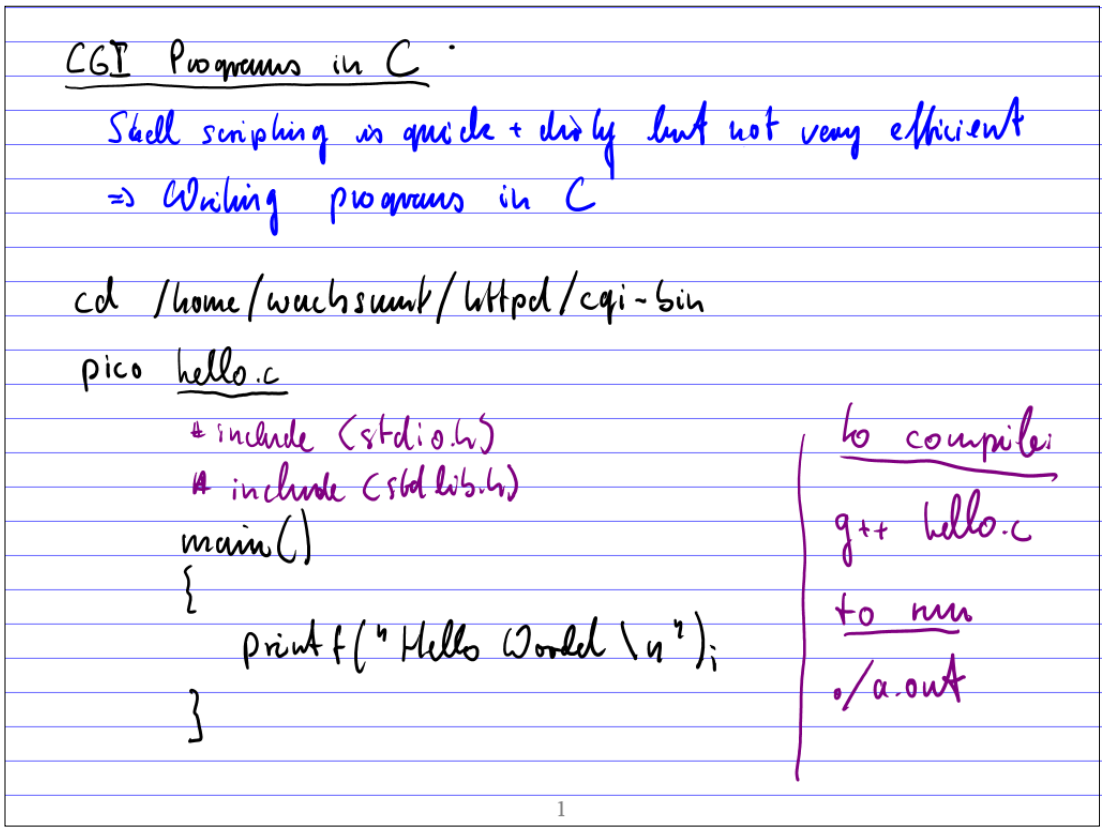
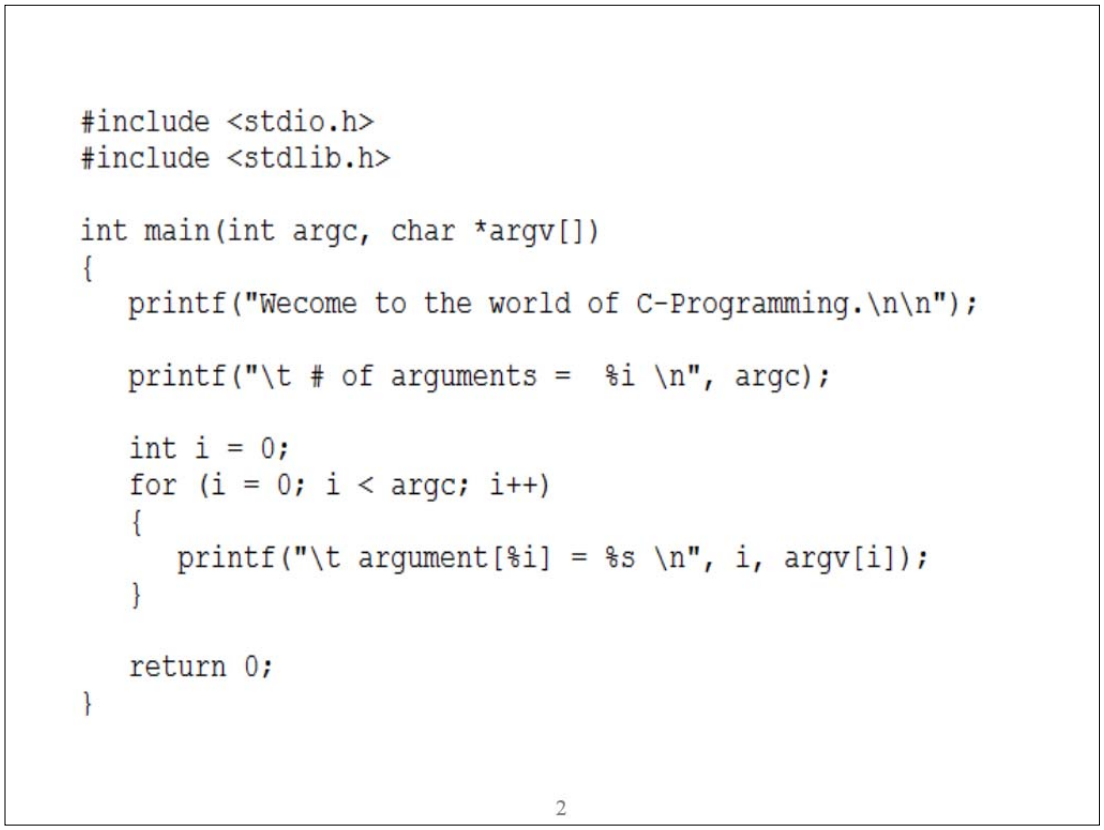
Panel 1



Panel 2



Panel 3

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   char *responses;
   responses = getenv("QUERY_STRING");

   printf("Content-type: text/html\n\n");

   printf("<html><body>");
   printf("<h1>Welcome to the world of C-Programming</h1>\n\n");
   printf("<p># of arguments =  %i</p> \n", argc);
   printf("<ul>");
   int i = 0;
   for (i = 0; i < argc; i++)
   {
      printf("<li>argument[%i] = %s</li>\n", i, argv[i]);
   }
   printf("</ul>");

   printf("Command-line responses: <b>%s</b>", responses);
   printf("</body></html>");

   return 0;
}
```

only works for get forms

3

Panel 4

Describe what this CGI script does and test it:

```
#include <stdio.h>

int incrementcount()
{
  FILE *f;
  int i;

  f=fopen("count.txt", "r+");
  if (!f)
  {
     sleep(1);
     f=fopen("count.txt", "r+");
     if (!f)
       return -1;
  }

  fscanf(f, "%d", &i);
  i++;
  fseek(f,0,SEEK_SET);
  fprintf(f, "%d", i);
  fclose(f);
  return i;
}

int main()
{
  printf("Content-type: text/html\n\n");
  printf("<html>\n");
  printf("<body>\n");
  printf("<h1>The current count is: %d</h1>\n", incrementcount());
  printf("</body>\n");
  printf("</html>\n");
  return 0;
}
```

4

Panel 5

Compiling + Linking

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

provides for output functions such as printf

main method: default starting point

source

More complicated programs are divided into many source code files:

5

Panel 6

Modular Programming

```
#include "hello_lib.h"
int main(void)
{
    say_hello();
    return 0;
}
```

include local header file

hello_main.c

```
void say_hello();
```

hello_lib.h

```
#include <stdio.h>
void say_hello()
{
    printf("hello\n");
}
```

include system header file

hello_lib.c

6

Panel 7

How to Compile Modular Programs

Files: hello_main.c , hello_lib.c , hello_lib.h

To compile: hello_main.c needs hello_lib.h
hello_lib.c needs system libraries

To link: hello_main.o needs hello_lib.o

Command line:

compile
- g++ -c hello_lib.c → makes hello_lib.o
- g++ -c hello_main.c → makes hello_main.o

link
- g++ hello_main.o hello_lib.o -c hello → executable

7

Panel 8

Make

Using the "make" utility you specify which files in your project depend on which and how to build each one.
You leave "when" to 'make'

make requires input file:
uses Makefile if available
else must use -f

Ex: make looks for Makefile
make -f any_file

8

Panel 9

Make file

Each entry in a Make file consists of a

target usually a file to create

dependencies what files the target depends on

commands necessary to make the target

Syntax:

target: [dependencies]

<tab>: command ↵

Ex: hello: hello_main.c hello_lib.c

g++ -c hello_main.c hello_lib.c

g++ hello_main.o hello_lib.o -o hello

9

Panel 10

Bad Make file

```
hello: hello_main.c hello_lib.c
        g++ -c hello_lib.c
        g++ -c hello_main.c
        g++ hello_lib.o hello_main.o -o hello
```

hello depends on *.o files, not *.c !

Good Make file

```
# Simple make file (with comments)

# hello depends on the object files that need to be linked
hello: hello_main.o hello_lib.o
        g++ hello_main.o hello_lib.o -o hello

# the main object file file depends on source and header file
hello_main.o: hello_main.c hello_lib.h
        g++ -c hello_main.c

# the library depends on one source file only
hello_lib.o: hello_lib.c
        g++ -c hello_lib.c
```

10

Panel 11

Better Make files

Make files are frequently used to compile the same project on different computers with different compilers, flags, file names, directories, etc.

=> Use Macros !!!

NAME = definition ← appears at the top of Makefile

Ex: CC = g++ # compiler

instead of g++ --- use

$(CC) --- everywhere

11

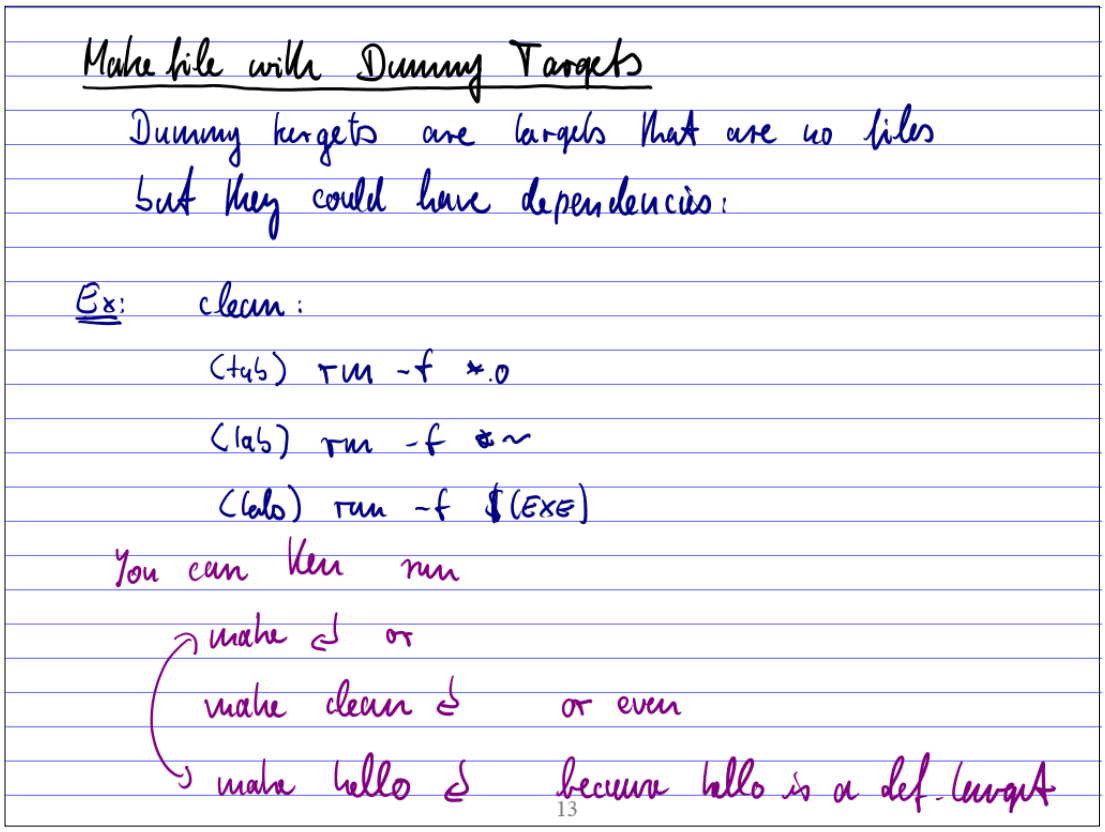
Panel 12

Better Make file

```
# Makefile with macros to create simple "Hello" program

# Defining macros

OBJS = hello_main.o hello_lib.o
EXE = hello
CC = g++

DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

# Defining targets and commands

hello: $(OBJS)
        $(CC) $(LFLAGS) $(OBJS) -o $(EXE)

hello_lib.o: hello_lib.c
        $(CC) $(CFLAGS) hello_lib.c

hello_main.o: hello_main.c hello_lib.h
        $(CC) $(CFLAGS) hello_main.c
```

12

Panel 13

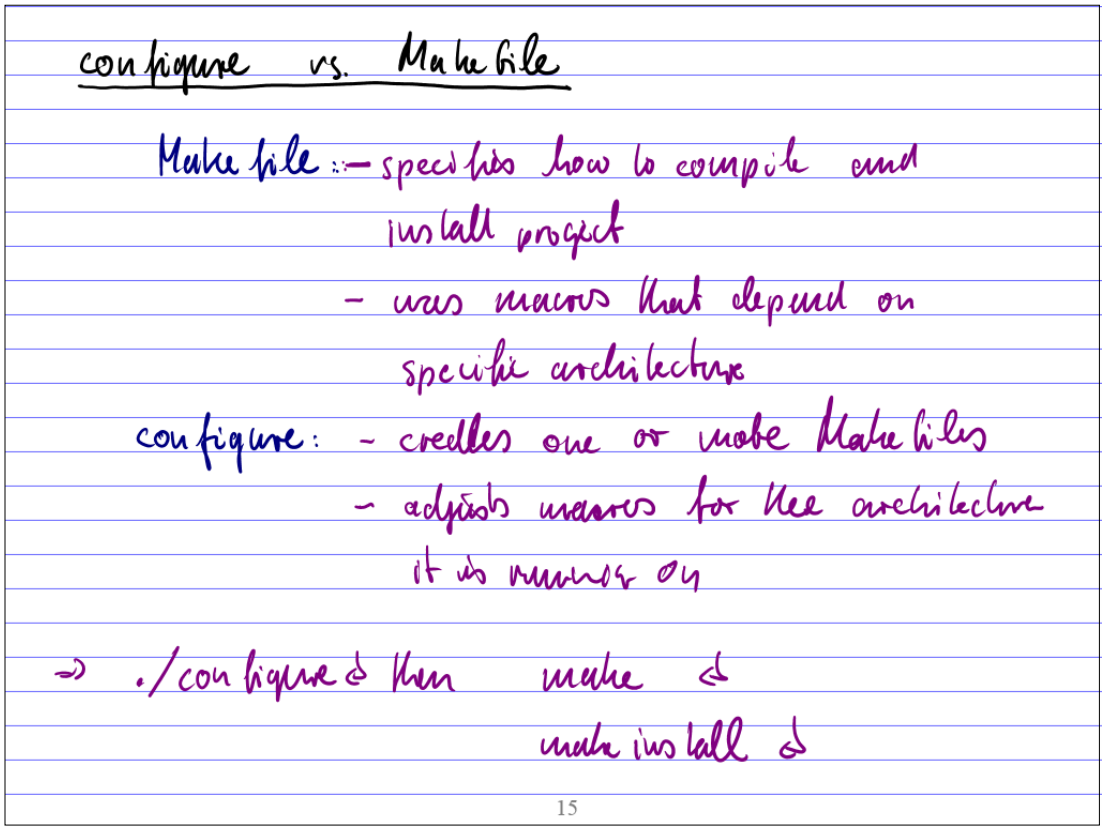


Panel 14

```
# Makefile to crete simple "Hello" program

# defining macros
OBJS = hello_main.o hello_lib.o
EXE = hello
CC = g++

DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

# Defining real targets for creating executable

hello: $(OBJS)
        $(CC) $(LFLAGS) $(OBJS) -o $(EXE)

hello_lib.o: hello_lib.c
        $(CC) $(CFLAGS) hello_lib.c

hello_main.o: hello_main.c hello_lib.h
        $(CC) $(CFLAGS) hello_main.c

# Defining dummy target 'all'
all: hello install

# Defining dummy target 'clean'
clean:
        rm -f *.o *~
        rm -f $(EXE) $(EXE).exe

# Defining dummy target 'install'
install:
        mv $(EXE) hello.exe
```

14

Panel 15


configure vs. Make file
Make file :- specifies how to compile and install project
- uses macros that depend on specific architecture
configure: - creates one or more Make files
- adjusts macros for the architecture it is running on
=> ./configure & then make &
make install &
15