Introduction to C
Day 2

Katia Bulekova

Research Computing Services

Control Flow: *functions*

Operations that might need to be repeated several times or a logically complete code are organized as functions.

Benefits:

- improves readability of the code
- improves reusability
- helps debugging
- reduces the size of the code

Control Flow: *functions*

```
float dist( float x, float y) {

  float dist;
  dist = sqrt( x*x + y*y );

  return dist;
}
```

Control Flow: *functions*

```
float dist( float x, float y) {

  float d;
  d = sqrt( x*x + y*y );

  return d;
}
```

The declaration of the function should correspond to the value it returns

Control Flow: *functions*

```
float dist( float x, float y) {...}

int main() {
  float a=3.0, b=4.0, c;

  c = dist(a, b);
  printf ( "c = %f\n", c);

  return 0;
}
```

The declaration of the function should happen *before* it is used!

Control Flow: *functions*

Compile:

```
$ gcc -lm -o mydist bc_09_mydist.c
```

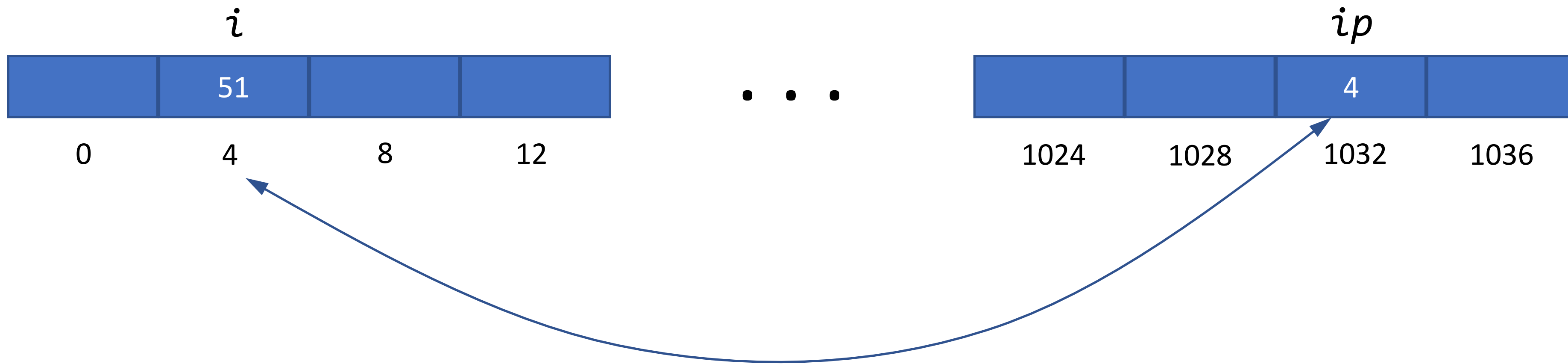
Hands-on exercises:

Write your own function that calculates the average value of x and y.

Call this function from the main() and print the result.

Pointers

A pointer in C a variable whose value is the memory address of another variable.




```
int i = 51;      // regular integer variable
int *ip = &i;    // pointer to an integer variable
```

Pointers

```
int i;     // regular integer variable
int *ip;   // pointer to an integer variable

i = 51;
ip = &i;   // get address of i and store it in ip

printf( "address of i is %p\n", &i);

printf( "value stored at the address %p is %d\n", ip, *ip);
```

Pointers

```
$ gcc –o pointer bc_10_pointer.c

$ ./pointer
```

Pointers

Notice:

Here we use * to declare that ip is a pointer to an integer value

```
int *ip;

printf( "value stored at the address %p is %d\n", ip, *ip);
```

Here we use * to get the value stored at the address of ip

Pointers

Hands-on exercises:

Declare 3 variables: *char*, *int* and *double* and assign values to them.

For each variable assign a variable that points to it.

Using *sizeof()* function find the memory size that is required to store each variable

Pointers

Hands-on exercises:

Declare int variable i and a pointer ip to it. Use ip variable to modify the value stored in i – increase the value by 1.

Pointers – common mistakes

```
float x = 3;
float *p;

// p is a pointer (address) but x is not
p = x;    // Error

// &x is address but *p is not
*p = &x; // Error

// both &x and p are addresses
p = &x;  // Correct

// both x and *p are values
*p = x;  // Correct
```

Pointers and arrays

The name of an array can be treated as a pointer and vice versa

```
float x[5] = { 1.0, 2.1, 3.2, 4.3, 5.4};
float *p = x; // p will point to x[0]

p = p + 2;

printf ("p points to %d\n", *p); // prints 3.2
```

So `x[2]` and `*(p+2)` refer to the same element

Pointers and arrays

Compile and run:

```
$ gcc –o parrays bc_11_parrays.c

$ ./parrays
```

Pointers and Strings

A C string is an array of characters
So, the name of the string variable is also a pointer:

```
char my_string[] = "Hello";

or

char my_string[] = {'H', 'e', 'l', 'l', 'o' , '\0’};

printf("third letter in the string is %c\n", *(my_string + 2));
```

Memory allocation

Suppose we do not know how many values we need to store in an array. We can then first declare it as a pointer and allocate memory later.

`malloc` `( size_in_bytes )` returns a pointer to a memory region that you can then assign to a variable of whatever type you like

To free the memory use ***`free()`*** function

Memory allocation

Suppose we do not know how many values we need to store in an array. We can then first declare it as a pointer and allocate memory later:

```
#include <stdlib.h>

float *x;

//allocate memory to store 10 elements
x = (float*) malloc ( 10 * sizeof(float) );

// free memory
free(x);
```

Memory allocation

Hands-on exercises:

Use `mem.c` file

Declare int array with unknown size (pointer).

Using `malloc()` function allocate 100 elements to be stored in this array.

Using for loop fill array with integer values from 1 to 100.

Command line arguments

Input parameters to the program can be passed using command line:

```
$ my_program 100
```

The number of arguments that were passed to the program and their values can be accessed using argc and argv variables:

```
int main (int argc, char **argv){

}
```

Command line arguments

argc – is equal to the number of arguments passed to the program.

The first argument is always the program name. So argc is always greater or equal to 1.

argv is an array of strings (hence **). Each string contains a value of an argument.

argv[0] contains the name of the program

```
int main (int argc, char **argv){

}
```

Command line arguments

To convert values stored in argv to integers or floating values, use `atoi()` and `atof()` functions respectfully:

```
i = atoi( argv[1] );
```

or

```
x = atof ( argv[2] );
```

Command line argument

Hands-on exercises:

Compile the program:

```
$ gcc –o mem bc_13_args.c
```

Run passing various integer values to the program:

```
$ ./mem 20
```

Putting it all together

- Let's define a function that takes a value N, allocates an integer array, fills it with values from 1 to N and returns a pointer to this array.
- Define another function mean() that takes array as an argument and calculates mean (average) of this array.
- Save both functions in a file with a name mean_fun.c
- In the main.c file define main() function. This function will read the command line and assign the input to N and then it will call first function array_init() and then mean(). Print the result at the end.

Putting it all together

Hands-on exercises:

Compile the program and run passing various integer values to the program:

```
$ gcc -c bc_14_mean_fun.c
$ gcc -c bc_14_ main.c
$ gcc -o mean bc_14_ main.o bc_14_ mean_fun.c
```

Automating the build process with GNU Make

The manual build process we used above can become quite tedious for all but the smallest projects. There are many ways that we might automate this process. The simplest would be to write a shell script that runs the build commands each time we invoke it:

```
#!/bin/bash

gcc -c hello.c
gcc hello.o -o hello
```

Automating the build process with GNU Make

Make is a mini-programming language unto itself. For the hello program, a Makefile might look like this:

```
all: hello

hello: hello.o
    gcc hello.o -o hello

hello.o: hello.c
    gcc -c hello.c

clean:
    rm hello hello.o
```

Automating the build process with GNU Make

You can issue "`make --help`" to list the command-line options; or "`man make`" to display the man pages.

The command make looks for a file named `Makefile` or `makefile` in the same directory by default. Other file names can be specified by the option -f:

```
make -f filename
```

Automating the build process with GNU Make

A `Makefile` consists of a set of rules.

A rule consists of 3 parts: a *target*, a list of *prerequisites* and a *command*:

```
target: prereq1 prereq2 ...
	command
```

The target and pre-requisites are separated by a colon (:).

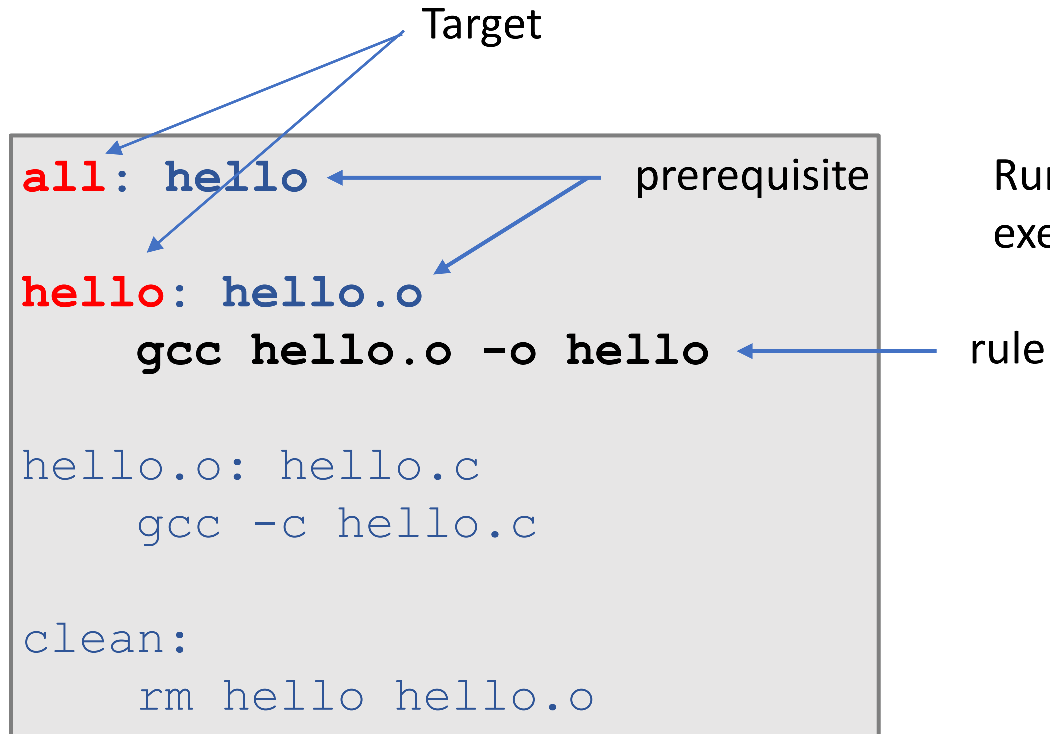
The command **must be preceded by a *tab*** (NOT spaces).

Automating the build process with GNU Make

Make is a mini-programming language unto itself. For the hello program, a Makefile might look like this:

```
target: prereq1 prereq2 ...
	command
```

Automating the build process with GNU Make



Running make command without an argument executes the target "all" in the Makefile.

Make workflow

```
all: hello

hello: hello.o
	gcc hello.o -o hello

hello.o: hello.c
	gcc -c hello.c

clean:
	rm hello hello.o
```

1. Find the default target, which is our executable file hello.
2. Check to see if hello is up-to-date. hello does not exist, so it is out-of-date and will have to be built
3. Check to see if the prerequisite hello.o is up-to-date. hello.o does not exist, so it is out-of-date and will have to be built.
4. The prerequisite hello.c is not a target, so there is nothing left to check. The command gcc -c hello.c will be run to build hello.o
5. Now hello.o is up to date, so make builds the next target hello by running the command gcc hello.o -o hello
6. Done.

Make workflow

```
all: hello

hello: hello.o
    gcc hello.o -o hello

hello.o: hello.c
    gcc -c hello.c

clean:
    rm hello hello.o
```

A target is considered out-of-date if:

1. it does not exist, or
2. it is older than any of the prerequisites.

Make workflow

```
all: hello

hello: hello.o
    gcc hello.o -o hello

hello.o: hello.c
    gcc -c hello.c

clean:
    rm hello hello.o
```

Note that the command under the *clean* target is not executed by make, because it is neither the first target nor a prerequisite of any other target. To execute this target, we need to specify the target name:

```
make clean
```

Make example

Let's look at the example for our first multi-file program mean:

```
all: mean

mean: main.o mean_fun.o
        gcc main.o mean_fun.o -o mean

main.o: main.c
        gcc -c main.c

mean_fun.o: mean_fun.c
        gcc -c mean_fun.c

clean:
        rm mean *.o
```

Make example

By default, make prints on the screen all the commands that it executes. To suppress the print, add an @ before the commands, or turn on the silent mode with the option -s: `make -s`

```
all: mean

mean: main.o mean_fun.o
	@ gcc main.o mean_fun.o -o mean

main.o: main.c
	@ gcc -c main.c

mean_fun.o: mean_fun.c
	@ gcc -c mean_fun.c

clean:
	@ rm mean *.o
```

Writing a good Makefile

A Makefile could be very complicated in a practical program with many source files. It is important to write a Makefile in good logic. The text in the Makefile should be as simple and clear as possible.

```
CC=gcc
OBJ=main.o mean_fun.o
EXE=mean

all: $(EXE)

$(EXE): $(OBJ)
	$(CC) main.o mean_fun.o -o mean

main.o: main.c
	$(CC) -c main.c

mean_fun.o: mean_fun.c
	$(CC) -c mean_fun.c

clean:
	rm $(EXE) *.o
```

Writing a good Makefile

Furthermore, we can upgrade the Makefile to a higher automatic level using the so-called "automatic variables":

```
CC=gcc
OBJ=main.o mean_fun.o
EXE=mean

all: $(EXE)

$(EXE): $(OBJ)
	$(CC) $^ -o $@
main.o: main.c
	$(CC) -c $<

mean_fun.o: mean_fun.c
	$(CC) -c $<

clean:
	rm $(EXE) *.o
```

Here we have used the following automatic variables:

$@ --- the name of the current target
$^ --- the names of all the prerequisites
$< --- the name of the first prerequisite

Writing a good Makefile

Both main.o and mean_fun.o are built using the same command. We can simplify the Makefile even further

```
CC=gcc
OBJ=main.o mean_fun.o
EXE=mean

all: $(EXE)

$(EXE): $(OBJ)
	$(CC) $^ -o $@
%.o: %.c
	$(CC) -c $<

clean:
	rm $(EXE) *.o
```

Useful Resources

1. GNU Make Manual: https://www.gnu.org/software/make/manual/
2. GCC and Make: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html#zz-2.
3. Writing Larger Programs
4. Makefile tutorial: https://makefiletutorial.com/

Thank you

Email: help@scc.bu.edu

Research Computing Services