Signals, Instruments, and Systems

School of Architecture, Civil and Environmental Engineering

EPFL, SS 2019-2020

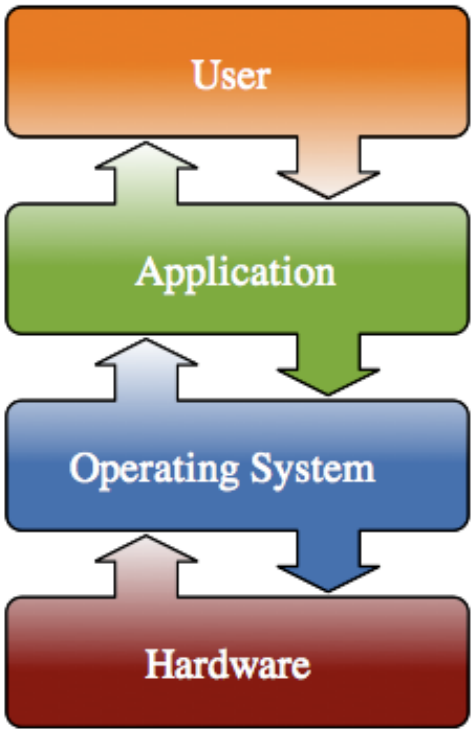
https://disal.epfl.ch/teaching/signals_instruments_systems/

Signals, Instruments, and Systems – W1

C Programming Refresher

Operating System

Interface between hardware and applications. Manages and coordinates activities and sharing of the limited computer resources



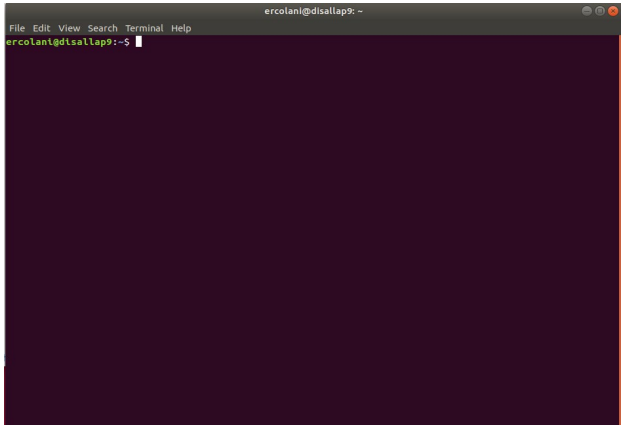
Linux

Source: http://en.wikipedia.org/wiki/Operating_system

Linux Operating System

Standard shell commands (`ls`, `cd`, `pwd`, `cp`, `mv`, `rm`, `mkdir`, `man`...)

- `ls` : lists files and directories
- `cd` : changes working directory (`cd ..` to go one level up)
- `pwd` : prints working directory
- `mkdir` : creates a new directory



Advanced Linux

- Absolute and relative paths
- Environmental variables (`$PATH`, `$HOME`)
- Redirection of input (<) and output (>)
  - `ls -al > list` : command `ls -al` will re-direct its output to a file named "list"
- Pipes (|): the output of the first command is used as input of the second
  - `ls -l | grep log` : this command prints all files in the local directory that have "log" in their name

Remember `man`?

- `man` shows information about functions in the standard libraries of C:
  - e.g. `man printf`
  - or `man atan2`
- To type in the terminal!

Compiled VS interpreted languages

Compiled Language

Code is directly translated (through a **compiler**) into binary that can be executed by the machine.

- ✓ Direct access machine resources (memory, processes)
- ✓ More efficient
- ✓ Faster

Examples: C, C++

Interpreted Language

Need for a running engine (e.g., JVM for Java codes) to be translated to binary (**usually at run-time**).

- ✓ Usually platform independent
- ✓ Easier to code

Examples: Java, Python, Matlab

Main differences between C and Matlab

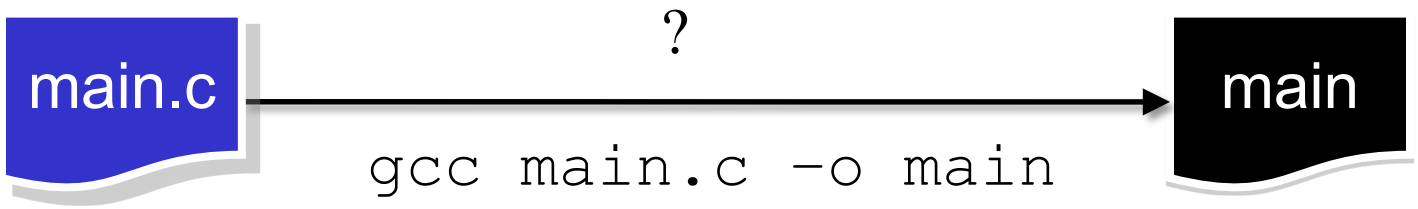
- Matlab is an interpreted language
- Matlab is optimized for matrix operations
- Syntax differences (loops, functions, etc…)
- No variable declarations

Object Oriented Languages

- Object Oriented programming = data is grouped into “objects” that can have properties and/or methods (functions)

- C is NOT an object-oriented language, C++ is

Compilation

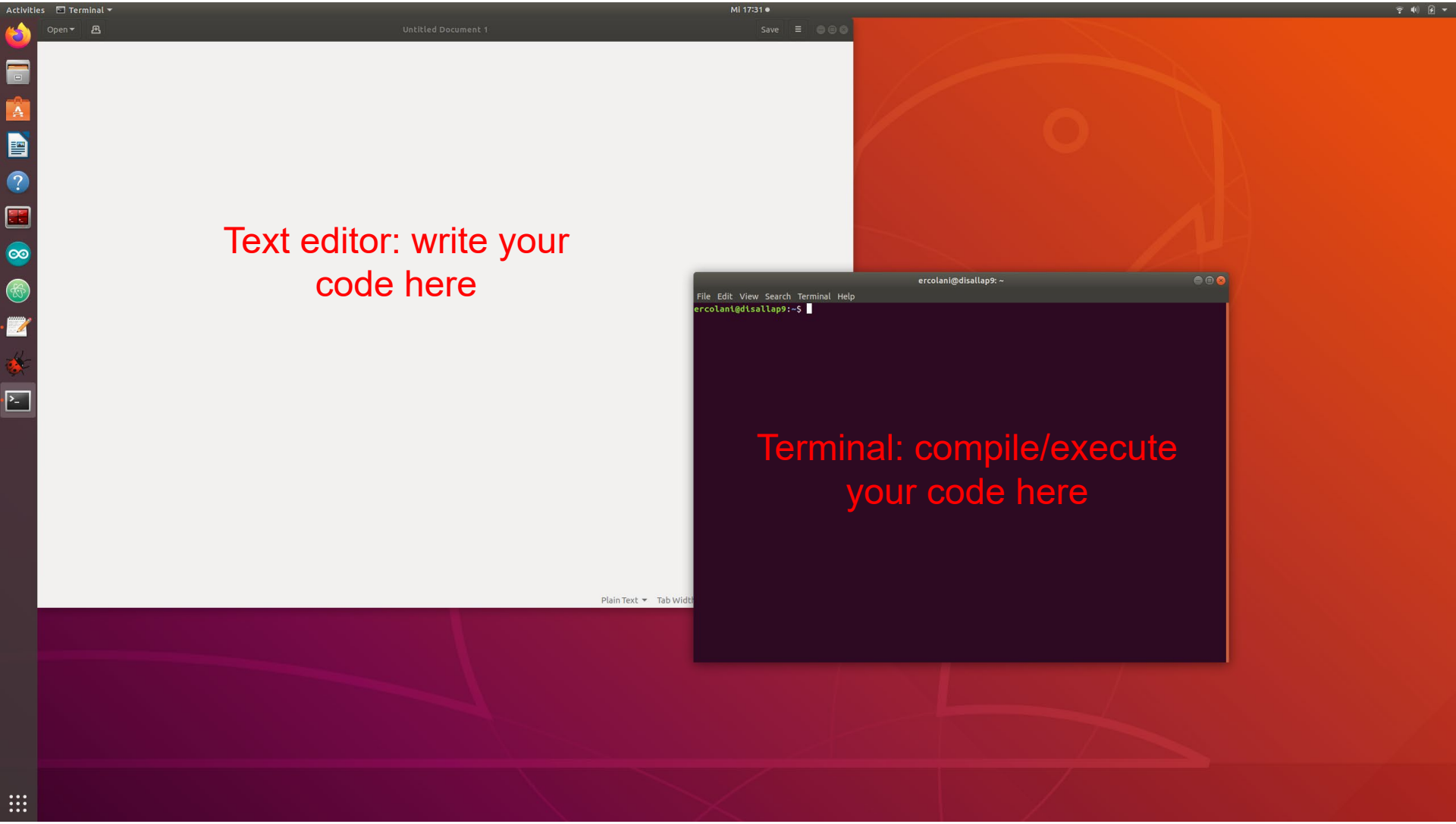


source file

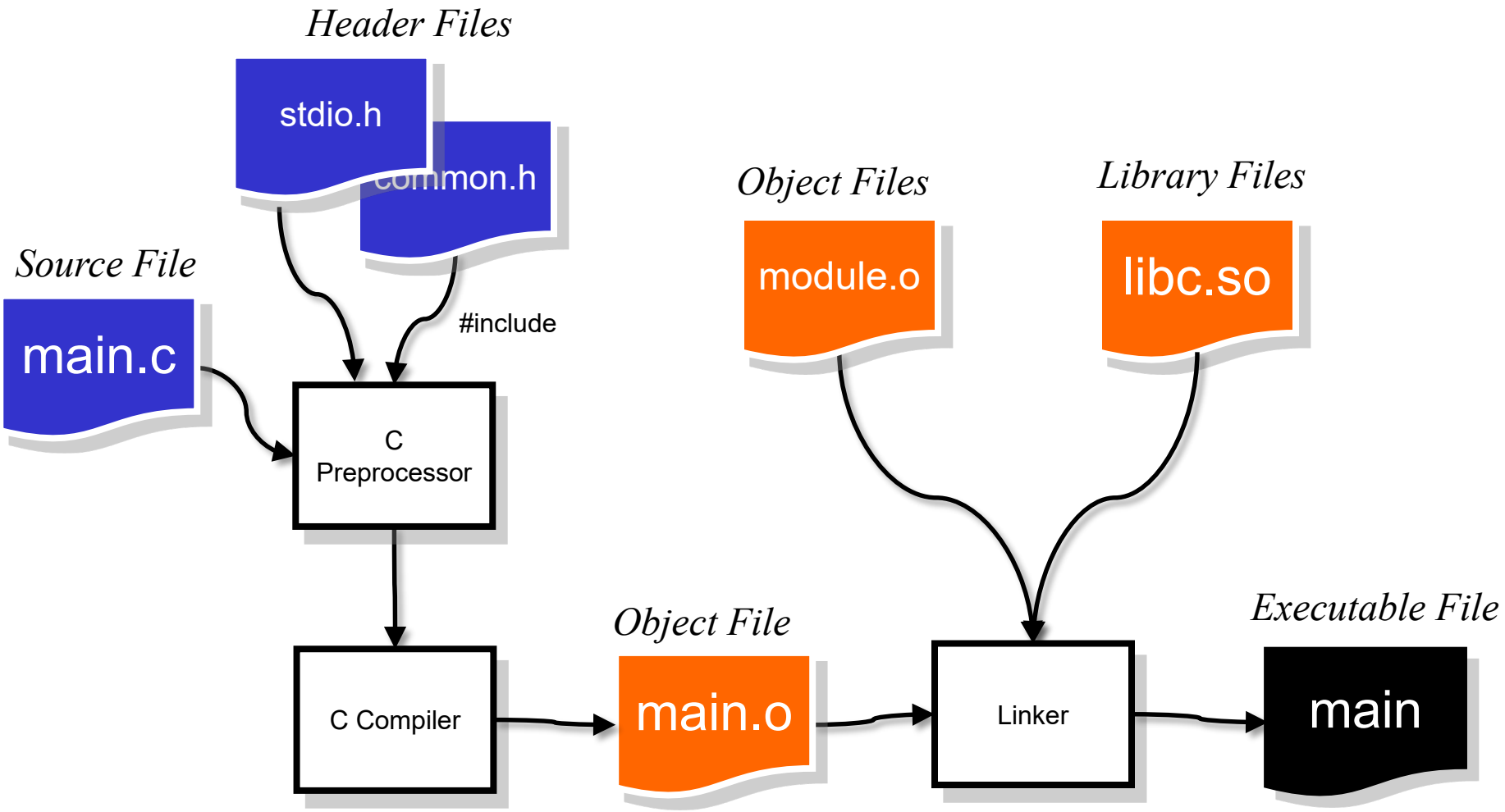
executable (binary)

```
int main() {
  int a = 5;
  double b = 4.3;
  return a * b;
}
```

```
10100101010010
10100101001010
10010100101001
00010101001111
00100101010100
```


Activities
Terminal
Mi 17:31
Open
Untitled Document 1
Save
Text editor: write your code here
Plain Text
ercolani@disallap9: ~
File Edit View Search Terminal Help
ercolani@disallap9:~$
Terminal: compile/execute your code here

The C Compiler Pipeline



“.h” vs. “.c”

- Usually header files (“.h” files) should contain all the necessary functions, structures, typedef and enum **declarations** such that **another** programmer can use your code without having to look at your c file.
- C files contain the actual implementation and “hidden” declarations.

Libraries

- Libraries provide special functionality in the form of collections of ready-made functions:

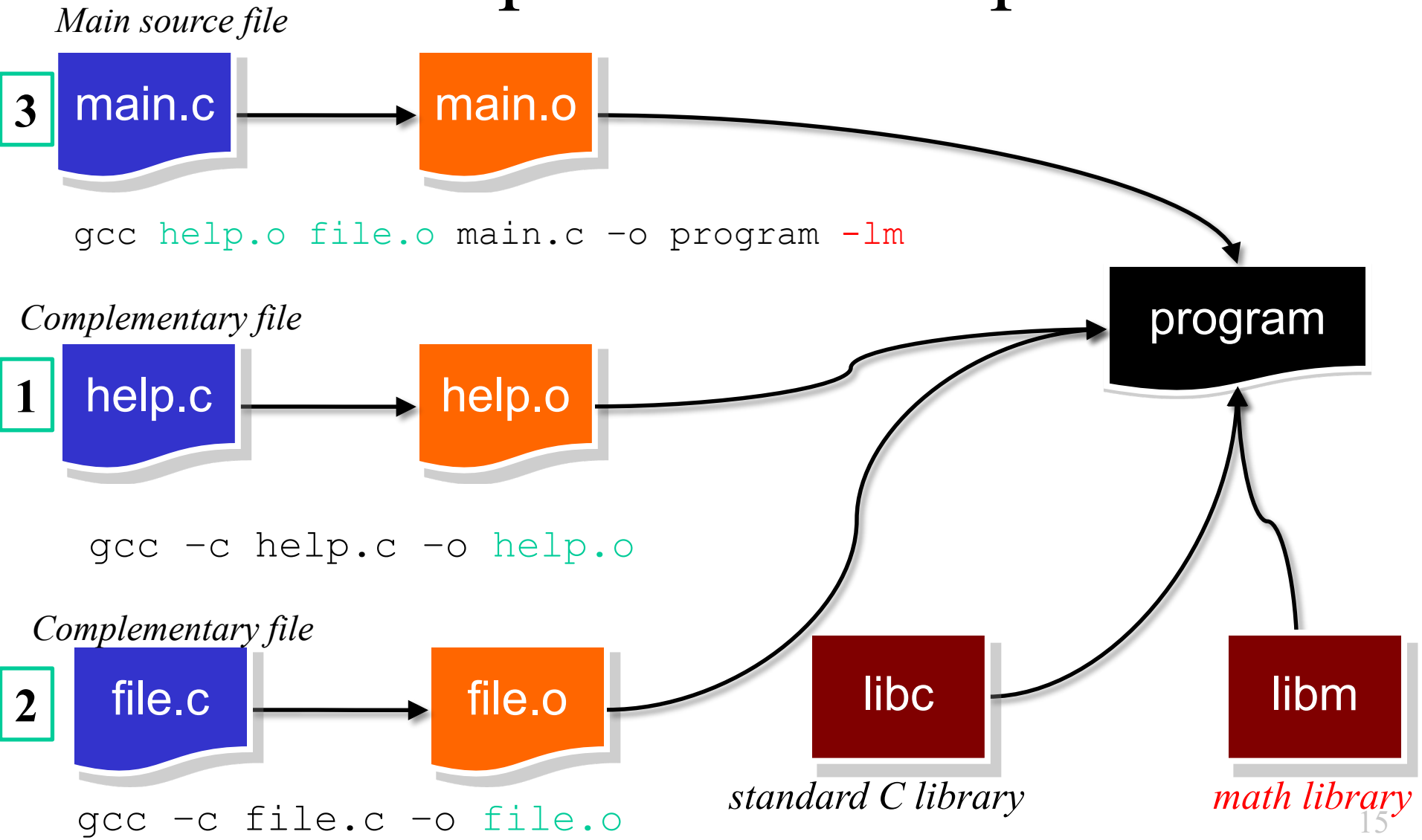
Library:	**Example:**
`stdio.h`	`printf(const char* format,…)`
`math.h`	`sqrt(double x)`
`time.h`	`gettimeofday()`
`stdlib.h`	`rand()`

Usage:

```
#include <stdlib.h>
#include "my_library.h"
```

: your own collection of function declarations

Compilation Example



Makefile: Example

```
CC = gcc
LDLIBS = -lm
all: program

program: main.o help.o file.o



clean:
    rm -rf *.o program
```

- compiler
- additional library
- targets

- label

- [TAB] !!
  `[TAB]rm -rf *.o main`

Note: Run `make clean all` for a totally new compilation

Variables

- There are several types of variables in C:
  - Integers: `char, short, int, long`
  - Floating point: `float, double`
  - Characters: `char`
  - There is no Boolean nor String type.
- Variables can be either signed or unsigned:
  - Ex.: `unsigned int, signed char`

Variables

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Variables

- Variables need to be declared before they are used.
- In correct ANSI C89, variables need to be declared at the beginning of a block (after "{").

```
int main() {
  int a = 5;
  double b = 4.3;
  return (int)((double)a * b);
}
```

Remark: A variable type can be modified with a *cast*.

Variables - Cast

- The cast operator is an operator which forces a particular type mould or type cast onto a value:

```
char ch = 'a';
int i;
i = (int)ch;
```

i = 97

ASCII for 'a'

```
int i = 1;
int j = 3;
double k;
k = i/j;
```

k = 0

careful!

```
int i = 1;
int j = 3;
double k;
k = (double)i/
    (double)j;
```

k = 0.3333

Variables - `enum`

- Enumeration of tags. The tags are numbered (0,1,..) by default.

Def.:

```
enum identifier { enumerator-list }
```

Ex.:

```
enum BOOLEAN {
   false = 0,
   true  =1,
};
enum BOOLEAN is_empty = false;
```

Variables - `typedef`

- Typedef allows you to define your own types.

Def.:

```
typedef type typedef-name
```

Ex.:

```
typedef int BOOLEAN;
BOOLEAN is_empty = 0;
```

Controlling the execution flow

- Algorithms are all about controlling the execution flow.
- Algorithms are all about deciding how to proceed depending on some conditional statements.

IF Operator

If **condition** is verified, do **something**, otherwise do **something else**

Example: find maximum between two values

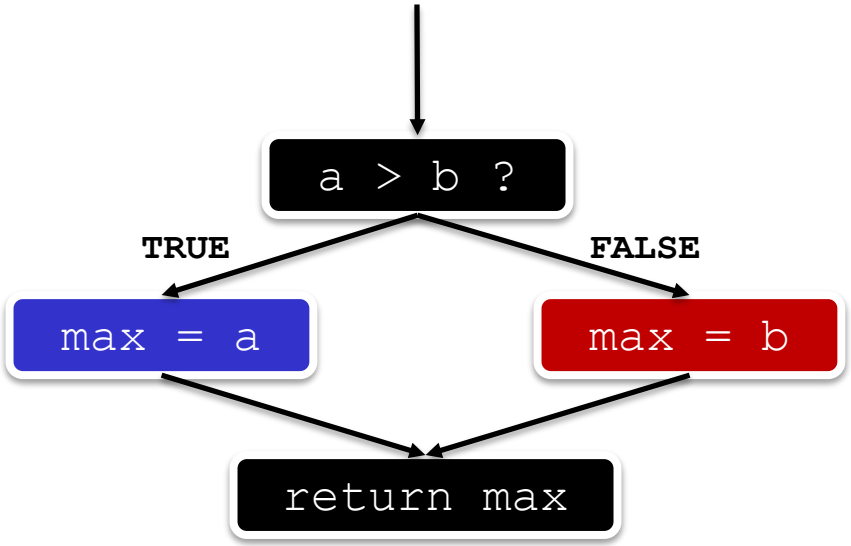
Condition : a>b?

Something: max = a

Something else : max = b

IF Operator

```
float max = 0.0;
float a = 5.0;
float b = 2.1;

if (a > b) {
  max = a;
} else {
  max = b;
}

return max;
```



Conditions

- Conditions can be expressed using logical expressions:
  - `>` (greater than)
  - `<` (less than)
  - `>=` (for greater than or equal to)
  - `<=` (for less than or equal to)
  - `!=` (not equal)
  - `==` (to test for equality)

- In C, there is no boolean variable (`true` or `false`). Instead, `true` is represented by any value not equal to 0 and `false` is represented by the value 0.

Conditions

do not confuse `a == 1` (equality) with `a = 1` (assignment)

WRONG

```
int a = 0;

if (a = 1) {
   // this code is reached
} else {
   // this won't happen
}
```

CORRECT

```
int a = 0;

if (a == 1) {
   // this won't happen
} else {
   // this code is reached
}
```

Conditional branches

- The `switch` structure is very useful when the execution flow depends on the value of a single integral variable (int, char, short, long).

```
switch (a) {
   case 1:
   {
      // if a == 1, do this
      break; // jump to the rest of the code
   }
   case 2:
   {
      // if a == 2, do this
      break; // jump to the rest of the code
   }
   default:
   {
         // otherwise, do this
   }
}
// rest of the code
```

Do not forget the `break` instructions, otherwise the statements in the rest of the `switch` will also be executed!

Conditional branches

```
switch (a) {
   case 1:
   {
      // if a == 1, do this
      break; // jump to the rest of the code
   }
   case 2:
   {
      // if a == 2, do this
      break; // jump to the rest of the code
   }
   default:
   {
         // otherwise, do this
   }
}
// rest of the code
```

```
if (a == 1) {
   // if a == 1, do this
} else if (a == 2) {
   // if a == 2, do this
} else {
   // otherwise, do this
}
// rest of the code
```

Both codes have exactly the same behavior!

Conditional loops

Conditional loops are a way of repeating one or more steps depending on a condition.

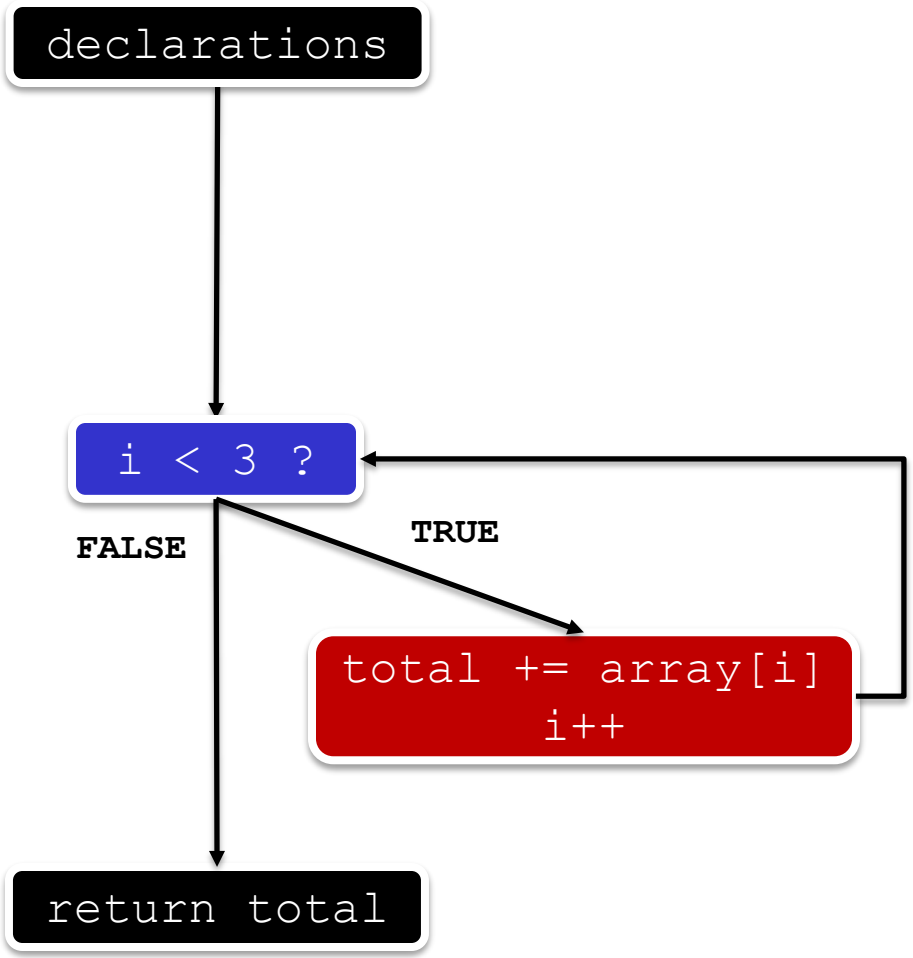
Three types of conditional loops:

- While
- Do-While
- For

While loop

Check condition **BEFORE** executing the loop

```
int total = 0;
int i = 0;
int array[3] = {12,3,-5};

while (i < 3) {
    total += array[i];
    i++;
}

return  total;
```



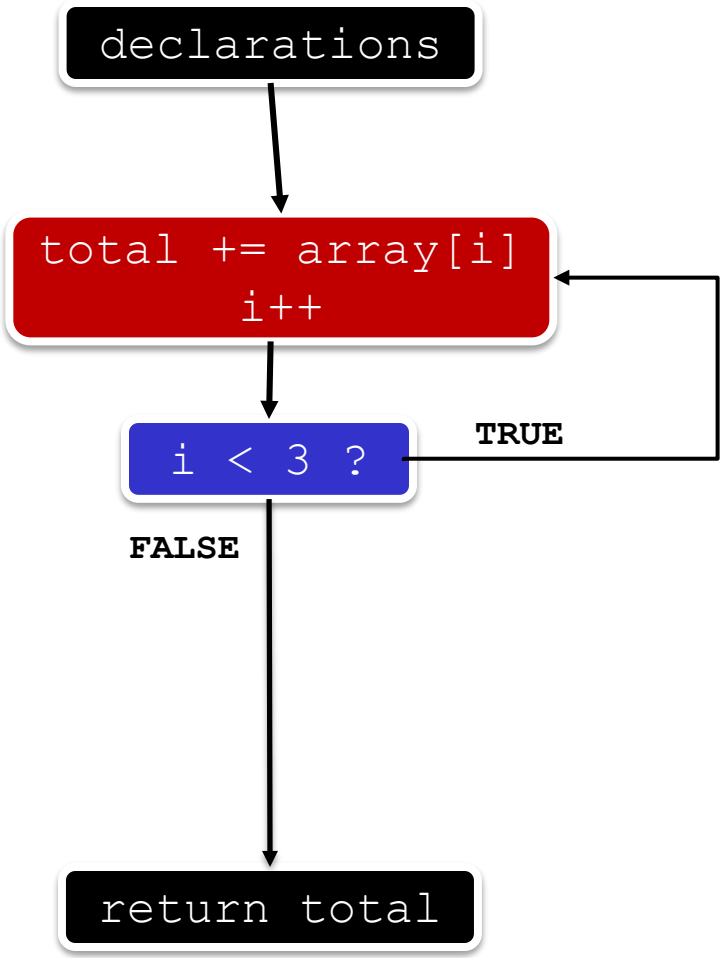
What is the value of total at the end of the program?

total = 10

Do-while loop

Check condition **AFTER** executing the loop. The code is executed at least once

```
int total = 0;
int i = 0;
int array[3] = {12,3,-5};

do{
    total += array[i];
    i++;
} while (i < 3);

return  total;
```



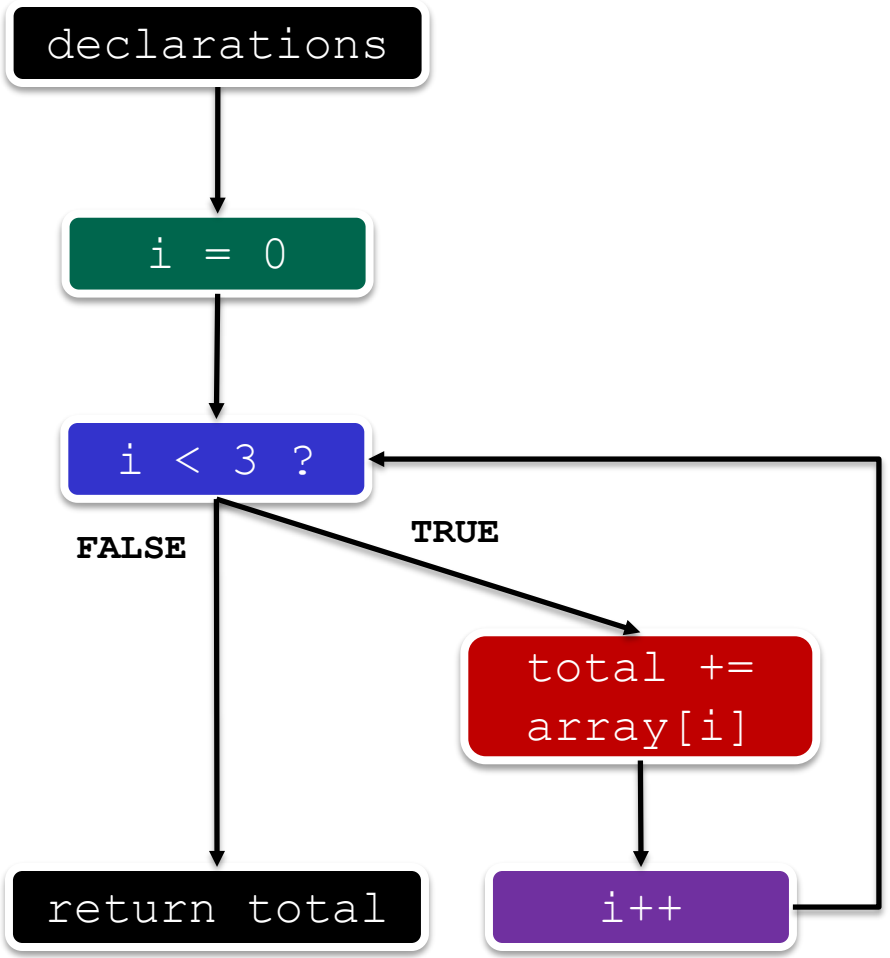
What is the value of total at the end of the program?

total = 10

For loop

The loop `for` is useful when an iteration count (`i` in the example below) needs to be maintained, but the number of iterations **must be known**.

```
int total = 0;
int i = 0;
int array[3] = {12,3,-5};

for (i = 0; i < 3; i++) {
    total += array[i];
}

return total;
```



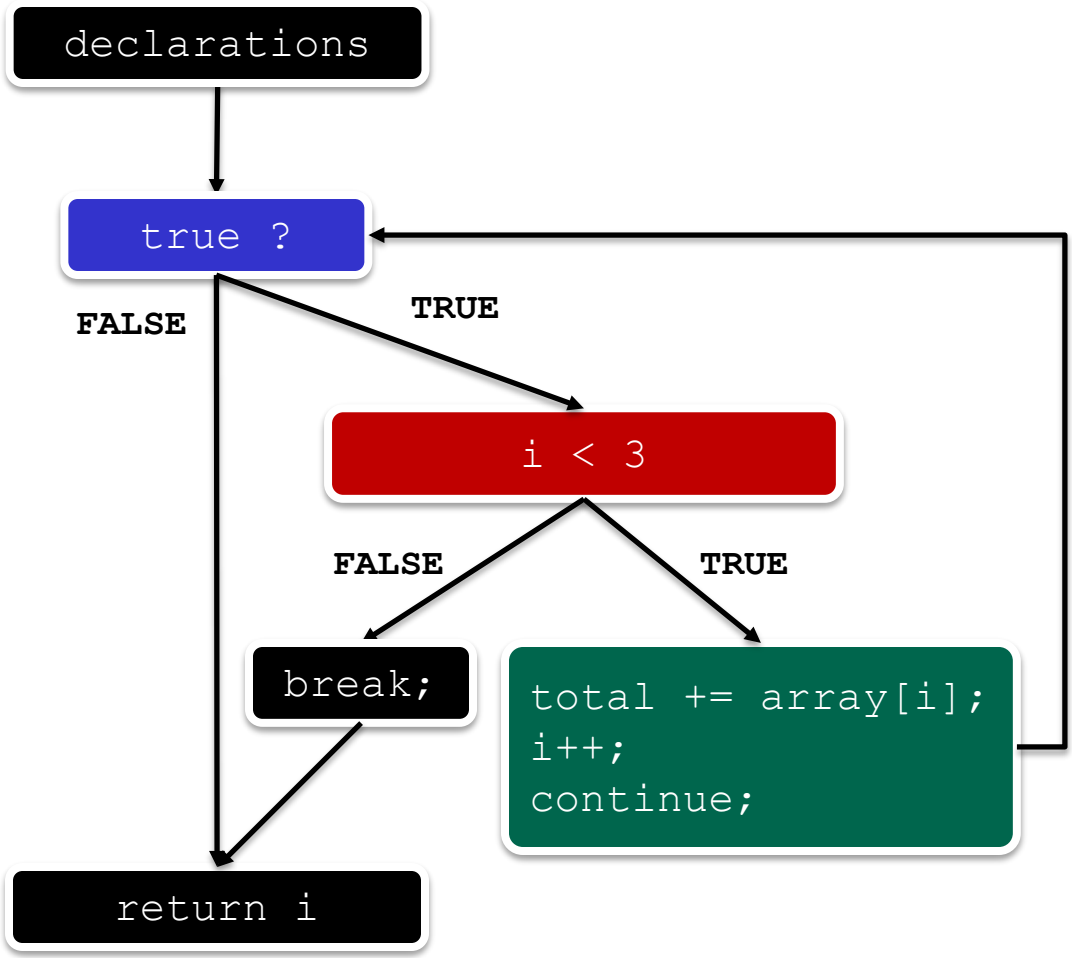
What is the value of total at the end of the program?

total = 10

`break` and `continue`

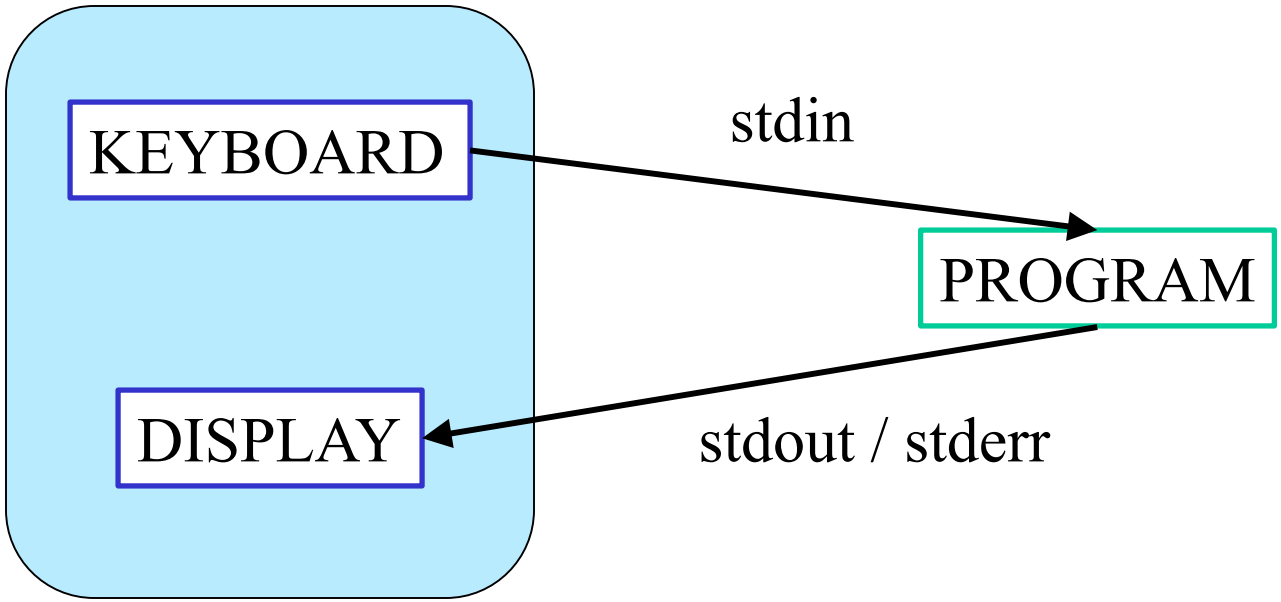
The statements `break` and `continue` cause the program to exit a loop or to jump directly to its next iteration, respectively.

```
int total = 0;
int i = 0;
int array[3] = {12,3,-5};

while (1) {
    if (i < 3) {
        total += array[i];
        i++;
        continue;
    } else {
        break;
    }
    // unreachable code
    printf("code\n");
}

return  i;
```



stdin - stdout

- Standard input and standard output are file streams
- `printf("Hello")` is equivalent to `fprintf(stdout,"Hello")`
- `scanf("%d", &i)` is equivalent to `fscanf(stdin,"%d", &i)`
- `stderr` is useful to display errors, because it is not affected by redirection `(>)`:
  `./my_program > out.txt` -- the errors will still be displayed!



Printing – Format

Most commonly used:

%d	integer
%u	unsigned integer
%f	double
%s	string
%c	char

Object	Control spec.	Actual output
42	%6d	`|    42|`
‘z’	%3c	`|  z|`
2.71828	%10f	`|  2.71828|`
2.71828	%10.2f	`|      2.71|`
“printf”	%s	`|printf|`

Indentation

- Indentation and spacing helps you and others read your code.
- It has to be **systematic** and **consistent**.

```
int main() {
  int i, j;
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      if (i == j){
        printf("%d\n", i);
      }
    }
  }
  return 0;
}
```

VS

```
int main() {
int i, j;
for (i =0; i< 3; i++)
{
for (j= 0; j < 3; ++j) {
if (i ==j){
printf("%d\n",i);}}}
return 0;
}
```

Complete Example

- Module for representing, adding and subtracting complex numbers
- Manual compilation vs. Makefile
- Compilation errors
- See the archive on Moodle

Operators

Operators - Logical

Operator	Meaning
a < b	less than
a <= b	less than or equal
a > b	greater than
a >= b	greater than or equal
a == b	equal to
a != b	not equal to
a && b	logical AND
a || b	logical OR

Operators - Arithmetic

Operator	Meaning
a + b	addition
a - b	subtraction
a * b	multiplication
a / b	division
a % b	modulo (integer remainder)

Operators - Shortcuts

Operator	Meaning
a += b	addition
a -= b	subtraction
a *= b	multiplication
a /= b	division
a %= b	modulo (integer remainder)

Operators - Unary

Operator	Meaning
a++	postfix increment
++a	prefix increment
a--	postfix decrement
--a	prefix decrement

Operators - Unary

```
#include <stdio.h>

int main() {
  int i = 0;
  while (i++ < 3) {
    printf("iteration %d\n", i);
  }

  return 0;
}
```

Operators - Unary

```
#include <stdio.h>

int main() {
  int i = 0;
  while (++i < 3) {
    printf("iteration %d\n", i);
  }

  return 0;
}
```

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR
~a	bitwise NOT

Bitwise calculus is used in advanced code (compression, encryption, or optimizations) and also in embedded systems

Binary numbers

0	0	1	0	1	0	1	1

Binary numbers

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	0	1	1

Binary numbers

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	1	0	1	0	1	1

Binary numbers

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
*	*	*	*	*	*	*	*
0	0	1	0	1	0	1	1
=	=	=	=	=	=	=	=
0	0	32	0	8	0	2	1

Binary numbers

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	1	0	1	0	1	1

$0 + 0 + 32 + 0 + 8 + 0 + 2 + 1 = 43$

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:

20 << 2 = ?

20 >> 2 = ?

20 | 6 = ?

20 & 6 = ?

20 ^ 6 = ?

Hint:

10100 << 2 = 1010000

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:

20 << 2 = ?

20 >> 2 = ?

20 | 6 = ?

20 & 6 = ?

20 ^ 6 = ?

Hint:

10100 << 2 = 1010000

10100 >> 2 = (00)101

Answer:

80

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	5
20 | 6 = ?	10100 | 00110 = 10110	
20 & 6 = ?		
20 ^ 6 = ?		

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	5
20 | 6 = ?	10100 | 00110 = 10110	
20 & 6 = ?		
20 ^ 6 = ?		

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	5
20 | 6 = ?	10100 | 00110 = 10110	22
20 & 6 = ?	10100 & 00110 = 00100	
20 ^ 6 = ?		

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	5
20 | 6 = ?	10100 | 00110 = 10110	22
20 & 6 = ?	10100 & 00110 = 00100	4
20 ^ 6 = ?	10100 ^ 00110 = 10010	

Operators - Bitwise

Operator	Meaning
a << b	left shift
a >> b	right shift
a | b	bitwise OR
a & b	bitwise AND
a ^ b	exclusive OR

Question:	Hint:	Answer:
20 << 2 = ?	10100 << 2 = 1010000	80
20 >> 2 = ?	10100 >> 2 = (00)101	5
20 | 6 = ?	10100 | 00110 = 10110	22
20 & 6 = ?	10100 & 00110 = 00100	4
20 ^ 6 = ?	10100 ^ 00110 = 10010	18

Conclusion

Summary

- Basics in Linux:
  - How to navigate, how to manipulate files
  - How to combine commands
  - Where to look for information (man pages)
- Refresher on C programming:
  - Compilation
  - Variables and main function
  - Basic control structures (if, case, while, for)

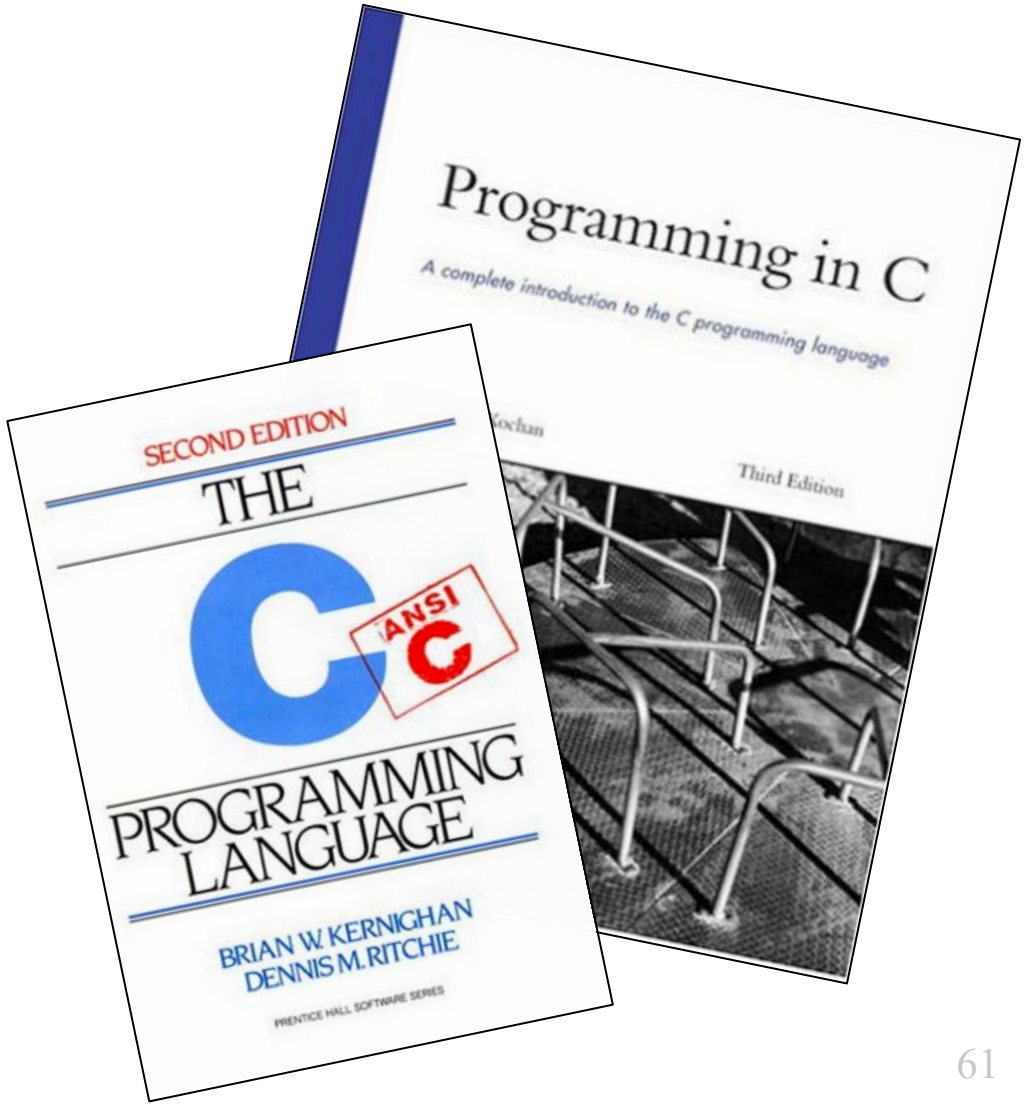
Additional Literature – Week 1

Programming in C
Stephen G. Koch

ISBN-13: 978-0672326660

C Programming Language
Brian W. Kernighan,
Dennis M. Ritchie

ISBN-13: 978-0131103627



Additional Literature – Week 1

- Nice summary on linux shell usage: http://linuxcommand.org/learning_the_shell.php
- Popular C link:

  http://www.c-faq.com/
- And many more – Google is your friend