Fall 2021: Computational Science I

Instructor: Mohammad Sarraf Joshaghani
(m.sarraf.j@rice.edu)

Module 4: C programming

history of the c programming language

- created at bell labs by dennis ritchie (1972-1973).
- a following up to the b programming language, which was created by ken thompson in 1969.
- code written in c is generally very **portable**.
- it is a compiled language, meaning that you need a system-dependent compiler to convert your typed computer program to machine instructions (as opposed to matlab in which you just write “scripts” that do not need to be compiled).
- updates to the language: C89, C90, C99, C11, C18. C18, released in 2018, is the current standard.

https://en.wikipedia.org/wiki/C_(programming_language)

what do we need to begin?

a compiler: your ubuntu os should have c compilers installed, which can be run via the commands: `cc` or `gcc`. try typing

```
gcc -v
```

in your terminal to see the version of your compiler and how it is configured.

a compiler will be the tool that we use to convert our *typed code* into *machine instructions*.

a brief detour regarding compilers

gcc refers to the C compiler from the GNU Compiler Collection.

g++ refers to the C++ compiler from the GNU Compiler Collection.

cc is probably the same thing as gcc on linux operating systems.

```
~$ which cc
/usr/bin/cc
~$ ls -l /usr/bin/cc
lrwxrwxrwx 1 root root 20 May 27  2017 /usr/bin/cc -> /etc/alternatives/cc
~$ ls -l /etc/alternatives/cc
lrwxrwxrwx 1 root root 12 May 27  2017 /etc/alternatives/cc -> /usr/bin/gcc
~$ which gcc
/usr/bin/gcc
~$
```

the above terminal commands show that cc is soft-linked to gcc, so either of these commands will do the same thing.

GNU Compiler Collection

this is a collection of compilers for many languages that is open-source.

created by the GNU Project, an open-source software “mission” initiated in 1983 by Richard Stallman.

GNU stands for “GNU’s not Unix,” which is apparently called a *recursive acronym*. Unix was a proprietary os at the time.

`https://en.wikipedia.org/wiki/GNU_Compiler_Collection`

tentative outline

- structure and syntax of a c program
- declaration of variables
- binary representation of a variable
- functions

structure and syntax of a c program

there are generally two types of files that comprise a c programming project:

1. **header** files with an extension "`.h`"
2. **source** files with an extension "`.c`"

header files contain prototypes of functions and structs.

source files contain implementation of functions. one of your source files must have a main function. this is the first function which is executed by the program.

```c
// hello world example
#include <stdio.h>

int main()
{
  printf("%s", "hello world!!\n");
}
```

../example/example_hello_world/hello_world.c

what is in a source file?

the source file contains code of three different flavors:

- preprocessor instructions, or “directives” beginning with #.
- simple instructions ending in a semicolon.
- multiple instructions contained in curly braces { and }.

```
1 // hello world example
2 #include <stdio.h>
3
4 int main()
5 {
6   printf("%s", "hello world!!\n");
7 }
```

../example/example_hello_world/hello_world.c

line 1 contains the inclusion of the standard input/output header file.

line 4 is the beginning of the `main` function.

line 6 calls a function defined in the standard input/output for writing to the terminal.

preprocessor directives

note that you do not need a semicolon after the line `#include blah.h`. this directive actually inserts the contents of the header file `blah.h` directly into the code.

you can also use preprocessor directives to define parameters or functions.

```
 1 // preprocessor directives example
 2 #include <stdio.h>
 3 // a macro that defines a random number
 4 #define A_RANDOM_NUMBER 15.2
 5
 6 int main()
 7 {
 8   // print out the number
 9   printf("%s %f %s", "check out this random number... ",
         A_RANDOM_NUMBER, "\n");
10 }
```

../example/example_preprocessor/preprocessor.c

line 4 is called a macro that defines a parameter `A_RANDOM_NUMBER`. what happens in this case is that before the code is compiled, all instances of `A_RANDOM_NUMBER` in the source code **are replaced** with the value provide in the directive, i.e. 15.2.

c language syntax

- lower case a through z and upper case A through Z
- numbers 0 through 9
- special characters like %, &, #, etc

note that lower and upper case letters are actually different characters in the c alphabet!

letters and numbers you can use in defining names for **variables** and **functions** are

a through z, A though Z, 0 through 9, and underscore “_”

these names can have up to 31 characters and **cannot** start with a number.

keywords

these are very special words in C and cannot be used as names for variables, functions, etc.

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

basic variables types in c

- `char`, a character
- `int`, an integer
- `float`, real single precision
- `double`, real double precision
- `void`, usually used for a function that returns **no value**, like

  ```
  void a_function_example/example(int foo)
  {
  // some implementation
  }
  ```

  or when a function takes **no parameters**

  ```
  int another_function_example/example(void)
  {
  // another implementation
  }
  ```

binary representation of variables

your computer will store variables as sequences of 1's and 0's.

individual 1's and 0's are called **bits**.

"bit" is a portmanteau of "binary digit."

a sequence of 8 bits is called a **byte**.

each data type, `char`, `int`, ... are composed of a fixed number of bytes.

01110101 $\leftarrow$ a single byte!

think about punch cards back in the day... either a hole could be punched or not, corresponding to 0 or 1.

https://en.wikipedia.org/wiki/Bit

a first example: binary representation of integers

binary integer representation corresponds to its base-2 representation.

before we discuss base-2, let us review base-10.

recall that the integer 1532 can be expressed as:

$$1532 = \mathbf{1} \times 10^{\mathbf{3}} + \mathbf{5} \times 10^{\mathbf{2}} + \mathbf{3} \times 10^{\mathbf{1}} + \mathbf{2} \times 10^{\mathbf{0}}$$

the **blue** numbers correspond to the digit in the base-10 representation, and the **red** numbers correspond to the digit location, with the right-most location corresponding to the power zero, moving from right-to-left.

sometimes (usually never), we use a subscript to indicate that the representation of a number is in base-10:

$$(1532)_{10}$$

base-b representation

for a positive integer b, a number represented in base-b would be:

$$(a_N a_{N-1} ... a_0)_b = a_N \times b^N + a_{N-1} \times b^{N-1} + \ldots + a_0 \times b^0$$

where the a_i's are the "digits" and are integers between 0 and $b-1$.

question: given a number in base-10, $(\beta)_{10}$, how do we represent it in base-b for any positive integer b?

$$(\beta)_{10} = a_N \times b^N + a_{N-1} \times b^{N-1} + \ldots + a_0 \times b^0$$

converting from base-10 to base-b

if we assume that $(\beta)_{10} > 0$, we can use the following fact: there exists unique integers q and r so that:

$$(\beta)_{10} = q \times b + r \quad \text{with } 0 \leq r < b.$$

q is called the **quotient** and r is called the **remainder**.

the mod operator in the c language is denoted "%" and will give us the remainder for a given base b, for example (if $b = 2$):

$$11 \,\%\, 2 = 1$$

since $11 = 5 \times 2 + 1$.

the quotient operator denoted "/" will give us the quotient:

$$11 \,/\, 2 = 5$$

so that we can write $11 = (11 \,/\, 2) \times 2 + 11 \,\%\, 2$.

an example converting from base-10 to base-2

suppose we want to express $(1532)_{10}$ in base-2, i.e. we need to identify numbers $a_0, \ldots a_N$, which are either 0 or 1, so that:

$$(1532)_{10} = a_N \times 2^N + a_{N-1} \times 2^{N-1} + \ldots + a_0 \times 2^0,$$

so our number in base-2 is:

$$(1532)_{10} = (a_N a_{N-1} \ldots a_0)_2.$$

Let us assume $N > 1$ (why is this true for this example?). Notice that we can express $(1532)_{10}$ using the quotient and remainder with 2 as the divisor:

$$(1532)_{10} = \underbrace{(a_N \times 2^{N-1} + a_{N-1} \times 2^{N-2} + \ldots + a_1)}_{= \text{quotient}} \times 2 + \underbrace{a_0}_{= \text{remainder}}$$

so we can immediately identify the first digit on the right:

$$a_0 = (1532)_{10} \,\%\, 2$$

an example converting from base-10 to base-2

we can continue this process in the same way to identify a_1:

$$\text{quotient from previous} = a_N \times 2^{N-1} + a_{N-1} \times 2^{N-2} + \ldots + a_1$$

so $a_1 = \text{quotient} \,\%\, 2$.

we can continue this process to identify all the other base-2 digits.

when does this process terminate?

exercise 1: convert $(1005)_{10}$ to base 2.

exercise 2: convert $(100)_{10}$ to base 16.

exercise 3: convert $(2000)_{10}$ to base 3.

exercise 4: convert $(15)_{10}$ to base 16.

exercise 1 worked out

$$\begin{aligned}
1005 &= 502 \times 2 + 1 & \Longrightarrow\ a_0 &= 1 \\
502 &= 251 \times 2 + 0 & \Longrightarrow\ a_1 &= 0 \\
251 &= 125 \times 2 + 1 & \Longrightarrow\ a_2 &= 1 \\
125 &= 62 \times 2 + 1 & \Longrightarrow\ a_3 &= 1 \\
62 &= 31 \times 2 + 0 & \Longrightarrow\ a_4 &= 0 \\
31 &= 15 \times 2 + 1 & \Longrightarrow\ a_5 &= 1 \\
15 &= 7 \times 2 + 1 & \Longrightarrow\ a_6 &= 1 \\
7 &= 3 \times 2 + 1 & \Longrightarrow\ a_7 &= 1 \\
3 &= 1 \times 2 + 1 & \Longrightarrow\ a_8 &= 1 \\
1 &= 0 \times 2 + 1 & \Longrightarrow\ a_9 &= 1
\end{aligned}$$

so we have $(1005)_{10} = (1111101101)_2$

binary representation for integers

type	bytes	bits	value range
`int`	4	32	-2^{31} to $2^{31}-1$
`short int`	2	16	-2^{15} to $2^{15}-1$
`long int`	8	64	-2^{64} to $2^{64}-1$
`unsigned short int`	2	16	0 to $2^{16}-1$
`unsigned long int`	8	32	0 to $2^{32}-1$

if a numeric variable is `unsigned`, then all the bits can be used to represent the numeric value. otherwise, one of the bits refers to the variable's sign.

example showing number of bytes

the `sizeof` function returns the storage size of a given variable type. the returned size is a multiple of the storage size for a `char`, which happens to be 1 byte.

```
#include <stdio.h>

int main(void){

  printf("size of char: %ld bytes \n", sizeof(char));
  printf("size of int: %ld bytes \n", sizeof(int));
  printf("size of float: %ld bytes \n", sizeof(float));
  printf("size of double: %ld bytes \n", sizeof(double));

  printf("size of short int: %ld bytes \n", sizeof(short
      int));
  printf("size of long int: %ld bytes \n", sizeof(long
      int));
  printf("size of long double: %ld bytes \n", sizeof(long
      double));
}
```

../example/example_sizeof/sizeof.c

value ranges for integers

for a signed `int`, 1 bit is used for the sign, so we have 31 bits leftover which are either `0` or `1`. a couting argument would imply that 2^{31} distinct integers can represented with 31 bits, i.e.

$$\begin{aligned}&(\text{2 choices for 2nd bit}) \times (\text{2 choices for 3rd bit})\\ &\quad \times \ldots \times (\text{2 choices for the 32nd bit}) = 2^{31} \text{ distinct integers}\end{aligned}$$

for a positive sign, since one of the possible integers is 0, the maximum positive integer is $2^{31} - 1$.

another possible way to argue this is to figure out the largest possible integer you can represent with 31 bits:

$$2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0 = 2^k - 1 \quad \text{for } k \geq 1.$$

exercise: try showing the above statement inductively.

what is wrong with this program?

```c
// value range example
#include <stdio.h>

int main()
{
  unsigned int foo = 66000;
  unsigned short int blah = 66000;
  printf("%s %d\n", "the value of the unsigned int variable
      foo is ", foo);
  printf("%s %d\n", "the value of the unsigned short int
      variable blah is ", blah);
}
```

../example/example_valueranges/valueranges.c

binary representation for other data types

it is not entirely obvious how to represent `float`'s or `char`'s or other data types using a binary representation. because of this, certain standards have been adopted for representing these types.

for example/example, a `float` is represented by 4 bytes (32 bits). for the **binary32** format from the IEEE 754 standard, one of those is for the *sign*, 8 are for the *exponent*, and 23 are for the *mantissa*.

let the bits for a `float` be denoted $b_{31}, \ldots, b_0$. the representation is:

$$(-1)^{b_{31}} \times 2^{(b_{30} \ldots b_{23})_2 - 127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right).$$

irrational numbers must first be rounded according to rules specified by the IEEE 754 standard.

https://en.wikipedia.org/wiki/Single-precision_floating-point_format

variable definition and declaration

A variable must undergo the following three steps before it can be used by the program

1. **definition** of the variable: allocate memory for storage.
2. **declaration** of the variable.
3. **initialization**: set the value of variable.

`extern` can be used to declare a variable which can then be defined somewhere else.

`static` variables are either global or local. global static variables have scope within the entire file they are defined. local static variables have scope in their function of definition.

`const` can be used to fix the value of the variable.

example using `extern`

```c
# include <stdio.h>
// declaration of float p
extern float p ;
int main ( void ){
  // definition , declaration and initialization of int y
  int y = 1;
  printf ("y = %d \n", y );
  printf ("p = %f \n", p );
}
```

../example/example_variables/variables.c

```c
// definition and initialization of a float y
float p = 3.14159;
```

../example/example_variables/foo.c

compile these two source files as

```
gcc -o main variables.c foo.c
```

example using static

```
# include <stdio.h>
// declaration and initialization of a global static
    variable
int foo = 10;

int my_function()
{
  // declaration of a local static variable
  static int blah = 23;
  printf("print foo in my function = %d \n", foo );
  printf("print blah in my function = %d \n", blah );
}

int main ( void )
{
  printf("print foo in main = %d \n", foo );
  printf("print blah in main = %d \n", blah );
  my_function();
}
```

../example/example_static/static.c

difference between `static` and `const`

```c
# include <stdio.h>
// declaration and initialization of a global static
    variable
static int foo = 10;

int my_function()
{
  // setting value of static variable
  foo = 15;
  printf("print foo in my function = %d \n", foo );
}

int main ( void )
{
  printf("print foo in main before my_function() call = %d
      \n", foo );
  my_function();
  printf("print foo in main after my_function() call = %d
      \n", foo );
  const double blah = 14.258;
  // setting value of const variable... cannot do this!!
  blah = 20.0;
}
```

../example/example_static/const_vs_static.c

function declaration

functions in c have the follwing syntax:

```
return_value function_name(input parameters)
{
// some implementation
}
```

generally, functions **must be implemented above** the function which call them in the source file. this requirement can be hard to deal with if you have a ton of functions that call each other.

incorrect function declaration

```
int main()
{
  int foo = 1;
  int blah = plus_one(foo);
}

int plus_one(int foo)
{
  return foo + 1;
}
```

../example/example_functions/incorrect.c

correct function declaration

```c
int plus_one(int foo)
{
  return foo + 1;
}

int main()
{
  int foo = 1;
  int blah = plus_one(foo);
}
```

../example/example_functions/correct_v1.c

correct function declaration

```c
// function prototype
int plus_one(int foo);

int main()
{
  int foo = 1;
  int blah = plus_one(foo);
}

// function implementation
int plus_one(int foo)
{
  return foo + 1;
}
```

../example/example_functions/correct_v2.c

correct function declaration

```c
// include header file that has function prototypes
#include "correct_v3.h"

int main()
{
  int foo = 1;
  int blah = plus_one(foo);
}

// function implementation
int plus_one(int foo)
{
  return foo + 1;
}
```

../example/example_functions/correct_v3.c

```c
// function prototype
int plus_one(int foo);
```

../example/example_functions/correct_v3.h

command line arguments

it is often useful to pass arguments into the `main` function on the command line. this can be done by including an `int` parameter `argc` and `char**` parameter `argv` in the `main` function:

```
int main(int argc, char* argv[])
{
// stuff here
}
```

the integer `argc` is equal to the number of command line inputs, including the executable `./main`, and the pointer to a collection of character pointers `argv` contains all of those inputs.

`include` **guards**

also called “macro guard,” “file guard,” “guard rails.”

this thing is a piece of code that can be added in the header file to make sure code is **not** included multiple times.

```
#ifndef EX_HEADER_COMP_H
#define EX_HEADER_COMP_H

int add_int(int,int);

extern double MY_PI;

#endif
```

../example/example_include_guard/ex_header_comp.h

important: names for include guards should be consistent and unique.

https://en.wikipedia.org/wiki/Include_guard

pointers

a **pointer** is a variable that contains the address of a variable, as defined by Kernighan and Ritchie.

why: with pointers, we can deal with variable values directly, instead of copies of them.

since all we need to represent a pointer is an address in memory, they are more efficient to deal with, like for example, as parameters passed into functions.

pointers directly “point” to the variable data in memory.

declaration of pointers

pointers are declared by putting an asterisk before the variable name:

```
int* foo_ptr;
```

or

```
int *foo_ptr;
```

are both fine declarations. a pointer can be initialized to the “zero” pointer as follows:

```
int* foo_ptr = NULL;
```

memory addresses and operators

the address of variable in memory is the first byte of memory in which the variable is stored. this means it is important for pointers to be declared with the appropriate type so that "it knows" how many bytes down-the-line correspond to the given variable.

the two important operators for pointers are:

- `&` ← the "address-of" operator.
- `*` ← the "value-of" operator. when used, it is called "dereferencing."

so if we declare a `double` variable as

```
double blah
```

then `&blah` would be the corresponding pointer. if we have a pointer to a variable `blah_ptr`, the value can be accessed as `*blah_ptr`.

an example

explain what is happening below:

```
int X = 2, Y = 3;
int* ptr_int;
ptr_int = &X;
Y = *ptr_int;
*ptr_int = 4;
```

an example

```
// declare and initialize X and Y
int X = 2, Y = 3;

// declare a pointer to an integer
int* ptr_int ;

// assign to the pointer the address of X
ptr_int = &X ;

// assign the value of Y to be the value of X
Y = *ptr_int ;

// change the value of X to be 4
*ptr_int = 4;
```

another example

what is wrong with the code below?

```
double Z = 5.0;
int foo = 14;
int* ptr_int;
double* ptr_double;
ptr_int = &foo;
ptr_double = Z;
```

another example

```
double Z = 5.0;
int foo = 14;
int* ptr_int;
double* ptr_double;

// WRONG!: this is trying to set an integer point
// to the address of a double variable.
ptr_int = &Z;

// WRONG!: this is trying to set a double pointer
// to the value of a double variable.
ptr_double = Z;
```

pointers and functions

```c
// second example with pointers
#include <stdio.h>
void add_one (int* , double*);
int main (void)
{
  int a = 2;
  double b = 3.0;
  printf("%s %d\n", "value of a before function call is ",
      a);
  printf("%s %f\n", "value of b before function call is ",
      b);
  add_one(&a, &b);
  printf("%s %d\n", "value of a after function call is ",
      a);
  printf("%s %f\n", "value of b after function call is ",
      b);
}
void add_one (int* x, double* y)
{
  *x = *x + 1;
  *y = *y + 1;
}
```

../example/example_pointers/ex2.c

pointer arithmetic

```
int ii = 2; // declare and initialize an integer

int* ptr_ii = NULL ; // declare and initialize a
                     // pointer to an integer

ptr_ii = &ii ; // pointer to the integer ii

*ptr_ii = 4; // change the value of ii

*(ptr_ii + 1) = 5; // set value in the ``next''
                   // memory address to 5
```

above, `ptr_ii + 1` corresponds to the **next address in memory**. since `ptr_ii` is a pointer to an integer, this will be the address that is 4 bytes after the memory address stored in `ptr_ii`.

caveat: we do not know if the memory address `ptr_ii + 1` is being used by the program, so we may get a `SEGFAULT`.

pointer dereferencing again

```
int ii = 2; // declare and initialize an integer

int* ptr_ii = NULL; // declare and initialize a pointer

ptr_ii = &ii; // store the address of ii in the pointer

*ptr_ii = 4; // change the value of ii

*(ptr_ii + 0) = 4; // equivalent to *ptr_ii = 4

ptr_ii[0] = 4; // equivalent to *ptr_ii = 4

ptr_ii[1] = 13; // equivalent to *(ptr_ii + 1) = 13
```

note that the last two lines above look like accessing array elements 0 and 1, where we can think about the pointer as an **array of integers**.

arrays in c

as seen on the previous slide, there is a connection between arrays of things and pointers, but we can also have arrays of pointers!

we can have arrays of `int`'s, `double`'s, `char`'s, etc...

the general form of an array declaration is:

```
variable_type array_name[array_size];
```

here are some examples:

```
int foo[10]; // an array of ten integers

int (*boo)[10]; // an pointer to an integer array

int *blah[10]; // an array of integer pointers
```

pointer to an array versus an array of pointers

a pointer to an array can be declared as:

```
variable_type (*array_name)[array_size];
```

an array of pointers can be declared as:

```
variable_type *array_name[array_size];
```

array initialization

an array can be initialized during declaration:

```
float my_array[4] = {3.2, 5.0, 6.375, 1.1};
```

or, you can initialize all elements to zero with:

```
int another_array[50] = {0};
```

if you want to assign a value to a part of an array:

```
my_array[3] = 1.1;
```

note: array indexing in c starts with 0!!!!!

arrays and pointers

arrays and pointers are really the same thing, i.e. the address in memory of the first element of an array is stored in the name of the array. if we declare and initialize an array of integers:

```
int an_array[3] = {1, 5, 6};
```

then

```
an_array[1]
```

and

```
*(an_array + 1)
```

are equivalent statements. what is their result?

multidimensional arrays

previously we were looking at arrays with only a single dimension. it is possible to have multidimensional arrays also:

```
double blah[10][4];
```

would be a 10 by 4 array of doubles. array initialization can be done during declaration also:

```
int foo[2][3] = { {4, 1, 12},    // first row
                  {-10, 2, 5} }; // second row
```

remark on memory allocation

```
int foo[2][2] =  { {1, 5},
                   {-15, 3} };
```

array declaration and initialization as above allocates memory on what is called the **stack**. this is a small chunk of memory used for temporary variables declared within functions.

for much larger arrays and data that will take up more memory, you will want to use the **heap** . memory on the heap must be explicitly allocated and deallocated. allocation will be done with functions like `malloc()` and `calloc()` and deallocation will done using `free()`.

arithmetic operators in c

we have already started using operators, but let's quickly discuss them:

`+`, `-`, `*`, `/` are the usual addition, subtraction, multiplication, and division.

`%` is the "mod" operator, returning the remainder after division.

other mathematical functions can be used after include the `math.h` header file. for example,

$$a^x$$

is expressed as `pow(a,x)` in c syntax.

assignment operators in c

used to modify the value of a variable.

`++` increments a variable by one, e.g. `ii++`

`--` subtracts one, e.g. `ii--`

`+=` add the value on the right to the variable on the left, e.g `a += b` is the same as `a = a + b`.

`-=` same as above but with subtraction.

`/=` same as above but with division.

`*=` same as above but with multiplication.

logical operators in c

these are used in conditional statements.

`&&` is the AND operator.

`||` is the OR operator.

`!` is the NOT operator.

what does this evaluate to?

```
( (5 < 2.3) || (3 == 4) ) && ( !(5 >= 5) )
```

control loops and statements

these consist of

- `for` loops
- `while` loops
- `if-else` statements

let us briefly go over the standard c syntax for these statements.

`if-else` **statements**

```
if ( condition1 )
{
// instructions if condition1 is true.
}
else if ( condition2 )
{
// instructions if condition1 false and condition2 true.
}
else
{
// instructions if condition1 and condition2 are false.
}
```

note that you can have as many or as few "`else if`" parts as you want. in fact, the "`else if`" and "`else`" parts of the statement above are not necessary, if you only want to do something when `condition1 == true`.

preprocessor `if-else` statements

we have already seen this implemented in the guard rails.

here are some examples:

check if a macro is defined:

```
#ifdef DEBUG
printf ("I am in DEBUG mode");
#endif
```

check if a macro is not defined (like for guard rails)

```
#ifndef HEADER_FILE_NAME_H
#define HEADER_FILE_NAME_H
// implementation of header file
#endif
```

preprocessor `if-else` statements

a normal `if-else` statement with macros

```
#if MACRO == 1
// instructions
#elif MACRO == 2
// instructions
#else
// instructions
#endif
```

`for` loops

the general form for a `for` loop in c is:

```
for ( initalization; condition; increment )
{
// loop instructions to do when the condition is true
}
```

here is an example that will print out the integers from 0 to 9:

```
for (int ii = 0; ii < 10; ii++)
{
printf ("ii = %d \n", ii);
}
```

`while` and `do-while` loops

the general form for a `while` loop is:

```
while ( condition )
{
// loop instructions to do when condition == true
}
```

the general form for a `do-while` loop is:

```
do
{
// loop instructions
} while ( condition );
```

note that in a `do-while` loop, the conditions within the curly braces are actually executed *at least once*.

break and continue commands

- break: terminate loop and continue with instructions that follow it.
- continue: force the program to go to the next iteration of the loop.

what does the following code do?

```
for (int ii = 0; ii <5; ii++)
{
        if (ii == 3)
        {
                continue;
        }
        printf ("ii = %d \n", ii);
}
```

a detour: newton's method

suppose we are given a function $f : \mathbb{R} \to \mathbb{R}$ and we want to find x^* so that $f(x^*) = 0$. a common approach is to set up an iteration:

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k).$$

fact: if our "initial guess" x_0 is close enough to x^*, then:

$$x_k \to x^* \text{ as } k \to \infty.$$

where does the iteration come from? try a taylor expansion:

$$f(x_{k+1}) - f(x_k) = f'(x_k)(x_{k+1} - x_k) + O(|x_{k+1} - x_k|^2)$$

for our next assignment, you will implement newton's method and apply it to some functions by using pointers and conditional statements.

an example using newton's method

consider the function $f(x) = ax$, so $f'(x) = a$.

if we write out the newton iteration, it looks like:

$$x_{k+1} = x_k - \frac{ax_k}{a} = 0,$$

so in this case the iteration converges after one step.

consider the function $f(x) = bx^2$. then the newton iteration is:

$$x_{k+1} = x_k - \frac{bx_k^2}{2bx_k} = \frac{x_k}{2},$$

so the iteration converges linearly... why not faster?

memory management in c

storage in memory is either *static* or *dynamic*

variables and functions stored in **static memory** will “last” the for duration of the program. variables which are stored outside the scope of functions are stored automatically in static memory. the keywords `static` or `extern` will also indicate the variable or function should be stored in static memory.

the memory reserved statically **cannot** be modified later.

```
#include <stdio.h>

// here is a variable stored in static memory
static int foo = 10

int main()
{
        // some implementation
}
```

memory management in c

variables and functions stored in **dynamic memory** may “last” for an amount of time which is shorter than the duration of the program. these might be local variables used within functions or variables that have their memory explicitly reserved by the programmer.

local variables declared without `static` or `extern` are automatically stored on the *stack*, a small part of the RAM. memory used for these variables is freed automatically when the program is done using the function they are declared within.

```
void my_function(double foo)
{
        // here is a variable declared on the stack
        float blah = 15.032;

        // some more code

} // memory used to store ‘‘blah’’ is freed here
  // at the end of the function
```

more on the stack

memory is managed on the stack in a "last in, first out" order (LIFO).

the metaphor that is often used here is *stacking* plates. if we have three plates called A, B, and C, and we insert, or *push* them onto the stack in that order, plate C will be on top, followed by B, and then A.

we can *pop* the plates off the stack in reverse order as C, B, A, i.e. C was the last in but the first out.

this is a simple approach to deal with memory, and it makes the stack very fast and efficient.

by the way, **push** and **pop** are parts of names for functions used in c and c++, related to this LIFO concept.

https://www.i-programmer.info/babbages-bag/263-stack.html?start=3

https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

what is your stack size?

```
$ ulimit -a
core file size              (blocks, -c) 0
data seg size               (kbytes, -d) unlimited
scheduling priority                 (-e) 0
file size                   (blocks, -f) unlimited
pending signals                     (-i) 62239
max locked memory           (kbytes, -l) 64
max memory size             (kbytes, -m) unlimited
open files                          (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority                  (-r) 0
stack size                  (kbytes, -s) 8192
cpu time                   (seconds, -t) unlimited
max user processes                  (-u) 62239
virtual memory              (kbytes, -v) unlimited
file locks                          (-x) unlimited
```

dynamic memory on the heap

memory on the heap must be explicitly managed by the programmer. this means doing *allocation* and *deallocation* of the memory being used. the heap size is only constrained by the RAM size.

you will be using the heap probably because you need more storage that what is available on the stack.

heap memory management in c is done primarily with

`malloc`, `calloc`, `realloc`, and `free`

usage for `malloc`:

```
pointer_type* pointer_name;
pointer_name = (pointer_type*) malloc(num_bytes);
```

`num_bytes` is really the product of the number of elements you want to store and the number of bytes per element.

dynamic memory on the heap

while memory initialization is undetermined with `malloc`, the function `calloc` reserves memory as well as initializes all the bits to zero:

```
pointer_type* pointer_name;
pointer_name = (pointer_type*) calloc(num_elements,
                                     bytes_per_element);
```

`num_elements` is the number of elements to be reserved, and `bytes_per_element` is the number of bytes needed for a single element.

note that you can get the number of bytes for a single element with:

```
sizeof(pointer_type)
```

dynamic memory on the heap

```
// make pointer
pointer_type* pointer_name;

// allocate memory of certain size
pointer_name = (pointer_type*) malloc(num_bytes);

// rellocate memory of a different size
pointer_name = (pointer_type*) realloc(pointer_name,
                                       new_num_bytes);
```

dynamic memory on the heap

the `free` function will deallocate memory.

```
// make pointer
pointer_type* pointer_name;

// allocate memory of certain size
pointer_name = (pointer_type*) malloc(num_bytes);

// deallocate memory
free(pointer_name);
```

multidimensional arrays in c

we can declare a multidimensional array as follows:

```
double a_matrix[5][12];
```

where we can think about this as a 5×12 matrix. on what part of the memory is this array stored?

we can imagine a matrix as an array of one-dimensional arrays, which can be read off as the matrix rows or the matrix columns. a question is how the entries *are actually stored* in memory. c uses **row major indexing** of multidimensional arrrays.

fortran uses **column major indexing**.

row major indexing

consider a 2×3 matrix $\begin{bmatrix} 1 & 4 & 2 \\ 5 & 3 & 1 \end{bmatrix}$.

row major indexing of this matrix would be:

$$1, 4, 2, 5, 3, 1$$

where as **column major** indexing would be:

$$1, 5, 4, 3, 2, 1$$

since arrays are stored in contiguous blocks of memory, in c, the values for the matrix above are stored in the row major ordering.

multidimensional arrays: example 1

say we want to allocate memory for an `int` matrix with number of rows “`rows`” and number of columns “`columns`.” there are several ways to do this. below would be using a *single* pointer:

```
int* ptr = (int*) malloc(rows * columns * sizeof(int));
```

to access an element of the array, you need to navigate to the correct place in memory given the fact that c uses row major indexing:

```
for (int ii = 0; ii < rows; ii++)
{
   for (int jj = 0; jj < columns; jj++)
   {
      // Set the value of the element (ii,jj)
      *(ptr + ii * columns + jj) = 3;
   }
}
```

to deallocate memory you can simply call `free(ptr)`

multidimensional arrays: example 2

how about using a *pointer to a pointer* ?

```
// allocate memory for the rows
int** ptr = (int**) malloc(rows * sizeof(int*));

for (int ii = 0; ii < rows ; ii++)
{
   // allocate memory for the columns... for each row!
   ptr[ii] = (int*) malloc(columns * sizeof(int));
}
```

accessing elements can be done as follows:

```
for (int ii = 0; ii < rows; ii++)
{
   for (int jj = 0; jj < columns; jj++)
   {
      // Set the value of the element (ii,jj)
      ptr[ii][jj] = 3;
   }
}
```

multidimensional arrays: example 2

in this case, deallocating memory needs to be done more carefully:

```
for (int ii = 0; ii < rows; ii++)
{
   free(ptr[ii]); // free chunk of memory corresponding
                  // to the columns for a given row.
}

free(ptr); // free chunk of memory for the pointer
           // to an array of pointers for the rows
```

multidimensional arrays: example 3

use a pointer to a pointer, but also explicitly make sure that you are using continguous memory.

```
// allocate memory for the row pointers
int** ptr = (int**) malloc(rows * sizeof(int*));

// allocate a memory block for the 2D array
ptr[0] = (int*) malloc(rows * columns * sizeof(int));

for (int ii = 1; ii < rows ; ii++)
{
   // initialize the memory address for each of the rows
   ptr[ii] = ptr[0] + columns * ii ;
}
```

to deallocate memory, you would call:

```
free(ptr[0]);
free(ptr);
```

memory issues and problems

- **uninitialized memory**. remember that when you call `malloc`, the values in memory are not initialized to anything!
- **memory overwrite**. writing to memory that is outside the bounds of what you have allocated:

  ```
  int* ptr;
  ptr = (int*) malloc(2*sizeof(int));
  ptr[3] = 4; // an overwrite
  ```
- **memory overread**. accessing memory which has not been allocated.

  ```
  int* ptr;
  ptr = (int*) malloc(2*sizeof(int));
  int number = ptr[3]; // an overread
  ```
- **memory leak**. this can happen in many ways, for example/example, like not freeing allocated memory or redefining an address of a pointer before it is freed. tools like **valgrind** and **gdb** can help diagnose these issues.

valgrind and gdb

to install valgrind, type in the terminal:

```
sudo apt-get install valgrind
```

and to install gdb, type in the terminal:

```
sudo apt-get install gdb
```

there are many features within these two pieces of software, but generically, **valgrind** is used for detecting memory leaks and **gdb** can be used for systematically peering into your code.

valgrind and gdb

given an executable `main`, valgrind can be run in the terminal with the following command:

```
valgrind --tool=memcheck --leak-check=full --log-file=out ./main
```

see `valgrind.org` for much more information.

to run gdb on an executable, you would first type in the terminal

```
gdb ./main
```

at which point a gdb prompt appears, in which you can type “run”:

```
(gdb) run
```

see `https://www.gnu.org/software/gdb/` for more info.

more on gdb

if you plan on testing your code with gdb, you should compile it with debugging symbols using the “-g” flag as follows:

```
gcc -g my_code.c -o main
```

note, if you have command line arguments like:

```
./main input_file
```

then you would run this through gdb like:

```
gdb ./main
```

and then

```
(gdb) run input_file
```

structures in c

a struct is defined with the following code:

```
struct personal_info {
   // elements of the structure
   int age;
   char* first_name;
   char* last_name;
};
```

note that this is only the definition of a struct, which is typically put in its own header file. you can think about a struct as a “new” data type in c that can help you organize your code in a logical way.

if you want to declare a variable with type `name_struct`, this would be done as:

```
struct personal_info me;
```

structures in c

“elements” of the structure can be accessed with the *dot operator*. for example, once you have declared a variable of type `name_struct`, you can initialize the elements as:

```
me.age = 31;
me.first_name = "John";
me.last_name = "Smith";
```

you cannot initialize elements in the struct definition, but you can initialize elements of the structure during declaration as follows:

```
struct personal_info me = {31, "John", "Smith"};
```

note that the order in the initializer list must be the same as the order of elements in the struct definition.

structures in c

if you want to initialize structure without respecting the order of the element, you can use the dot operator inside the initializer list:

```
struct personal_info me
 = {.first_name = "John", .last_name = "Smith", .age = 31};
```

instead of dealing with a variable type “`struct personal_info`,” it often makes sense to just call the variable type “`personal_info`.” this can be done by using the `typedef` keyword as follows:

```
typedef struct {
   // elements of the structure
   int age;
   char* first_name;
   char* last_name;
} personal_info;
```

note that the `typedef` keyword generically lets you rename data types.

structures in c

so with the following definition

```
typedef struct {
   // elements of the structure
   int age;
   char* first_name;
   char* last_name;
} personal_info;
```

we can declare a variable of type `personal_info` as:

```
personal_info me;
```

guard rails and structs

typically you will want access to a given struct in *multiple* source files. you can do this by putting the struct in a header file and then including it in a source file with the `#include` directive.

we only want the struct defined once, though, so use guardrails when you define it:

```
#ifndef MY_STRUCT_H
#define MY_STRUCT_H

// struct definition
// function prototypes

#endif
```

the macro `MY_STRUCT_H` should only be defined here and nowhere else. standard practice to have the macro be the actual name of the header file, with “`_H`” replacing “.h”

structs, pointers, and functions

structs can be treated like any other data type. they can be **return values for functions**, you can make **pointers to them**, and you can **pass them into functions** (or preferably, pass pointers to them!)

suppose we declare and initialize a struct and declare a pointer:

```
// declare and initialize struct
personal_info  me = {31, "John", "Smith"};

// declare pointer to struct
personal_info* ptr_me;
```

Then we can set the pointer to the struct using the address-of operator:

```
ptr_me = &me;
```

extracting members from struct pointer

note that once we have a pointer to a struct, we can simulataneously dereference it and get a member of it using the “->” operator as follows:

```
// print out age
printf(‘%d’, ptr_me->age);

// set a new first name
ptr_me->first_name = "chuck";

// equivalent to above
(*ptr_me).first_name = "chuck";
```

in class example: a struct for vectors

what exactly do we need? some bare minimum things are:

- functions for allocating and deallocating memory. these are usually called "constructors" and "destructors" respectively.
- members that specify the vector length, if it has been properly allocated, and a pointer to the vector.
- functions for either reading from or writing to an entry of the vector.

vector operations:

- addition of two vectors.
- multiplication of a vector by a scalar.
- dot product of two vectors.

generating executables

we have been converting our source files to executables, which we usually call `main`, with a single command like:

```
gcc -o main source.c -lm
```

recall the “`-lm`” part links in the standard c math library.

this compilation process can actually be broken into two steps:

1. generate an object file from each source file.
2. from the object files, generate an executable.

generating object files

given a source file `source.c`, an object file can be generated using the compiler and the "`-c`" option as follows:

```
gcc -c source.c
```

the above command will create an object file with name "`source.o`"

the object file contains "symbols" which correspond to the functions used in the file. the symbols can be displayed with the terminal command "`nm`."

```
nm source.o
```

each symbol, or function, is either defined in the corresponding source file, denoted "`T`," or is undefined or defined elsewhere, denoted "`U`."

object files continued

why do we care about generating separate object files for each source file?

answer: if we are editing a large number of source files, each of which needs to be compiled, we only need to regenerate object files for the *modified* source files.

once the object files are generated, they can be compiled into an executable with the command:

```
gcc -o main source.o -lm
```

summary

to summarize, the original one-line command to generate an executable from source files:

```
gcc -o main source.c -lm
```

has now been broken into two separate commands.

```
gcc -c source.c
```

which generates `source.o`. this is followed by

```
gcc -o main source.o -lm
```

which links the object file in this case with the standard math library and then generates the executable `main`.

what is the software `make`??

it allows you to put in rules, name libraries to be linked, and specify compiler options within a single file that can then be used to generate an executable.

in order to use `make`, you need to write a makefile. this file is composed of *macros* and *rules*. macros define environment variables, like which compiler you want to use, etc. rules consists of the following:

- a **target**: this represents an action to do, or a file name that is going to be compiled.
- **prerequisites**: these are the files that are needed (are used by) the target. note that targets defined elsewhere can be used as prerequisites.
- a **command**: possibilities for this include compilation, linking, removing, etc.

Makefile syntax

a rule has the following syntax within a makefile:

```
# here are some comments
target_name: prerequisites
	command
```

note that there must be a tab before the command. also, comments within the file can be added following the # symbol.

a makefile is usually named "`Makefile`" and the commands specified within it are run by typing the command "`make`." when `make` is run, the software will try to run the first target in the makefile. if that target has prerequisites that are also targets, those are run first.

you can also run a particular target by typing:

```
make target_name
```

a first makefile example

```
# set the compiler
CC = gcc

# targets defined below:
# for the executable
main: source.o
	$(CC) source.o -o main

# for the object file
source.o: source.c
	$(CC) -c source.c

# remove executable and object file
clean:
	rm -f main source.o
```

a first makefile example

to compile your code, you can either type

“`make main`” or simply just “`make`.”

if you type `make`, the software will run the prerequisite for the first target to generate the object file, and then it will run the first target to generate the executable.

the command “`make clean`” will delete the object files and executables.

tips and tricks for makefiles

most likely you code will contain many source files, and you do not necessarily want to write a target for each source file. what you can do is put all the source files names into a single macro as:

```
SOURCES = source1.c source2.c
```

given this macro for the source, you can make a macro for the object files like

```
OBJECTS = $(SOURCES:.c=.o)
```

data and file io in c: write data to terminal

recall that we can write to the terminal with `printf`. here is an example:

```
printf("the first three significant
        figures of pi are %f\n", 3.14);
```

note that the within the format specifier you can use the following:

- `%c` refers to one character
- `%s` refers to a string of characters
- `%d` refers to an integer
- `%f` refers to a float
- `%p` refers to the address contained in a pointer

read data when executing the program

this can be done in several ways. we have already seen how to pass arguments into the main function on the command line.

```
int main(int argc, char** argv){
        // stuff!
}
```

where argc is the number of arguments given on the command like, and argv is an array of strings, i.e. a pointer to a character pointer.

read data during program execution

this can be done with the function `scanf`. it has the following syntax:

```
scanf(type_of_variable, path_of_variable);
```

`type_of_variable` is the data type for the variable to be read.

`path_of_variable` is the memory address where we want to store the variable.

`scanf` should usually be preceded with a call to `printf` so the user knows exactly the data type to be entered into the terminal:

```
int ii;
printf("please enter an integer into the terminal: ");
scanf("%d", &ii);
```

in the above code, the program will wait after execution of `printf` for the user to enter an integer and then press "ENTER"

read data during program execution

you can also read in multiple things at the same time:

```
int ii;
float ff;

printf("please enter an integer and
        float into the terminal: ");

scanf("%d" "%f", &ii, &ff);
```

input and output using files

data files are an important way to store, or “dump,” important output from your program during its execution.

as before you need to include the header file `stdio.h`.

you also need to declare a pointer to an object of type `FILE`.

the are functions you can then use with this object, for example to open a file:

```
// declare file pointer
FILE* file_ptr;

// open file
file_ptr = fopen("filename", "mode");
```

input and output using files

the `mode` parameter for `fopen` can have different values, depending on how you want to interact with the file:

- `r`, open to read
  returns `NULL` if file does not exist.
- `w`, open to write
  file is created if it does not exist, if it does, it is overwritten.
- `a`, open to append
  file is created if it does not exist, if it does, data is appended.
- `r+`, open to read and write
  returns `NULL` if file does not exist.
- `w+`, open to read and write
  file is created if it does not exist, if it does, it is overwritten.
- `a+`, open to read and append
  file is created if it does not exist

input and output using files

for binary files, the `mode` parameter must be modified to include "`b`."

for example, if you want to read a binary file, you would have to open it with mode "`rb`."

also, the `filename` parameter can *include the full path* to the file location in case the file is not in the working directory.

when you are done working with a file, you can (and should) close it via:

```
fclose(file_ptr);
```

writing to a file

the functions fputc and fputs allow you to write a single character or string to an opened file, respectively.

```
// write the single character
// corresponding to the integer c
// to the file corresponding to file_ptr
fputc(c, file_ptr);

// write a string to the file
// correspondingn to file_ptr
fputs("a_string", file_ptr);
```

writing to a file

if you want to write something other than a character or a string to a file, use `fprintf`.

```
FILE* file_ptr;

file_ptr = fopen("example/example.txt","w");

int ii = 5;

// write a string
fprintf(file_ptr, "Hello \n");

// write a string that contains an integer
fprintf(file_ptr, "ii = %d \n", ii);

fclose(file_ptr);
```

reading from a file

the function fgetc will read a single character and return the ASCII representation of that character as an integer.

```
int ii = fgetc(file_ptr);
```

the character that is read from the file is the one where the “stream” is currently located. after reading that character, the stream moves to the next one.

the function fgets can read a string:

```
char buffer[BUFSIZ];
fgets(buffer, BUFSIZ, file_ptr);
```

BUFSIZ is a macro defined in stdio.h which is an integer at least as big as 255. it is supposed to be a “guess” for the number of characters in a given line of the file.

reading from a file

the function `fscanf` can read things from a file of other types:

```
double random;

fscanf(file_ptr, "%lf", &random);
```

in class example: a struct for vectors

what exactly do we need? some bare minimum things are:

- functions for allocating and deallocating memory. these are usually called “constructors” and “destructors” respectively.
- members that specify the vector length, if it has been properly allocated, and a pointer to the vector.
- functions for either reading from or writing to an entry of the vector.

vector operations:

- addition of two vectors.
- multiplication of a vector by a scalar.
- dot product of two vectors.