

Introduction to C

Blue Waters Petascale Institute 2018

Pre-Institute Training Material

Authors: Mobeen Ludin and Aaron Weeden, Shodor

KEY: **\$** indicates the start of command. If you want to enter the command, you should not type the **\$** sign. means type a space, **** means type a space but do not hit enter yet, **<ENTER>** means type the enter key, and things in **this color** should be replaced with relevant information. Source code file has an extension of **.c** and executables have a file extension of **.exe**.

Begin by following these steps

Log in to the Shodor testbed server

```
$ ssh username@testbed.shodor.org<ENTER>
```

Download the starter code onto the Shodor testbed server:

```
$ git clone \
https://github.com/aaronweeden/pi2018-intro-to-c.git<ENTER>
```

Change into the examples directory:

```
$ cd pi2018-intro-to-c<ENTER>
$ cd examples<ENTER>
```

Disclaimers

This is a short tutorial on C. It is intended to be a quick introduction to programming and more specifically C syntax. You should definitely take a look at reference material for in-depth tutorials and information on C. Also, only the first program will be explained line by line, every other example will basically have the same structure and some new concepts.

What is C

C is a programming language designed for writing very fast applications, because it allows the programmer to manage multi-level memory as well as I/O devices.

How to compile and run the examples

Each example we will show you has its code in a file with the extension **.c**. Remember you can use the **less**, **vi**, or **cat** commands (or any others you know) to view the code in that file.

Many Linux systems (including testbed.shodor.org) have a built-in GNU C compiler, and the way to compile is below:

```
$ gcc example.c -o example.exe <ENTER>
```

This creates an executable file named **example.exe**. To run the file, we simply provide the path to it, i.e.:

```
$ ./example.exe <ENTER>
```

On Blue Waters, compiling is different. Blue Waters has a compiler wrapper command called **cc** which contains additional options besides the default ones (and uses the Cray C compiler by default instead of the GNU C compiler). However, the command line looks similar:

```
$ cc example.c -o example.exe <ENTER>
```

Running on Blue Waters is more complicated, and we will not discuss it here.

For each of the examples we will go through, we recommend reading the code, compiling it, and running it. For further exploration, we recommend making changes to the code to see how it works.

First C Program [greetings.c]

Oftentimes the best way to learn a programming language is to take a look at a working example, so let's get started with our first simple program that basically prints a message to the screen.

NOTE: the line numbers are provided for reference; they are not part of the code. Some editors will have them, or allow the programmers to turn them on.

```
1/*****
2 * Filename: greetings.c
3 * Author: Mobeen Ludin
4 * Description: first c program. This simple program basically
5 *             prints a greetings message to screen
6 * How to Compile: gcc -g greetings.c -o greetings.exe
7 * How to Run: ./greetings.exe
8 * Output: Greetings from: Hello World
9 *****/
10 #include <stdio.h> //C standard Library that defines printf
    function.
11
12 int main(){
13
14     printf("Greetings from: Hello World\n");
15
16     return 0;
17
18 } // END: main()
```

Line 1-9: This is a block comment in C. It is a good programming practice to have comments in the code that describe the algorithm, functions, variables, and data structures. For multi-line comments in C, you use the `/*` (forward slash and a star) to start the comment section and `*/` (star and forward slash) to close the comment block. Anything `/* INSIDE */` is ignored by the compiler and will have zero effect on your program behavior.

Line 10: Demonstrates the use of the `#include` directive which is an instruction to the compiler. It tells it to import the file whose name is specified between the angle brackets. We use the `#include` directive to include external libraries or header files (`*.h`) in our programs. These external libraries define useful functions and variables which we might use in our program. In this program we include the "standard input/output" [`stdio.h`] header file. This header file contains many of the functions which are used to print to the screen or grab an input from the user; it defines the function `printf()` which we will use in the main program. C also allows the programmer to have inline comments or single-line comments. A single-line comment starts with `//` (two forward slashes); everything after the `//` is ignored by the compiler until the end of the line. Inline comments are useful for a brief description of a data

structure or variable in the code as well as marking the end of a function or loop. They can also be used to temporarily disable lines of code for testing or debugging purposes.

Lines 11, 13, 15, and 17: These are empty lines, ignored by the compiler.

Line 12: This is the '**main** function'; it is the first bit of code that runs when the program is started. In C, a main function must be given the name **main**. The function name is followed by a pair of parentheses. For the main function, the parentheses may contain two named arguments, **argc** and **argv** (we will discuss these in detail later), separated by a comma. Blocks of code, such as the code inside the main function, are enclosed by a pair of curly brackets **{}**. Note that each statement in C that is not followed by brackets must be terminated with a semicolon, **;**.

Line 14: `printf()` is a defined function in the **stdio.h** library. It allows the programmer to print some text to the screen. In this line we are using the **printf()** function to print a message, or a string of characters, to the screen: **Greetings from: Hello World**. The message string has to be enclosed in double quotes. The **\n** is used to start a new line at the end of the message. More on functions later.

Line 16: The **return 0**, indicates that, at the end of the **main()** function, no problems were detected during the run of the program. Note that in the UNIX shell, **0** indicates success/true, which is why we're using it here.

Line 18: `}` is used to close the block of code belonging to the **main()** function. Because it can be easy to confuse which curly braces belong to which block, I have used a single-line comment to mark the end of the block (in this case, the end of the **main()** function).

Variables and Data Types in C | `variables.c` |

In our previous examples, we used **printf()** function to print a simple output message to the screen. However, printing just a greetings message is not doing anything useful. In a typical program we will have some data that must be stored in computer's memory, and we will have to perform some operations on these data. When you need to store values in memory with your program, you need to declare variables. A variable is simply an identifier for a location in memory to which some value can be assigned.

A variable has the following attributes:

- Name (assigned by the programmer, can't start with a number).
- Address (a location in memory, assigned by the runtime system).
- Data type (examples: **int** (integer), **float** (single-precision decimal), **double** (double-precision decimal), **char** (character)).
- Value (can be defined either at compile time or runtime).
- Scope (code blocks can be nested inside each other, with variables only visible to certain blocks depending on where they are declared).

```
11 #include <stdio.h> //C standard Library, defines printf function.
12
13 #define PI 3.14159265358979323846 // Defining constants
14
15 int main(){
16     int radius;           // Variable declaration
17     float area = 0.0;     // Variable initialization
18     double pi = PI;       // C is case sensitive
19     radius = 8;
20     area = pi * radius * radius;
21
22     printf("Area of a circle with radius: %d is: %.5f \n", radius,
area);
23     return 0;
24 } // END: main()
```

Line 13: Constant Variables in C: as the name suggests, the value of constant variables are constant and unchanging. They are defined at the compile time, and their values cannot change during the execution of the program. The **#define** directive is used to create an identifier and associate the text on its right to it. When the code is compiled, the compiler's preprocessor will replace all each instance of the identifier with its associated text. In this example, every time **PI** is used in the program, the preprocessor does a literal text substitution of it with the value **3.14159265358979323846**. Since the preprocessor does a literal search and replace before the code is compiled, if you have a typo in your **#define** statements, you can get some compiler errors that may be hard to diagnose!

Line 16 declares an integer variable named **radius**.

Line 17 declares a variable **area** of type **float** (to store area of the circle as a decimal number) and assigns it the initial value **0.0**.

Line 18 declares a variable **pi** and initializes it to have the value of the constant **PI**. Note that C is case-sensitive, which means that **pi** and **PI** are two distinct identifiers.

Line 19 initializes the variable **radius** to have value of **8**.

Line 20 computes the area of the circle using the formula **A = pi * r^2**. Note that the symbol **^** does NOT mean exponentiation in C (it performs a different operation altogether), so we use **r * r** instead.

Line 22 uses the **printf()** function to print the value of **radius** and the computed **area** of the circle. The **printf()** function is used to print what is called a *formatted string*. When the computer executes this statement and reaches the **%d** and **%.5f** symbols, the **printf()** function looks for other arguments (inputs) to the **printf()** function and replaces **%d** and **%.5f** with those values: in this case, **radius** and **area**, respectively. **%d** means to format the given variable as an integer, while **%.5f** says to format the given variable as a floating-point number with 5 digits after the decimal point.

Conditionals (if and else) [conditionals.c]

In our previous example, we defined some variables and computed the area of a circle. Now let's assume our circle has the following attributes: area, diameter, and circumference. We can do three distinct computations about the same object, as you can see in the image below, which is obtained from Google by searching "area of circle".

Circle

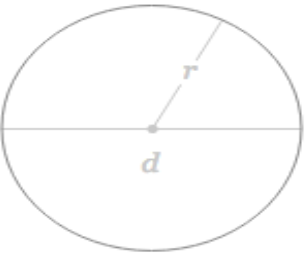
Solve for **area** ▾

$A \approx$

radius
diameter
circumference

r Radius

Solution
 $A = \pi r^2 = \pi \cdot 8^2 \approx 201.06193$



Sometimes we want our program to make decisions based on user input and by evaluating conditional expressions. C provides an **if** statement for evaluating expressions. The conditional expression is placed between the parentheses and must be able to evaluate to **true** (any non-0 integer) or false (0).

In this example the program is little bit more interactive. It makes use of the **printf()** function to print a message to the user about the type of computations it can do, i.e. finding the area, diameter, or circumference of a circle. The program then makes use of the **scanf()** standard C input function to ask the user for the type of computation they would like to do. Finally, the program uses **if** and **else** statements to evaluate the user input. If the user entered **1**, it will execute the statement inside the first **if** statement which is to compute the area of a circle. However, if the user entered **2**, the first **if** statement will evaluate to false (0), so it will escape executing the statement for computing the area and move on to evaluating the next expression for calculating diameter. If that **if** statement is true, it calculates the diameter, otherwise it moves on and calculates the circumference. Below is some C-like pseudo-code for this algorithm.

```
int solve_for;           //Variable to hold user's option.
scanf("%d", &solve_for); //Get user input

if (solve_for == 1)
    A = π * r^2;         //Compute area
else if (solve_for == 2)
    d = 2 * r;           //Compute diameter
else
    c = 2 * π * r;       //Compute circumference
```

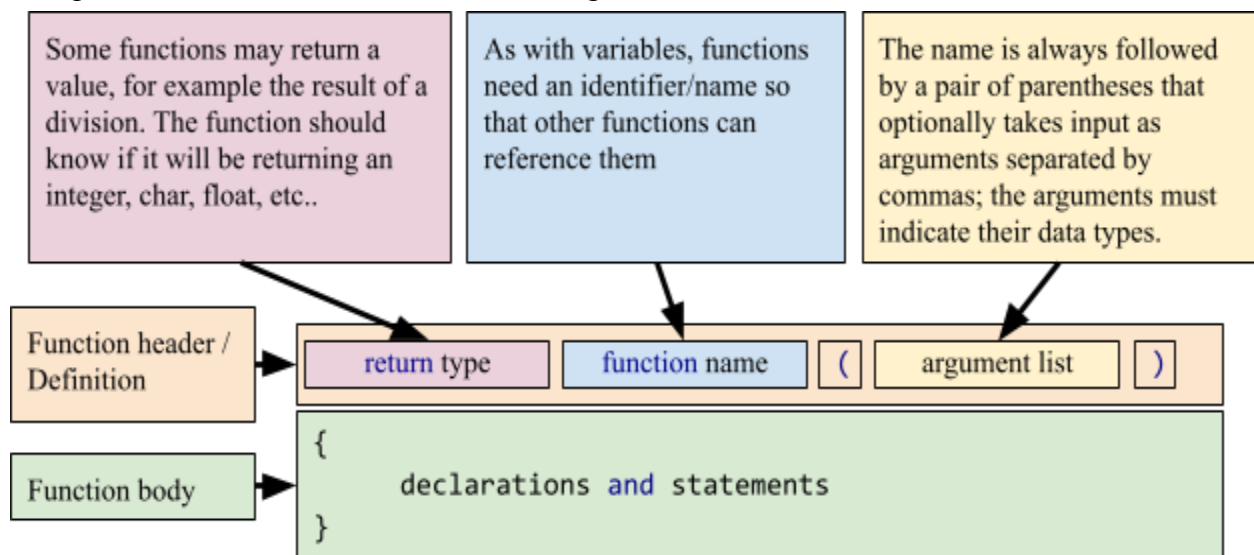
The **if** conditional statement is one of the most powerful concepts in programming. To avoid writing too many **if** and **else** statements for evaluating a huge number of test cases, C provides a **switch** statement to reduce complexity. See [**functions.c**] for an example of a **switch** statement. It does pretty much the same thing, except in fewer lines of code, and the programmer doesn't have to worry about too many open- and close-parentheses and/or brackets.

In an **if/else** statement, you can use the operators in the table below to test for a condition, e.g. to compare the values of two variables. You can also use parentheses to define the order of operations.

Operator	Example	Description
<code>==</code>	<code>x == y</code>	Is x equal to y ?
<code>!=</code>	<code>x != y</code>	Is x NOT equal to y ?
<code><</code>	<code>x < y</code>	Is x less than y ?
<code><=</code>	<code>x <= y</code>	Is x less than or equal to y ?
<code>></code>	<code>x > y</code>	Is x greater than y ?
<code>>=</code>	<code>x >= y</code>	Is x greater than or equal to y ?
<code>&&</code>	<code>a && b</code>	Are statements a AND b both true?
<code> </code>	<code>a b</code>	Is either statement a OR statement b true, or both?
<code>!</code>	<code>!a</code>	Is statement a NOT true? (i.e. is statement a false?)

Functions in C [`functions.c`]

C programs are organized into sets of instructions we call functions. In this example, we will see two main reasons we use functions in our programs. One is to divide a program into subtasks, and the other is to reuse the same code for similar subtasks instead of rewriting that code multiple times. C functions have the following structure:



Example:

```
// Division Function
float division(int num1, int num2){
    float result = 0;
    result = num1 / num2;
    return result;
}
```

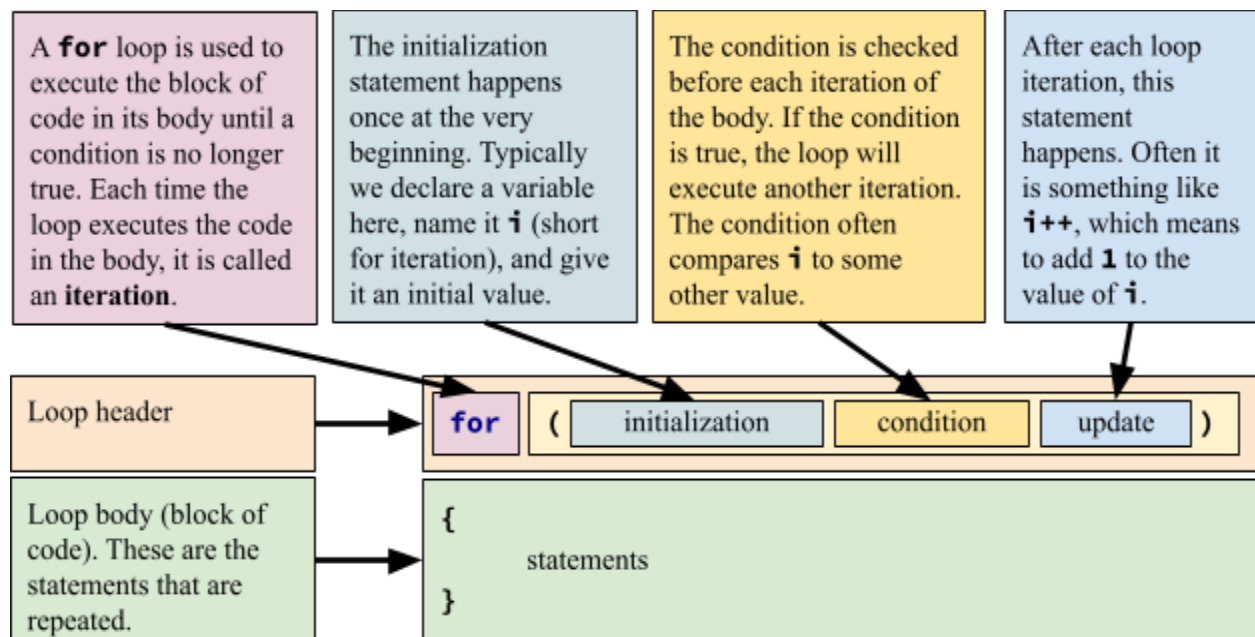
To make a function execute, you must **call** the function by providing its name, a pair of parentheses, and a list of any values to use as inputs. The number and types of inputs provided must match the number and types of arguments in the declaration of the function.

Example: **division(2.5, 3.7);**

Loops and Arrays [for loop.c]

In this example we will use arrays to declare three **float** vectors: **vectA[VEC_SIZE]**, **vectB[VEC_SIZE]**, **vectSum[VEC_SIZE]**; where **VEC_SIZE** = number of elements of each vector. And **for** loops are used to perform operations on arrays.

Computers were invented to perform repeated operations and task. Sometimes we have to perform some operations until a condition is met or repeat it a fixed/known number of times. In almost all modern programming languages including C we use the **for** loop to perform some kind of task some number of times.

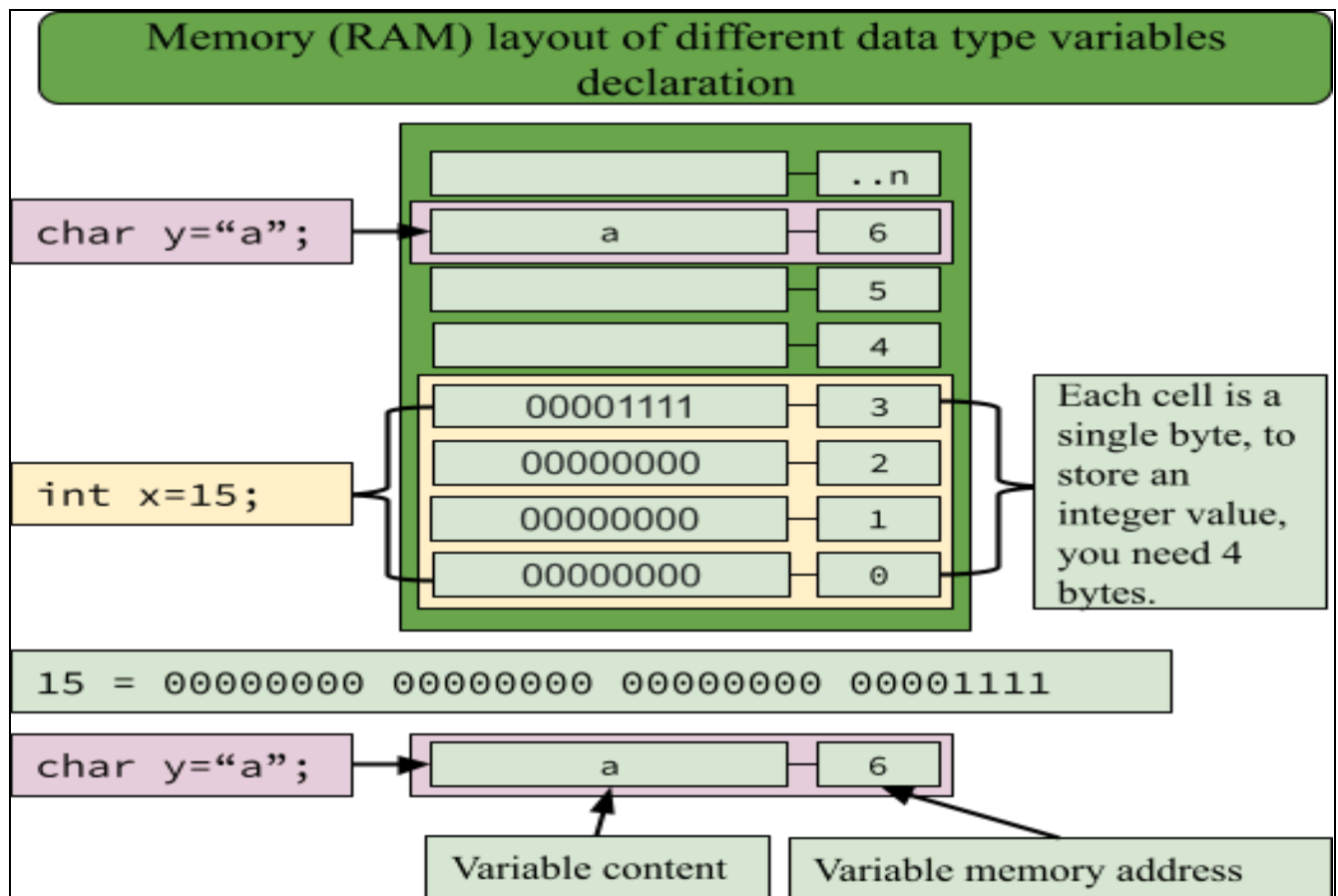


Arrays:

Suppose we give all of you (let's assume there are 35 of you) a quiz question to solve and grade. We want to be able to store all the scores and compute the average. One way to do that is to create 35 variables (s0,...,s34) to hold individual scores, and another variable (average) to hold the average grade. It's a lot of work declaring 35 variables, but still feasible. But what if we wanted all the University of Illinois student to take the quiz? or maybe nationally. Declaring thousands and millions of variables could easily get out of control, not to mention hard to debug that code or read. In C we use the **array** data structures to group all the students under one name and store each score. For example: `float student[35];`

Arrays are used only to group variables of the same data type for example **float**. Also you must declare the size of the array when initialized, because the compiler need to know how to manage memory in way that all the elements are stored sequentially.

Array memory layout: You think of computer memory as gigantic array of **cells/bytes**. Each cell has a name starting from 0 to MAX_MEM_SIZE. The name of the cell is also known as the **address** of cell location in memory.



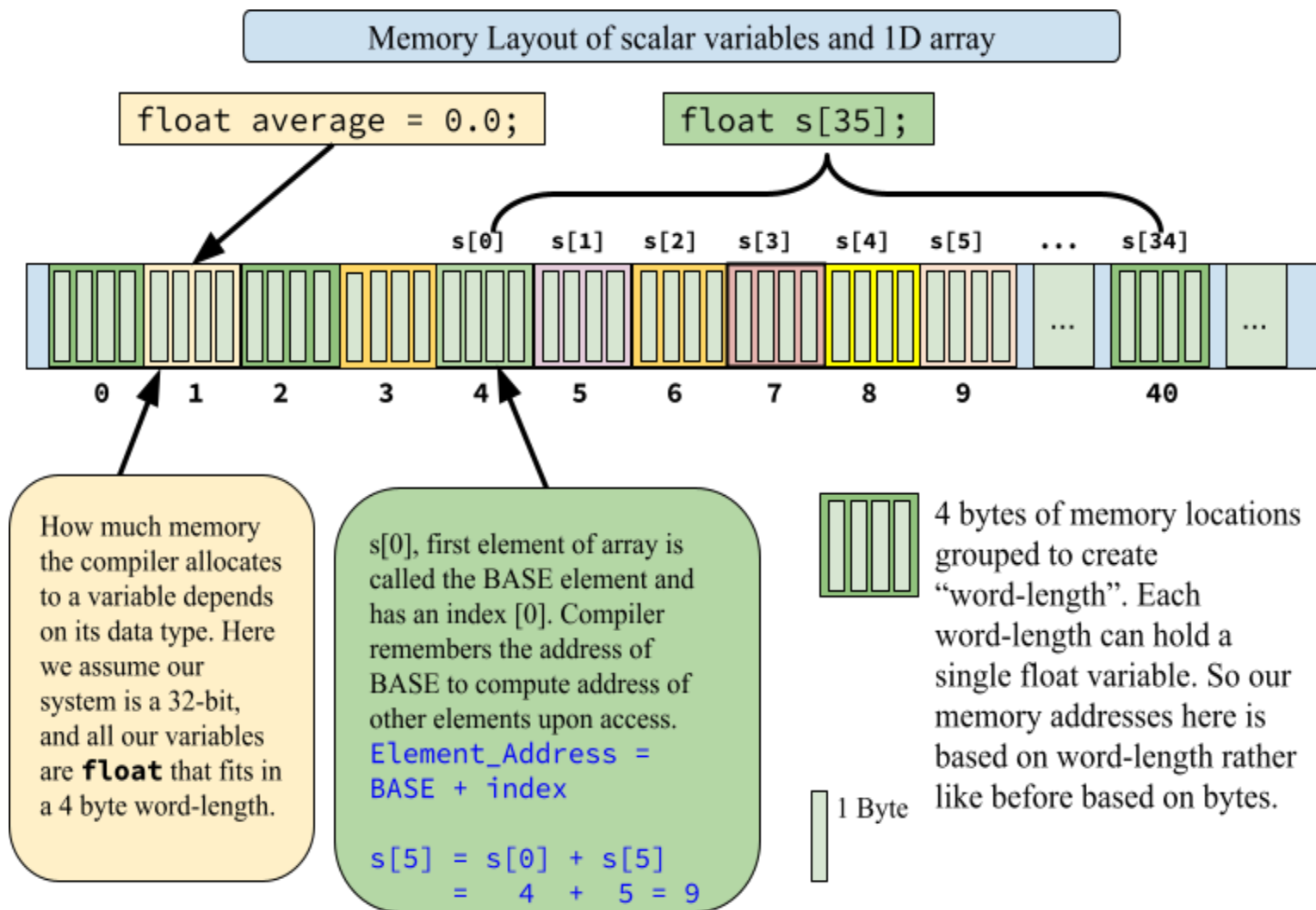
Type	Size in Bytes	Size in Bits
int	4 Bytes	32 bit
float	4 Bytes	32 bit
double	8 Bytes	64 bit
char	1 Byte	8 bits

You can think variables memory locations as a box, that always contain exactly one thing (**integer/float/char**). And the value of the variable is like the content of the box.

Boxes comes in different size. The **char** box is like 1cm x 1cm, **int** is 4cm x 4cm and a **double** is like 8cm x 8cm. The bigger the box the higher it costs to ship or purchase. It's the same concept with variables. A **double** costs a lot more than a **float**. Because we have to allocate 8 bytes instead of four.

Why it's important to know. As a programmer you have to be conservative when declaring variables because it can have a huge impact on the program performance. As you can see in the example above, **x** is declared as an integer with value **15**. Memory is allocated based on the size of the data type. Even if only **address 3** is enough to store value **15**, the compiler would still allocated the preceding three address and fill it with zeros. If we would have declared x as double, we would be wasting 7 addresses instead.

This is important especially when dealing with arrays, because arrays are made of contiguous grouped elements of the same data type. Each individual array element is treated as a variable.



Static 2D Arrays in C [arrays.c]

This example is a matrix multiplication. The main objectives of this example are:

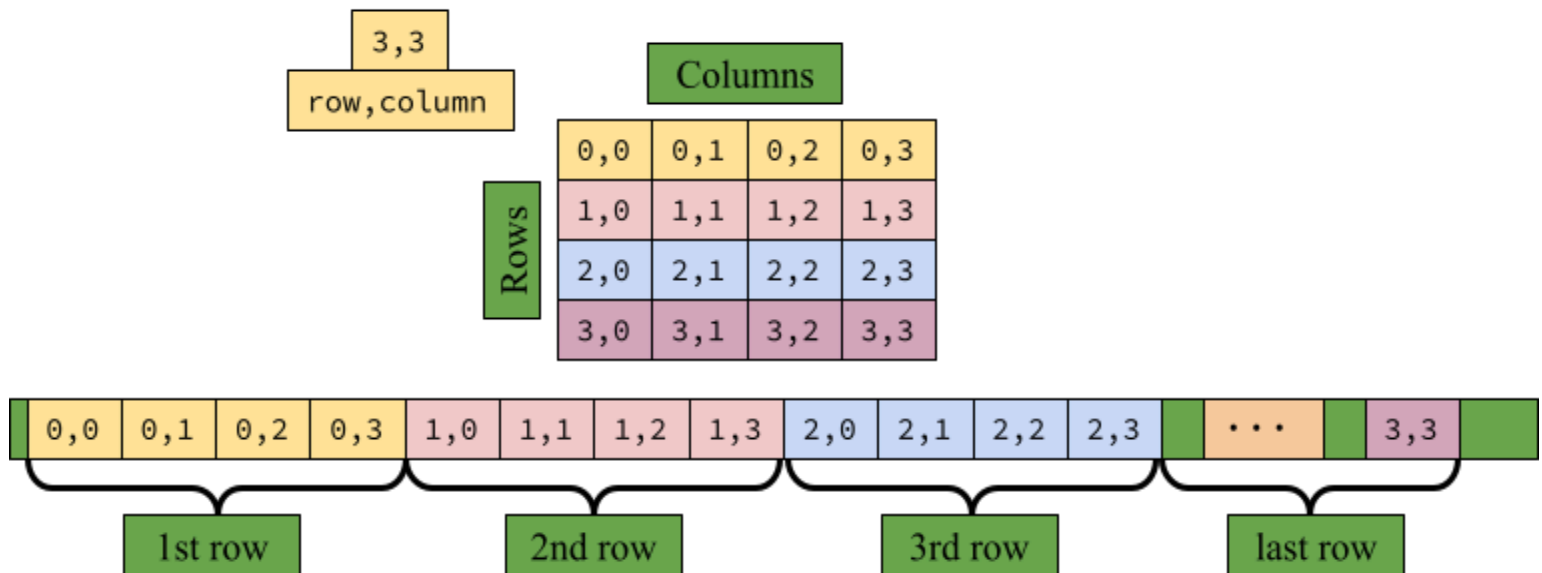
- How declare and initialize 2D arrays of various size.
- How to populate the array with known elements.
- Naïve matrix multiplication algorithm.
- How use for loops to print 2D arrays/matrices.

Multi-Dimensional Arrays Memory Layout:

Depending on language/compiler or systems, multi-dimensional arrays are stored either row-major or column-major. You can think of 2D array as a matrix of certain size.

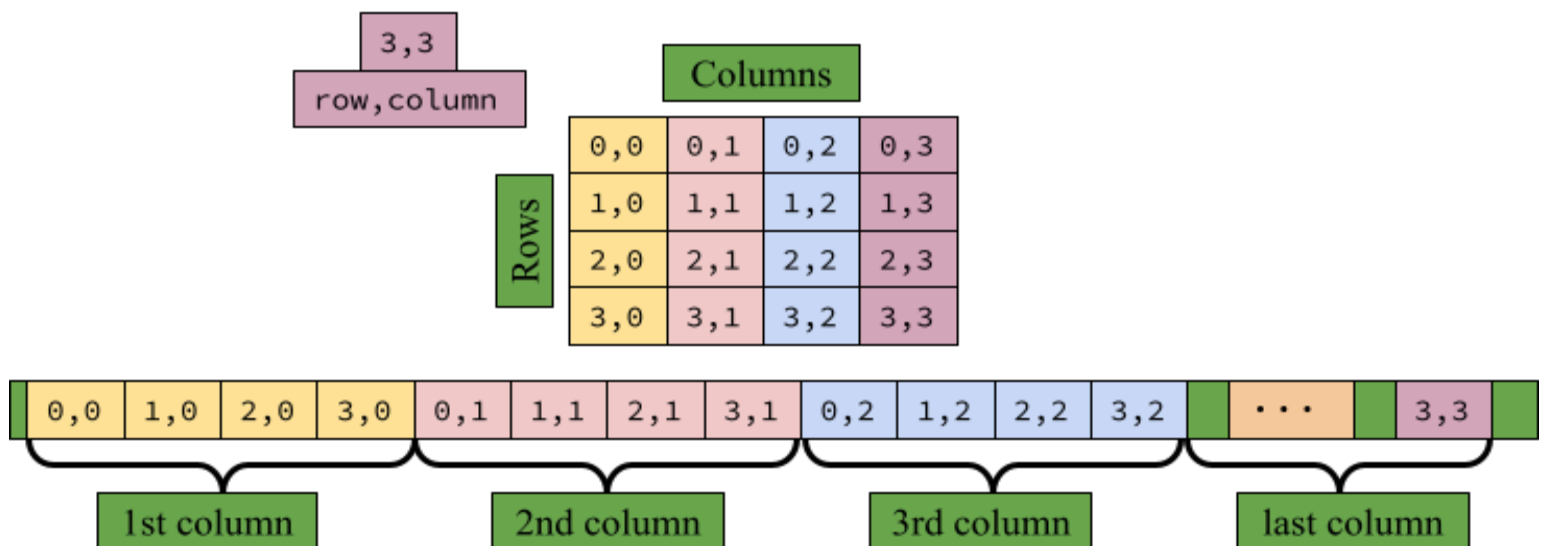
Row-major: the row-major puts the first row in contiguous memory, then the second row right after it, and so on. C/C++ stores array in a row-major order.

Row-major: 2D array memory representation



Column-major: the column-major on the other hand puts the first column in a contiguous memory starting with the base/first (0,0) element until last element of first column (3,0). Then the second column, and so on. Fortran uses column-major order for storing arrays.

Column-major: 2D array memory representation



Pointers and Arrays in C [pointers_and_arrays.c]

This example introduces that concept of pointers. So far we have learned that when we declare variables, the compiler allocates memory for them based on their data type. As you can see in the example in the right. We know that every variable has an address and value associated with it. For example the variable “**y**” has an address “**6**”, and value of “**a**”. Variables that needs more than one byte are usually addressed by the address of their first byte. If we wanted to update the value of a variable, for example, **x = 20**. The computer will go to its lookup table to find variable named **x**, grabs its address and update the value at that address from **15** to **20** (well of course in binary). This is all great and powerful magic that usually is happening behind the seen by the compiler and control units. But what if as a programmer we want to know the address of variables and operate upon these memory address.

Well, with great power comes great

responsibility. Fortunately, C/C++ allows the programmer to directly manipulate memory addresses with the use of pointers.

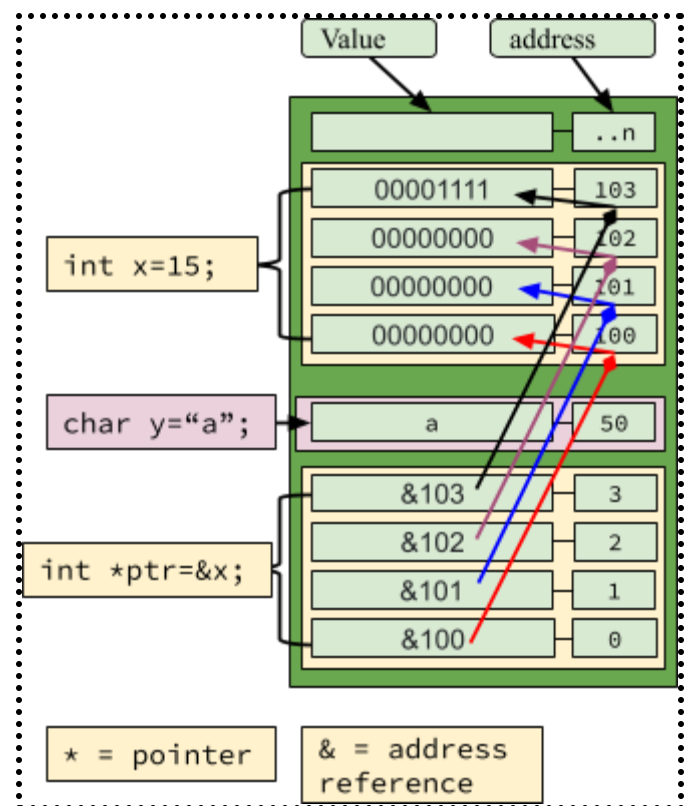
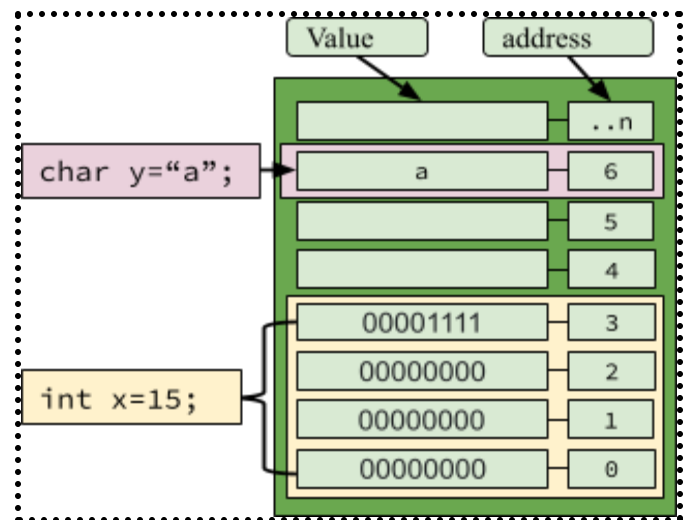
Pointers: are variables that store the addresses of other variables. For example:

```
int x = 15;
char y = "a";
int *ptr = &x;
```

First we declare a variable **x** and initialize it with value **15** and it's stored at the address **100...103**.

We then declare another variable **ptr**, the type of which is a pointer to an integer. It stores the address of variable **x**. When declaring pointers all we have to do is to put an ***** before the variable name. To get the address of the variable, we put **&** before the name of the variable. Like: `int *ptr = &x;`

One thing to keep in mind is that **ptr** still occupies the same amount of memory as variable **x**.



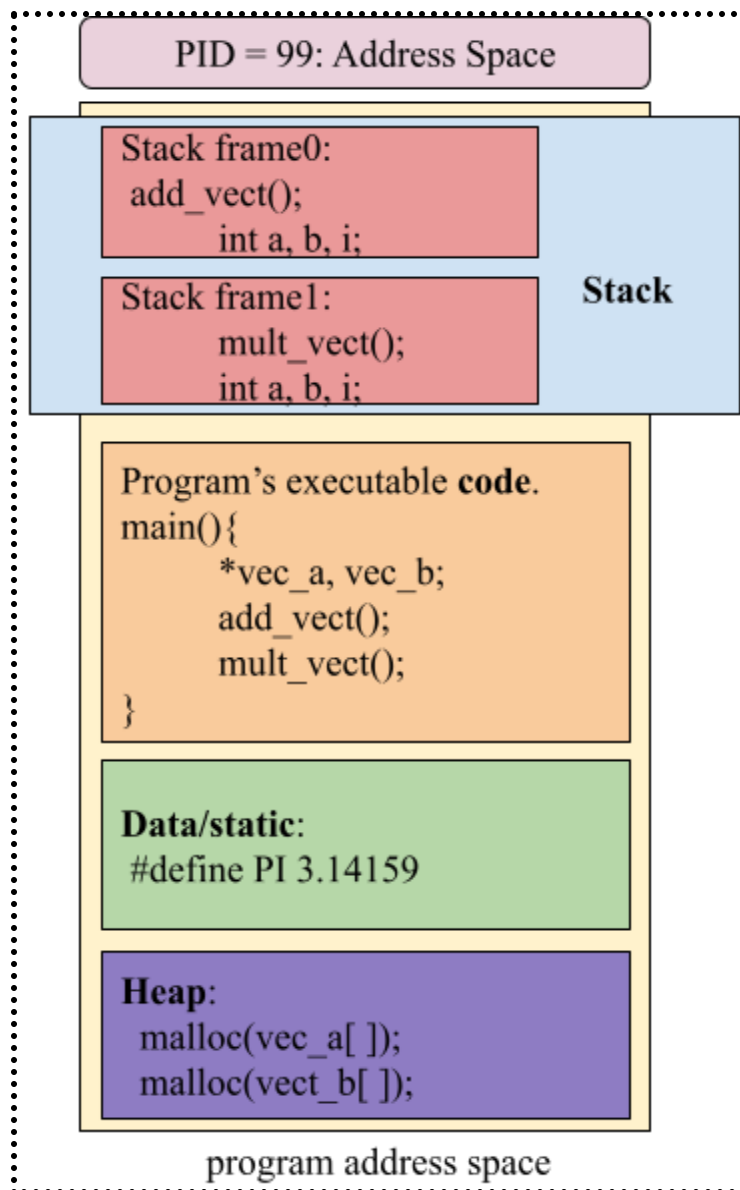
That also means that if we change the addresses in **ptr** from **100...103** to **230...233**, or whatever, it will be pointing now to a different variable in memory.

```
void main(){
    int x = 5;
    int *ptr;
    ptr = &x;
    printf("ptr = %p \n", ptr);
    printf("&x = %p \n", &x);
    printf("&ptr = %p \n", &ptr);
    printf("*ptr = %d \n", *ptr);
    *ptr = 8;
    printf("new x = %d \n", *ptr);
}
```

Output:

```
ptr = 0x7fff4fe63a0c
&x = 0x7fff4fe63a0c
&ptr = 0x7fff4fe63a00
*ptr = 5
new x = 8
```

NOTE: some platforms represent address in hexadecimal for compactness.



Function Arguments as pointers and Dynamic Arrays:

Pointers are very useful when you want to manipulate or perform operations on variables of another function or outside the function being called.

When we run a program, let's say:

./pointers_and_arrays, the operating system will set aside some memory called address space for the execution of this program. The address space is usually divided into four chunks:

1. **Stack:** which is used for functions and short term and temporary data variables local to each function. When the program execution reaches a function call, it will create a stack frame for that function. Each function will have their own stack frame and variables that are not accessed by other functions.

2. **Code:** where instructions and program executable lives.
3. **Data/stack:** where constants, and global variables are stored.
4. **Heap:** The data in heap is for long/life time of the program execution. Mostly dynamically allocated memory accessible by all functions.

In our example, we have three arrays for storing vectors.

```
float *vectA;    //declaring a pointer variable named vectA
float *vectB;    //declaring a pointer variable named vectB
float *vectSum;  //declaring a pointer variable named vectSum
```

Each array is first declared as a pointer to a `double` variable and then dynamically allocated memory to it. As we saw in the array example, each array itself is a pointer. For example, when we declared the array `float student[35];`. Here the size of the array (35) means to declare 35 `float` variables. Since the size is known, this array is declared on the stack and 35 contiguous 4-byte segments (totalling 140-bytes) must be allocated to store all 35 `floats`. The variable `student`, which we refer to as an array, is actually a pointer that references the start of the array, the memory at the location `student[0]`.

Often in a program we want to either put variables on heap to be globally accessible by all the functions in our program, or be able dynamically allocate memory for resizable variables/arrays. To do so in C, we can use the `malloc()` function from the standard library (`stdlib.h`) that takes the number of bytes to be allocated as its argument, and it returns a pointer to a memory region on the heap of the requested byte-size.

For example:

```
vectA = malloc(sizeof(double) * VEC_SIZE);
```

Here is a code snippet to allocate memory to store `vectA` as an array of `double` on heap. First, to allocate an array of `double` on the heap, we have to know how **big** the array is going to be, that is, what size is it? This could be determined either at the runtime or compile time.

In the example we define a macro `VEC_SIZE 10`. Here the `malloc` function takes an argument that is the total number of bytes required to hold 10 `double` elements. Its computed based on:

- `sizeof(double)`: which is itself a C function that finds the size of a data type. For example for a `double` it will return 8-bytes.
- `VEC_SIZE`: the size of array (how many elements).

Since a `double` is 8 bytes in size, `malloc()` will allocate 80-bytes (8*10) of memory on the heap in which the array of `double` can be stored. `malloc()` then returns the memory address of the newly allocated memory, which is assigned to `vectA`. Since `vectA` is located on heap and globally accessible, it is accessible by all the functions in our program. We do not have to redefine the array in each function in our program.

Once we are done using the array, we have to free that allocated memory using the `free(vectA)` function from C standard library (`stdlib.h`). If we don't free the memory it could cause different problems such as memory leak and more; however, those are advanced topics beyond the scope of this lesson.

Exercise: Computing π as an area under a curve.

Task:

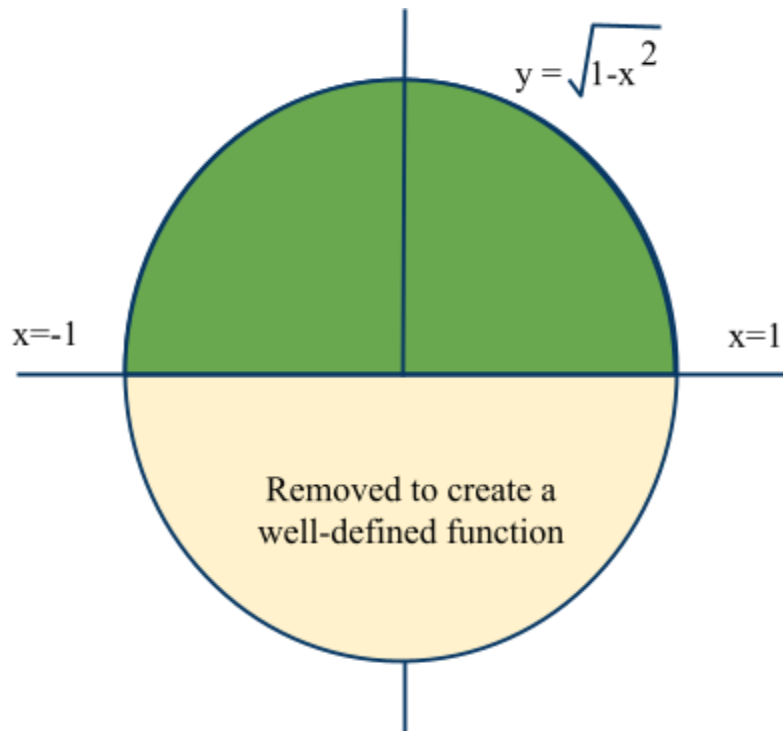
You do not need all the materials you learned about C today to solve this problem. However, with a subset of most common C concept/syntax you should be able to solve this problem.

Problem definition:

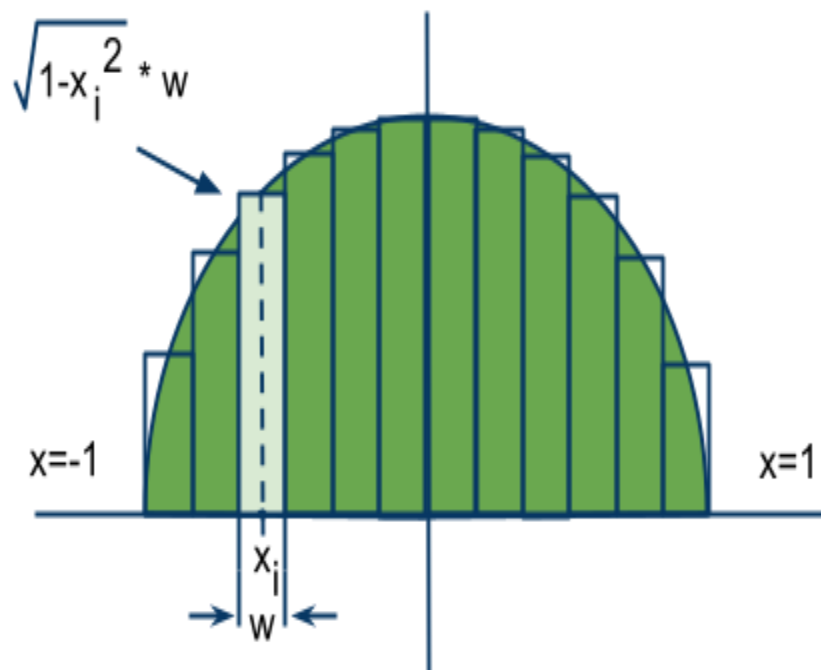
Often, we can estimate a desired quantity by finding the area under a curve (an integral). As an example of this type of computation, we will estimate the value π . As everyone knows, the area of a circle is πr^2 , and therefore for a unit circle the area is just π .

Algorithmic strategy

The formula for the unit circle is $x^2 + y^2 = 1$. Solving for y , we get the formula $y = \pm \sqrt{1 - x^2}$. For every value of x , there will be two values of y computed. If we want to calculate π we should compute the area under half of the circle between $x = -1$ and $x = 1$ and multiply that answer by 2. (In terms of integrals, we have $\pi = 2 \int_{-1}^1 \sqrt{1 - x^2} \, dx$.)



We will approximate this total area value (of the blue region in the diagram above) by adding up the areas of rectangles that approximately cover the area of that semicircle, as illustrated below.



This diagram shows 12 rectangles of equal width w that approximately cover the green semicircular area. Each rectangle is positioned over a subinterval of the x -axis interval $[-1, 1]$, and the height of a rectangle is the function's value at some value x_i in that rectangle's

subinterval. Thus, the area of a rectangle is $\sqrt{1 - x_i^2} * w$. We must add up the areas of all these rectangles, then double that sum to get the approximation of π .

The more rectangles we use, the greater accuracy we expect in our rectangle approximation to the exact area under the semicircular curve $y = f(x)$. Therefore, we will compute the sum with millions of thin rectangles in order to get better accuracy for our estimate of π .

Note: The accuracy of our estimate depends on how many rectangles we use, and to a lesser extent on how we choose the values x_i . We have illustrated in the diagram above choosing x_i as the midpoint of each subinterval, which visibly causes the sum of rectangle areas to be a reasonable approximation to the exact semicircular area we seek. If x_i represents the midpoint of the i th rectangle, then

$$x_i = -1 + (i + \frac{1}{2}) * w$$

where

$$h = 2.0/\text{NUM_RECTANGLES}$$

Reference

[https://cvw.cac.cornell.edu/\(X\(1\)S\(rfa5ssionwc00wyn0y31qejx\)\)/Cintro/default.aspx](https://cvw.cac.cornell.edu/(X(1)S(rfa5ssionwc00wyn0y31qejx))/Cintro/default.aspx)