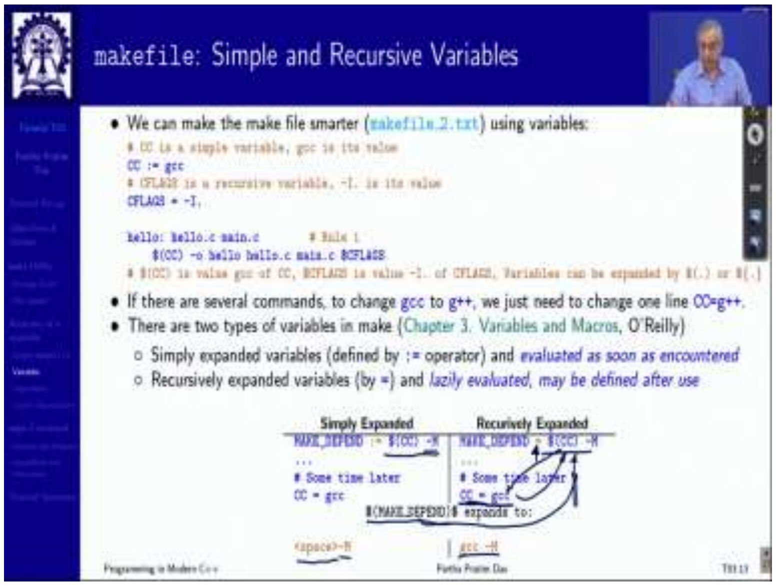


Now, what happens is. I mean you may want to, retain this or you may want to skip this also that is what actually happens internally is based on the target and the components, there is a dependency graph, because every target depends on the components. So there is a dependency graph and that has to be that dependency graph must be a cyclic it cannot be, cannot have a cyclic but because then you will not be able to make that.

So that is through the rule syntax, it converts into a semantic graph. And that itself is the make file, this is what we are doing for creating the make file which we will we are going to learn. Then there is a analyzer for the make which decides on the target the prerequisites, the components and the command, those are created in the analyzer, then there is a make processor, there is another make compiler which compiles the makes script, make is becoming a language.

So there will be a compiler for that which is called the make processor, which actually takes a build command. And from the build that has been done, it generates the actual commands on your command prompt or the shell prompt. So that is the that is the overall flow of that make architecture that you will have.

(Refer Time Slide: 19:22)



makefile: Simple and Recursive Variables

- We can make the make file smarter (makefile_2.txt) using variables:

```
# CC is a simple variable, gcc is its value
CC := gcc
# CFLAGS is a recursive variable, -I. is its value
CFLAGS = -I.

hello: hello.c main.c        # Rule 1
    $(CC) -o hello hello.c main.c $CFLAGS
# $(CC) is value gcc of CC, $CFLAGS is value -I. of CFLAGS, Variables can be expanded by $(.) or ${.}
```

- If there are several commands, to change gcc to g++, we just need to change one line CC=g++.
- There are two types of variables in make (Chapter 3. Variables and Macros, O'Reilly)
  - Simply expanded variables (defined by := operator) and *evaluated as soon as encountered*
  - Recursively expanded variables (by =) and *lazily evaluated, may be defined after use*

Simply Expanded	Recurively Expanded
MAKE_DEPEND := $(CC) -M	MAKE_DEPEND = $(CC) -M
...	...
# Some time later	# Some time later
CC = gcc	CC = gcc

```
$(MAKE_DEPEND)# expands to:
```

<space>-M	gcc -M

Programming in Modern C++ Partha Pratim Das T01.13

Now, there could be variables here, for example, you will find in this we can write CC colon equal to gcc, which means cc is a simple variable. So, wherever I write, CC within dollar parenthesis, it will actually mean this variable.

Similarly, C flags is another variable, which I have written is equal to - i dot. There is a certainty here you will find that In this case I have written it as colon equal to which says that cc is a sample variable. And here I have written it without the colon, which says that C flags is a recursive variable, we will just immediately discuss what they mean.

A simple variable is expanded as soon as it is encountered. So this example is illustrative I write a variable make expand, do colon equal to, which means that I am doing a simple variable, so which means that if I do $(CC) - m is the meaning of this variable. So wherever I will use this variable and every variable is to be used by a within a dollar.

So, as I do dollar make depend, then dollar will expand to whatever it could be. So, here is the recursive one. The difference in the recursive one is that it is lazy in evaluation. What do I mean by that? Look at it here very carefully, here, while I am doing make depend, I am saying that make depend is whatever $(CC) is followed by - m.

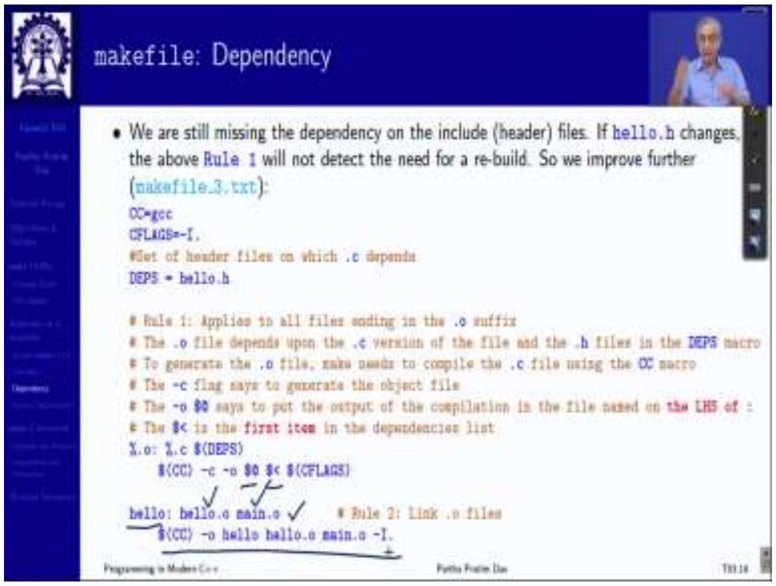
So, it is referring to another variable. Incidentally, that variable is not been defined as yet that variable happens later. The definition of the variable happens later. I do the similar thing here and also the variable difference later. Now, what happens if it is a simple variable, as soon as the definition of that variable comes as soon as this comes, it is evaluated. So, what is $(cc) has not been defined.

So make by default issues that it is a blank space, so it expands it to a space then - m, - m, but when I have it as lazy as a recursive definition, by virtue of not having the colon here. Then it does not immediately expand $(CC) it just remembers this is so when later on it finds the definition for $(CC) and after that, when it actually has to do make depend. At that time, it goes back to the definition and start evaluating.

So, this goes back now here by this time it has found CC, so it knows CC is gcc. So it takes and puts gcc here and expands to gcc - m. So, in a simple variable, whenever you encountered you evaluate in terms of a recursive variable, you remember the definition and you evaluate only when you are going to use it only when you are going to expand it.

So, in between this, if certain missing variables have been defined, then they will be properly used. So that is a basic difference between these two. So, if you have if I have to use only simple variable, then in this case, I have to use I have to put CC equal to gcc before I put makedepend equal to $(CC) - m, but if I use recursive variable then I can put them in any order. But obviously, before I try to evaluate make depend.

(Refer Time Slide: 23:59)



makefile: Dependency

- We are still missing the dependency on the include (header) files. If hello.h changes, the above Rule 1 will not detect the need for a re-build. So we improve further (makefile.3.txt):

```
CC=gcc
CFLAGS=-I.
#Set of header files on which .c depends
DEPS = hello.h

# Rule 1: Applies to all files ending in the .o suffix
# The .o file depends upon the .c version of the file and the .h files in the DEPS macro
# To generate the .o file, make needs to compile the .c file using the CC macro
# The -c flag says to generate the object file
# The -o $@ says to put the output of the compilation in the file named on the LHS of :
# The $< is the first item in the dependencies list
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

hello: hello.o main.o ✓     # Rule 2: Link .o files
    $(CC) -o hello hello.o main.o -I.
```

Programming in Modern C++ Partha Pratim Das T03.14

So this is my this is the way I can have variables and variables basically make it easy to write. For example, I am saying CC is, I am saying CC is a variable defined as gcc, how does it tell wherever I have to write this compiler name I will put this. So later on, if I decide no I do not want gcc here I want the g++ here or I want CC here. CC is a is another Linux compiler which is which is very, very, has been widely used as well.

I do not have to go and make changes at every point. I just have to change the definition here. So in a way, I mean if you think in terms of principles and terms of philosophy, you can say that this is like doing a hash define. We are hash defining symbols why because that symbol is used at multiple places. And instead of having to change it everywhere, I can make changes only at once. So, and also naturally using variables make it easier to, read the code makes it easier for me to put commands and understand what I am doing and so on.

So, here I have two variables both recursive and then I have another third variable which say, what is the dependency that is what is the name of the header file on which the bill depends and then I write a rule like this. Just to understand this look at the rule. Left hand side is the target. So what am I saying is the target I am saying that the target is %.o.

What does percentage mean? Percentage means is any string which will match here, if it is followed by .o. So any name of any file, which has .o extension will match the target. So I am saying so, in other words, I am saying that to build a .o file, you have to make use of the dependency on the right. What are the components?

The first component is percentage .c, that is whatever you have taken as .o file name say hello.o, then by the same name, you will have a component called hello.c. If I do main if I found also main.o, then I have a main.c. So, this is the first component and the second I say depends on $Deps. That is what is Deps? I say it is depends on hello.h because that is what have defined as a variable.

Now the advantage of saying doing this is if I do not do this, then I will have to write one rule for hello.o, another rule for main.o, if there are 20 Translation units, then I will have 20 rules, but all of them will have some kind of some .c producing .o and a set of files on which it is dependent. So, by the single simple rule, I am able to capture the entire process of source files being compiled into object files as a proper target dependency requirement.

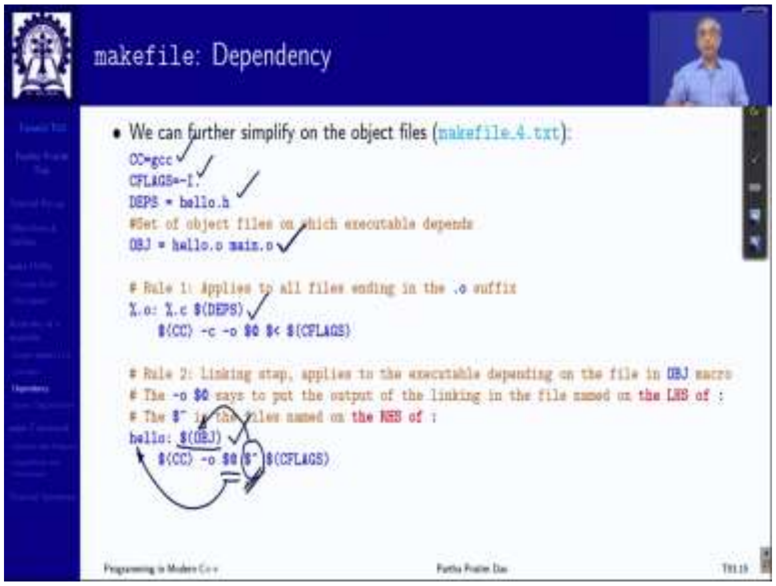
Now, What is the command? The command to build is CC is my compiler whatever I will put here will come here automatically then I say it is - c. So, I am saying that you generate the .o file. Now, I am saying, what is the my and I have this flag which says that the flags is - i dot which says that anything that I include will come from the same folder.

Now, the question is what is the input and what is output? What do I need? I need if my target is hello.o. Then my dependency source has to be hello.c. If my target is main.o my dependency target must be a dependency must be on main.c. So, I say that by these special make symbols, which say dollar at the rate, dollar at the rate means the left hand side of the colon, that is a target and dollar less than means the first component.

So, if I have to make hello.o which is coming from dollar at. Similarly, if I have to make main.o, I have to come from that will have to come from the dollar at. Now, so, then this process in a very compact way I can define the multiple rules into one simple rule I do not need because all these are structures of the same only thing different is what is the .o .c file name what is a .o file name?

And that is where these potential make symbol or make language instructions will take in place. And then I write another rule, which says that my target is Hello, which is the final executable, and these are my components, it depends on the .o files and this is my command. So, this is a simple process that will be involved in putting the dependency together.

(Refer Time Slide: 30:17)



makefile: Dependency

- We can further simplify on the object files (makefile.4.txt):

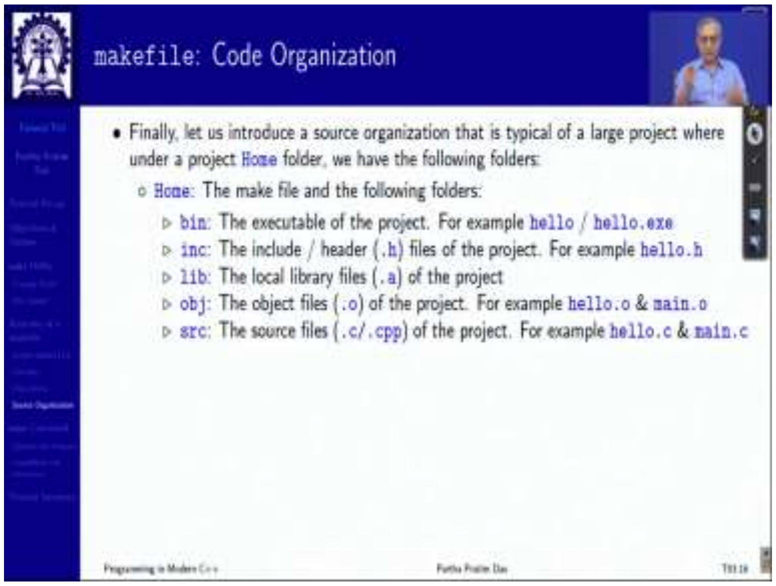
```
CC=gcc
CFLAGS=-I.
DEPS = hello.h
#Set of object files on which executable depends
OBJ = hello.o main.o

# Rule 1: Applies to all files ending in the .o suffix
%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

# Rule 2: Linking step, applies to the executable depending on the file in OBJ macro
# The -o $@ says to put the output of the linking in the file named on the LHS of :
# The $^ is the files named on the RHS of :
hello: $(OBJ)
	$(CC) -o $@ $^ $(CFLAGS)
```

Programming in Modern C++ Partha Pratim Das T01.15

So, if we want to extend that we have this, this this we have seen, we can also want to make the OBJ files into a, set of dependency. So, with that this is same, also the last line, I make it compact, I do not list hello.o main.o, like that, those are all listed at one point. And I can just use them as $OBJ. Again, I use the dollar at to mean the left hand side.

And these are there will be multiple of them. So, dollar hat, are names on the right hand side, all of them will be expanded in here.

(Refer Time Slide: 31:13)



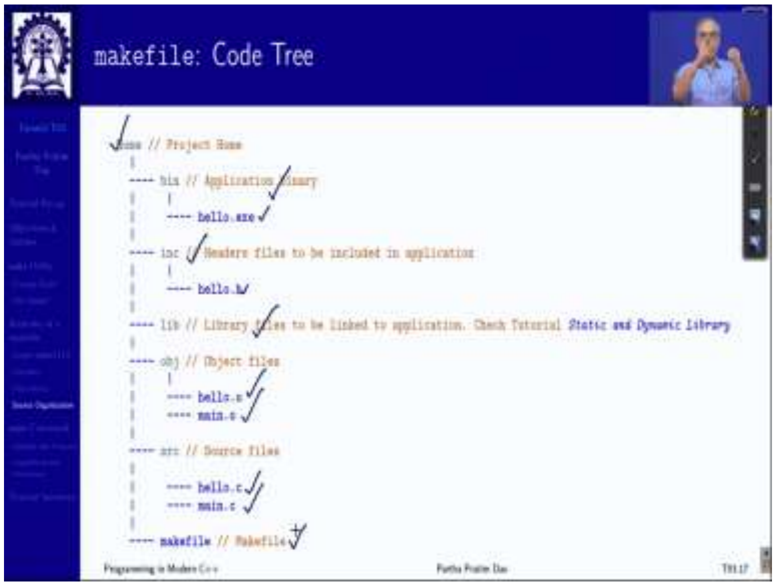
makefile: Code Organization

- Finally, let us introduce a source organization that is typical of a large project where under a project Home folder, we have the following folders:
  - Home: The make file and the following folders:
    - bin: The executable of the project. For example hello / hello.exe
    - inc: The include / header (.h) files of the project. For example hello.h
    - lib: The local library files (.a) of the project
    - obj: The object files (.o) of the project. For example hello.o & main.o
    - src: The source files (.c/.cpp) of the project. For example hello.c & main.c

Programming in Modern C++ Partha Pratim Das T01.16

So, this is how the make can actually work, I am sure you have got a quarter sense of this. So, let us take a little bit of, example to understand more. So, what I do is, we will assume a certain source organization in a big project, you will not put everything in your in a single source file in a single folder. Rather, he will have different folders having different types of files bin for executable that is binary, inc for include, lib for library files, we will see what they are obj for object files, src for source files and so on.

(Refer Time Slide: 31:50)



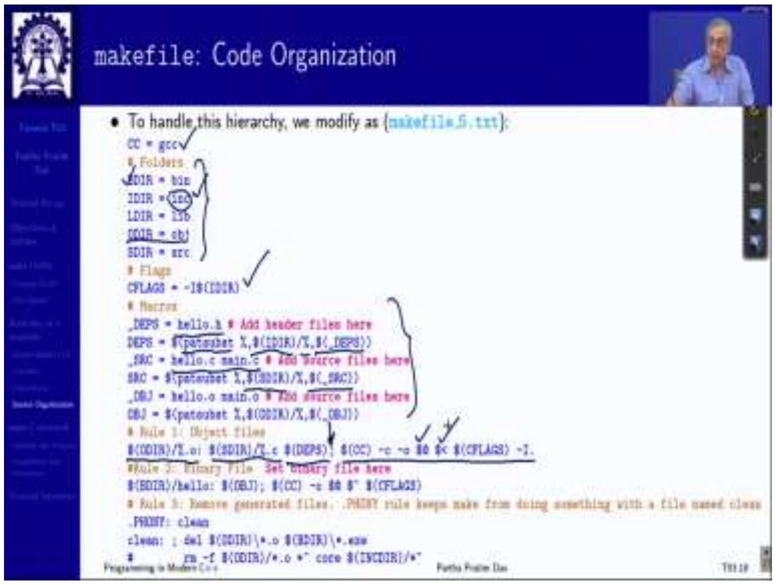
makefile: Code Tree

```
Home // Project Home
  |
  ---- bin // Application Binary
  |     |
  |     ---- hello.exe
  |
  ---- inc // Headers files to be included in application
  |     |
  |     ---- hello.h
  |
  ---- lib // Library files to be linked to application. Check Tutorial Static and Dynamic Library
  |
  ---- obj // Object files
  |     |
  |     ---- hello.o
  |     ---- main.o
  |
  ---- src // Source files
  |
  |     ---- hello.c
  |     ---- main.c
  |
  ---- makefile // Makefile
```

Programming in Modern C++ Partha Pratim Das T01.17

So, let us look at this sample code tree. This is my home project wherever you can put it. Then I have a application binary folder, which will have my exe finally, this is my include folder. This is if I have some libraries, we still are not talking about libraries, we will talk in the next tutorial, we have the object files in obj, source files and in the route that is in the home, I have my make file. So all that I need to do is to go to my home and do make and that should build my entire project and put the files accordingly.

(Refer Time Slide: 32:31)



makefile: Code Organization

- To handle this hierarchy, we modify as {makefile.5.txt}:

```
CC = gcc
# Folders
BDIR = bin
IDIR = inc
LDIR = lib
ODIR = obj
SDIR = src
# Flags
CFLAGS = -I$(IDIR)
# Macros
_DEPS = hello.h # Add header files here
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))
_SRC = hello.c main.c # Add source files here
SRC = $(patsubst %,$(SDIR)/%,$(_SRC))
_OBJ = hello.o main.o # Add source files here
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))
# Rule 1: Object files
$(ODIR)/%.o: $(SDIR)/%.c $(DEPS); $(CC) -c -o $@ $< $(CFLAGS) -I.
#Rule 2: Binary File  Set binary file here
$(BDIR)/hello: $(OBJ); $(CC) -o $@ $^ $(CFLAGS)
# Rule 3: Remove generated files. .PHONY rule keeps make from doing something with a file named clean
.PHONY: clean
clean: ; del $(ODIR)\*.o $(BDIR)\*.exe
#        rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

Programming in Modern C++ Partha Pratim Das T03.18

So that is that is what I do here. This is my command and I define different symbols for my folders. Why would I do that because somebody might want to call the include directory as inc somebody might want to call it include all these different names. So I can just change it here. So that I do not need to change it anywhere Subsequently, I have my flags and then I use a set of macros to actually use these variable names and create the actual expanded name.

So all that we are saying that pattern substitute what do we substitute $IRIR, what is $IDIR? Is inc slash percentage. What is slash percentage? Slash is obviously separated, Percentage is whatever comes from the dependency. So whatever is the dependency underscore dependency as hello.h. So this will give me a string which is inc slash hello.h.

Similarly, I do that for SDIR with src as source which has two source files so it gives me src slash hello.c, src slash main. c like this. Very standard way of writing do not have to remember this form you can always look up the syntax or copy paste from here then my rule becomes very simple with this structure also, my object files will have will be in ODIR, object dir slash percentage .o. You have already seen that dependent on the source files and the dependencies. Very nice.

Here you see an alternate syntax I have put a semicolon normally I have to put the rule in the next line after the tab, but we can put it right after the components by giving a semicolon and then here is my command. You have already understood what this symbols are. They one by one will take the .o and the .c and create so expand into so many commands.

Similarly, I can have it for the Binary File rule the binary folder the obj object has to be it must depend on the object files and the command for that. And so on. It is very typical that you might want to also have a clean target. The clean target is if you if you have got all now all your different folders are getting different files generated and so, what if you want to, it is everything and start a fresh or then you can do a clean one.

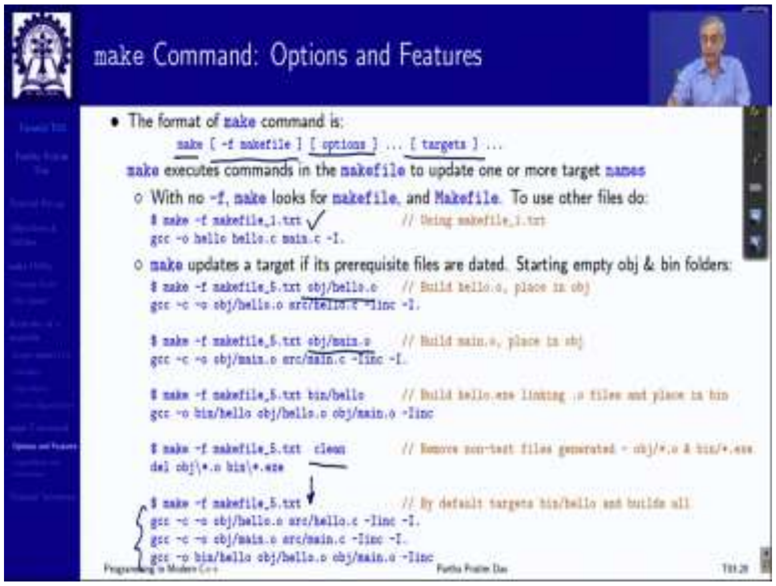
So, which is say in Windows, it will be del object directories cleaned binary directory is clean or rm - f in lunix which will do the same thing. Now, since clean is not actually a file is not a file. Normally targets has to be a file which you generate. So, you have a very typical very interesting exception given in make, where you say that this target is .phony, .phony is known as .phony and you put whatever name on the dependency and that is treated as if it is a file and you can build with that file.

(Refer Time Slide: 36:14)

make Command

make Command

Programming in Modern C++ Partha Pratim Das T01.19



make Command: Options and Features

- The format of make command is:

  `make [ -f makefile ] [ options ] ... [ targets ] ...`

  make executes commands in the makefile to update one or more target names
  - With no -f, make looks for makefile, and Makefile. To use other files do:

    ```
    $ make -f makefile_1.txt ✓              // Using makefile_1.txt
    gcc -o hello hello.c main.c -I.
    ```
  - make updates a target if its prerequisite files are dated. Starting empty obj & bin folders:

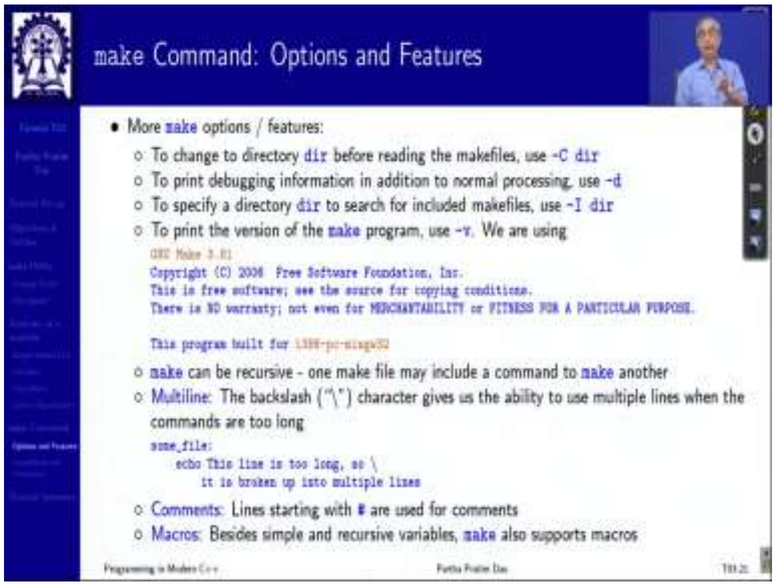
    ```
    $ make -f makefile_5.txt obj/hello.o    // Build hello.o, place in obj
    gcc -c -o obj/hello.o src/hello.c -Iinc -I.

    $ make -f makefile_5.txt obj/main.o     // Build main.o, place in obj
    gcc -c -o obj/main.o src/main.c -Iinc -I.

    $ make -f makefile_5.txt bin/hello      // Build hello.exe linking .o files and place in bin
    gcc -o bin/hello obj/hello.o obj/main.o -Iinc

    $ make -f makefile_5.txt  clean         // Remove non-text files generated - obj/*.o & bin/*.exe
    del obj\*.o bin\*.exe

    $ make -f makefile_5.txt                // By default targets bin/hello and builds all
    gcc -c -o obj/hello.o src/hello.c -Iinc -I.
    gcc -c -o obj/main.o src/main.c -Iinc -I.
    gcc -o bin/hello obj/hello.o obj/main.o -Iinc
    ```

Programming in Modern C++ Partha Pratim Das T01.20

So, this is what the whole structure is. So, the make command is now simple it has make, then optionally you can specify the file name - f, you can give different options for doing. For example, you can ask it to be verbose, you can ask it to actually print whatever it is doing, and then you put the target that you want to build. So, if you just do this without giving anything, it will do the basic routine stuff.

Otherwise, you can say my target is to build hello.o or my target is to build main.o, so you can put anything which is on the left hand side in the target put that or you say that my target is to do a clean, I want to remove everything. And if I do not give a target, then it does all according to the dependency, it does .o's then it will do different .o's and then link them to the final executable. So that is the basic process.

(Refer Time Slide: 37:26)



make Command: Options and Features

- More make options / features:
  - To change to directory dir before reading the makefiles, use -C dir
  - To print debugging information in addition to normal processing, use -d
  - To specify a directory dir to search for included makefiles, use -I dir
  - To print the version of the make program, use -v. We are using

```
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

This program built for i386-pc-mingw32
```

  - make can be recursive - one make file may include a command to make another
  - Multiline: The backslash ("\") character gives us the ability to use multiple lines when the commands are too long

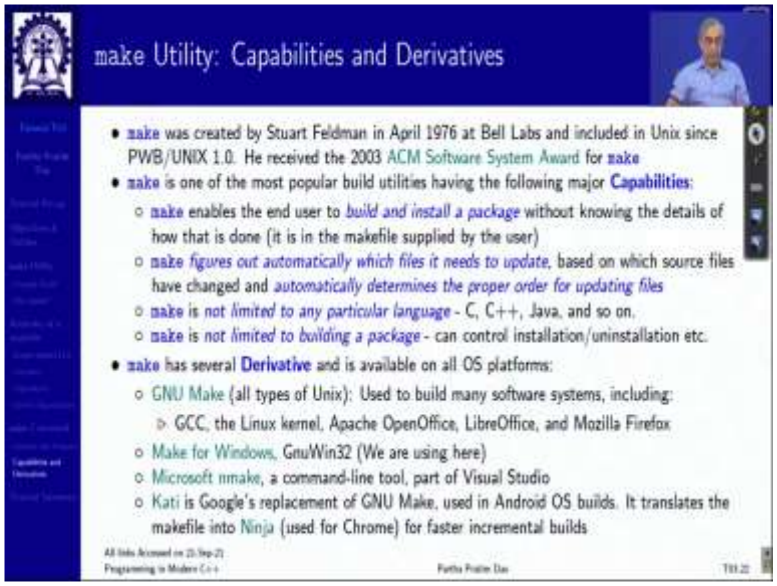
```
some_file:
    echo This line is too long, so \
        it is broken up into multiple lines
```

  - Comments: Lines starting with # are used for comments
  - Macros: Besides simple and recursive variables, make also supports macros

Programming in Modern C++ Partha Pratim Das T01.21

There are other options like you can set the directory, you can put that make debugging information, you can have include directory, and so on. So all these are listed here. And if you are one make command is too long, you can continue it with the backslash in and go to the next line. We had talked about this in the CPP also. So it is exactly the same type of functionality that you have here.

(Refer Time Slide: 37:58)



make Utility: Capabilities and Derivatives

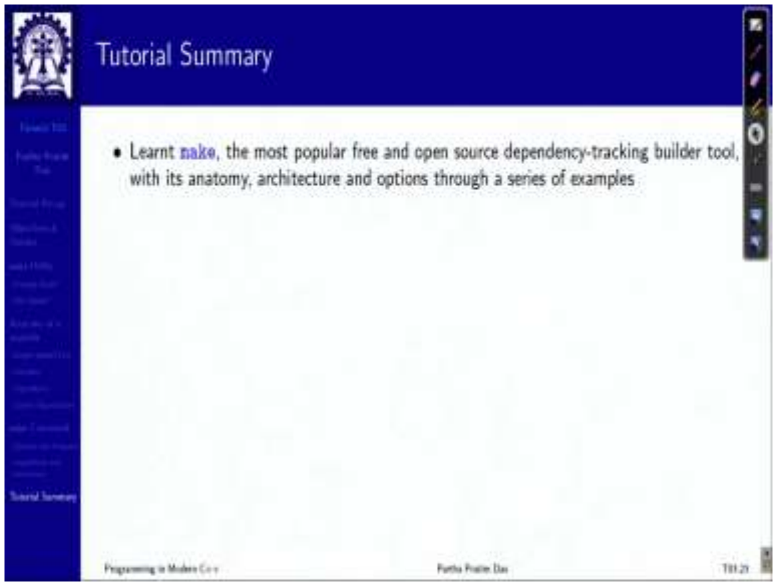
- make was created by Stuart Feldman in April 1976 at Bell Labs and included in Unix since PWB/UNIX 1.0. He received the 2003 ACM Software System Award for make
- make is one of the most popular build utilities having the following major **Capabilities**:
  - make enables the end user to *build and install a package* without knowing the details of how that is done (it is in the makefile supplied by the user)
  - make *figures out automatically which files it needs to update*, based on which source files have changed and *automatically determines the proper order for updating files*
  - make is *not limited to any particular language* - C, C++, Java, and so on.
  - make is *not limited to building a package* - can control installation/uninstallation etc.
- make has several **Derivative** and is available on all OS platforms:
  - GNU Make (all types of Unix): Used to build many software systems, including:
    - GCC, the Linux kernel, Apache OpenOffice, LibreOffice, and Mozilla Firefox
  - Make for Windows, GnuWin32 (We are using here)
  - Microsoft nmake, a command-line tool, part of Visual Studio
  - Kati is Google's replacement of GNU Make, used in Android OS builds. It translates the makefile into Ninja (used for Chrome) for faster incremental builds

All links Accessed on 21-Sep-21
Programming in Modern C++ Partha Pratim Das T03.22

There are a variety of make naturally available, there are different and do not make we are discussing in terms of building projects is not limited to building projects. It can also be used to build any kind of package or can be used for building, installing something uninstall , it is kind of a process to automate what goes on in the system where you have certain conditional things to be written.

So the dependency is basically like writing if else in a different way in the built world. And do not think that make is there only for C C++, though, that is very common, but it is available for any language whatsoever. And there are a variety of make utilities which are available, we are going to use the GNU make, but there is make for Windows earlier known as, there is one from our from Microsoft called nmake and so on, as Chrome specifically has a different one, and so forth.

(Refer Time Slide: 39:06)



Tutorial Summary

- Learnt make, the most popular free and open source dependency-tracking builder tool, with its anatomy, architecture and options through a series of examples

Programming in Modern C++ Partha Pratim Das T03.23

So we have in this series of the third tutorial, we have learned about the make utility, which is a popular free open source dependency tracking builder tool and which can help you really automate your build process and make it efficient and like small examples as I have given and the make files also I would request you to just start using them and get familiar with it and the syntax is not a very friend syntax friendly language, some peculiar dollar at and so on.

So you do not really need to remember everything have a kind of a sample make file with you. And wherever you need you can edit it appropriately and create your respective make file. I hope this tutorial will really help you and take you ahead in the in the automated build process and see you in the next tutorial when we will close about the C C++ build processes.