Practical C/C++ Programming
Part 1

Feng Chen
HPC User Services
LSU HPC & LONI
sys-help@loni.org

Louisiana State University
Baton Rouge
June 27, 2018
Things to be covered today

- You should be able to understand basic C/C++ after the training
- Use them for your research.

- Introduction to C and C++ language
- Basic syntax and grammar
- Variables and data types, operators
- Control Flow
- Functions
- Input/Output control
Who are they?

Become a Hipster
Sell Stolen Ideas
Invent C
and UNIX

Praised by Media as Jesus of Computing

Without Steve Jobs (February 24, 1955 – October 5, 2011) we would have:
- No iProducts
- No over expensive laptops

Without Dennis Ritchie (September 9, 1941 – October 12, 2011) we would have:
- No Windows
- No Unix
- No C
- No Programs
- A large setback in computing
- No Generic-text Languages.
- We would all read in Binary...

They died in the same year and the same month but it seems only few notice the death of Dennis Ritchie compared to Steve Jobs.
C/C++ programming language overview

- C language
  - Developed by Dennis Ritchie starting in 1972 at Bell Labs

- C++ language
  - Developed by Bjarne Stroustrup starting in 1979 at Bell Labs

- C/C++ is most widely used programming languages of all time

- C/C++ compilers are available on most platforms, dominant on most science and engineering software packages

- Most of today's state-of-the-art OS and softwares have been implemented using C/C++. (Ref: [https://www.mycplus.com/featured-articles/top-10-applications-written-in-c-cplusplus/](https://www.mycplus.com/featured-articles/top-10-applications-written-in-c-cplusplus/))
C compiler overview

- **What is a compiler?**
  - A compiler is a computer program (or set of programs) that transforms *source code* written in a programming language (the source language) into another computer language (the target language, often having a binary form known as *object code*).

- **What does a compiler do?**
  - *Lexical analysis* (scanning), *syntax analysis* (parsing), *intermediate code generation*, *code optimization*, *target code generation*
  - *In short, translate C/C++ source code to binary executable*

- **List of common C compilers**
  - GCC GNU Project (MinGW, Cygwin)
  - Intel Compiler
  - PGI Compiler
  - Microsoft/Borland Compilers
  - XL C (IBM)
  - Xcode
Writing your first C Program

➤ hello_world.c

```c
#include <stdio.h>

/* This is our first C program. 
   for part 1 */
int main ( void ) {
   /* print a line to screen */
   printf( "Hello World!\n" );
   return 0;
}
```
Compile your first C program

- LONI and HPC users, start an interactive session:
  
  ```
  # example on Philip
  # ssh yourusername@philip.hpc.lsu.edu
  # start an interactive job (session)
  $ qsub -I -l nodes=1:ppn=8 -l walltime=02:00:00 -q workq -A your_allocation
  ...start the interactive job...
  $ cd
  $ git clone https://github.com/lsuhpchelp/cprog1.git
  $ cd cprog1/examples
  $ gcc hello_world.c
  ```

- Execute the program by typing:
  
  ```
  $ ./a.out
  Hello World!
  ```
For Non LONI/HPC users

- You can use online compiler for today’s tutorial:
  - https://www.onlinegdb.com/
  - Copy and paste code from the repository
    - https://github.com/lsuhpchelp/cprog1

Click Run to execute the code

Select C or C++
Structure of the hello_world.c file

- `#include` statements and preprocessor definitions
- Define `main()` function

```c
int main(void)
{
    Function body
    return 0;
}
```
The `#include` macro

- Header files: constants, functions, other declarations
- `#include <stdio.h>` — read the contents of the header file
- `stdio.h` - this is the **C Standard Input and Output Library** definition (below are some ugly details):

```c
#define FILE struct __file
#define stdin (__iob[0])
#define stdout (__iob[1])
#define stderr (__iob[2])
#define EOF   (-1)
#define fdev_set_udata(stream, u) do { (stream)->udata = u; } while(0)
#define fdev_get_udata(stream)   ((stream)->udata)
#define fdev_setup_stream(stream, put, get, rwflag)
...
```
Basic C syntax

- C is a **case sensitive** programming language: Var, var
- Each individual statement is ended with a semicolon “;”.
- Except inside a character string, whitespace (tabs or spaces) is never significant.
- All C statements are defined in free format, i.e., with no specified layout or column assignment. The following program would produce exactly the same result as our earlier example:

```c
#include <stdio.h>
int main() /*first program*/ {
    printf("Hello World\n"); return 0;
}
```

- Comments in C:
  - /* this is a single line comment */
  - /* This is
    a multiline comment */
  - Always use proper comments in your code.
  - Comments are completely ignored by compiler (test/debug code)
Some more details on printf()

- /* print formatted data to stdout (your screen) */
  
  int printf (const char * format, argument_list);

- If format includes format specifiers (start with %), the additional arguments following format are formatted & replacing the specifiers.

- Format specifier prototype: %[flags][width][.precision][length]specifier

- Common format specifiers:

<table>
<thead>
<tr>
<th>specifier</th>
<th>Output</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f</td>
<td>decimal float</td>
<td>3.456</td>
</tr>
<tr>
<td>%7.5f</td>
<td>decimal float, 7 digit width, precision 5</td>
<td>8.52000</td>
</tr>
<tr>
<td>%d</td>
<td>decimal integer</td>
<td>180</td>
</tr>
<tr>
<td>%s</td>
<td>String of characters</td>
<td>“hello world”</td>
</tr>
<tr>
<td>%e</td>
<td>decimal float, scientific notation (mantissa/exponent)</td>
<td>3.141600e+05</td>
</tr>
<tr>
<td>\n</td>
<td>new line</td>
<td></td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
<td></td>
</tr>
</tbody>
</table>
Some more details on printf()

- An example showing the `printf()` usage

```c
/* printf example showing different specifier usage */
#include <stdio.h>

int main()
{
    printf ("Characters: %c %c \n", 'a', 65);
    printf ("Decimals: %d %4d \n", 2014, 65);
    printf ("floats: %7.5f \t%f \t%e \n", 3.1416, 3.1416, 3.1416);
    printf ("%s \n", "hello world");
    return 0;
}
```
Variables and data types, operators
Data types in C

- Data types in C:
  - Basic types:
    - integers - `char`, `int`, short, long.
    - floating point - Defined using float and double.
    - void - no value is available
  - Derived types - (a) Pointer types, (b) Array types
  - Custom types - structure/union/enum/class, will be detailed in Part 2

- For science and engineering, mostly used types:
  - integer
  - floating point
  - In C there is no logical type (available in C++ as `bool`)
    - 0 (zero) as false
    - non-zero as true
### Integer Types: signed and unsigned

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage size</th>
<th>Value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>-128 to 127 or 0 to 255</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>1 byte</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>int</td>
<td>2 or 4 bytes</td>
<td>-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>2 or 4 bytes</td>
<td>0 to 65,535 or 0 to 4,294,967,295</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2 bytes</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>long</td>
<td>4 bytes</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4 bytes</td>
<td>0 to 4,294,967,295</td>
</tr>
</tbody>
</table>
Floating Point Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage size</th>
<th>Value range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4 byte</td>
<td>1.2E-38 to 3.4E+38</td>
<td>6 decimal places</td>
</tr>
<tr>
<td>double</td>
<td>8 byte</td>
<td>2.3E-308 to 1.7E+308</td>
<td>15 decimal places</td>
</tr>
<tr>
<td>long double</td>
<td>10 byte</td>
<td>3.4E-4932 to 1.1E+4932</td>
<td>19 decimal places</td>
</tr>
</tbody>
</table>

void Types

<table>
<thead>
<tr>
<th>Situation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>function returns as void</td>
<td>function with no return value</td>
</tr>
<tr>
<td>function arguments as void</td>
<td>function with no parameter</td>
</tr>
<tr>
<td>pointers to void</td>
<td>address of an object without type</td>
</tr>
</tbody>
</table>
Constants in C

- Constants refer to fixed values that the program may not alter during its execution.

### Integer constant

- `275 /* integer */`
- `215u /* unsigned int */`
- `85 /* decimal */`
- `31 /* int */`
- `31u /* unsigned int */`
- `31l /* long */`
- `31ul /* unsigned long */`

### Floating point constant

- `3.1416`
- `3.14159E-5 /* 3.14159 */`
- `2.1E+5 /* 2.1x10^5 */`
- `3.7E-2 /* 0.037 */`
- `0.5E7 /* 5.0x10^6 */`
- `-2.8E-2 /* -0.028 */`

### Character constant

- `'a' /* character 'a' */`
- `'Z' /* character 'Z' */`
- `\? /*? character */`
- `\ /*\ character */`
- `\n /*Newline */`
- `\r /*Carriage return */`
- `\t /*Horizontal tab */`

### String constant

- /* normal string */
- "hello, world"
- /* multi-line string */
- "c programming \
  language"
Define constants

Two ways to define constants in C
- Using `#define` preprocessor (defining a macro)
- Using the `const` key word (new standard borrowed from C++)

```c
#include <stdio.h>
/* define LENGTH using the macro */
#define LENGTH 5
int main()
{
    /*define WIDTH using const */
    const int WIDTH = 3;
    const char NEWLINE = '
';
    int area = LENGTH * WIDTH;

    printf("value of area : %d", area);
    printf("%c", NEWLINE);

    return 0;
}
```
Basic variable types

- A variable is a name given to a storage area.
- Each variable in C has a specific type, which determines the size and layout of the variable's memory;

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>A single character. (interchangeable with integer).</td>
</tr>
<tr>
<td>int</td>
<td>integer type.</td>
</tr>
<tr>
<td>float</td>
<td>A single-precision floating point value.</td>
</tr>
<tr>
<td>double</td>
<td>A double-precision floating point value.</td>
</tr>
<tr>
<td>void</td>
<td>Represents no type.</td>
</tr>
</tbody>
</table>
Define variables - variable names rules

- A variable name consists of any combination of alphabets, digits and underscores. Please avoid creating long, meaningless variable name.
- The first character of the variable name must either be alphabet or underscore. It should not start with the digit.
- No special symbols (including blanks or commas) other than underscore are allowed in the variable name.
- Examples:
  ```
  int count;
  float safety_factor;
  double normal_force;
  ```
Define and initialize variables

- C is a strong type language, variables must be declared before use.
- Syntax for defining variables:
  
  ```
  type list_of_variables_names;
  ```

- Examples of variables definition:
  ```
  int i, j, k;
  char item, name;
  float force, factor;
  double value;
  ```

- Variables initialized via assignment operator at declaration:
  ```
  int a = 31;
  float phi = 31.2345835;
  ```

- Can declare(initialize multiple variables at once:
  ```
  int a, b, c = 0, d = 51;
  ```
C reserved keywords

- C reserved words may not be used as constant or variable or any other identifier names.

<table>
<thead>
<tr>
<th>auto</th>
<th>else</th>
<th>long</th>
<th>switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>enum</td>
<td>register</td>
<td>typedef</td>
</tr>
<tr>
<td>case</td>
<td>extern</td>
<td>return</td>
<td>union</td>
</tr>
<tr>
<td>char</td>
<td>float</td>
<td>short</td>
<td>unsigned</td>
</tr>
<tr>
<td>const</td>
<td>for</td>
<td>signed</td>
<td>void</td>
</tr>
<tr>
<td>continue</td>
<td>goto</td>
<td>sizeof</td>
<td>volatile</td>
</tr>
<tr>
<td>default</td>
<td>if</td>
<td>static</td>
<td>while</td>
</tr>
<tr>
<td>do</td>
<td>int</td>
<td>struct</td>
<td>_Packed</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operators in C - 1

- **Arithmetic Operators**
  
  ```
  + , - , *, / 
  % /* mod */
  ++ /* increases integer value by one */
  -- /* decreases integer value by one */
  ```

- **Relational Operators**
  
  ```
  == /* equal */
  != /* not equal */
  > /* greater than */
  < /* less than */
  >= /* greater than or equal to */
  <= /* less than or equal to */
  ```

- **Bitwise Operators**
  
  ```
  &, |, ^, ~, <<, >>
  ```
Operators in C - 2

- **Assignment Operators**
  
  =  /* simple assignment */
  
  /* Reverse Polish Notation (RPN) */
  
  +=  /* C += A <=> C = C + A*/
  
  -=  /* C -= A <=> C = C - A*/
  
  *=  /* C *= A <=> C = C * A*/
  
  /=  /* C /= A <=> C = C / A*/

- **Logical Operators**
  
  !  /* not */
  
  &&  /* and */
  
  ||  /* or */
Misc Operators

`sizeof()` /* Returns the size of an variable. */

`&` /* Returns the address of an variable. */

`*` /* Pointer to a variable. */

`?=:` /* (ternary conditional operator),
condition is true ? Then value X : Otherwise value Y*/

`,` /* comma separates expression and evaluates to the last */
Operators example

- Define variables x and y:
  ```
  int x, y;
  ```

- Simple arithmetic:
  ```
  x+y, x-y, x*y, x/y, x%y
  ```

- C statement examples:
  ```
  x+y, x-y, x*y, x/y, x%y;
  x++, x--;
  y = x+5*x/(y-1);
  x += y; /* x=x+y Reverse Polish notation (RPN) */
  x -= y, x *= y; /* Comma operator */
  x>0?y=x+1:y=x-1; /* ternary operator */
  ```

- Use parentheses to override order of evaluation
Type Conversion

- You can convert values from one type to another explicitly using the cast operator: `(type_name) expression`
  - `(float)3`

- Arithmetic Conversion: `int->float->double->long double`

```c
#include <stdio.h>

main() {
    int a = 4, b = 3;
    float c;
    c = a / b;
    /* make sure you are doing the right conversion */
    printf("c = %f\n", c);
    c = b / a;
    printf("c = %f\n", c);
    c = (float)a / b;
    printf("c = %f\n", c);
}
```
char and int type in C

- In C **char** and **int** are interchangeable, C allows assign **char** to **int**, and vice versa (char_int.c):

```c
#include <stdio.h>
/* interchangeability between char and int */
int main() {
    char a=120; /* ascii value for 'x' is 120 */
    int  b='y'; /* ascii value for 'y' is 121 */
    printf("%c,%c\n",a,b);
    printf("%d,%d\n",a,b);
    printf("a-b=%d\n",a-b);
    return 0;
}
```

- Memory layout of a and b:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Using variables and operators-Example

```c
#include <stdio.h>

int main() {
    int i = 3;
    int j = 4;
    int x, y;
    int kronecker_delta;
    float a = 4.5;
    double b = 5.25;
    double sum;

    x=1, y=2, x = (x, y);
    /* 1. calculate the kronecker_delta using ?= */
    kronecker_delta = (i==j)?1:0;
    /* 2. calculate the sum of a and b */
    sum = a+b;

    printf("x= %d, y= %d\n", x, y);
    printf("i/j= %d\n", i/j);
    printf("j/i= %d\n", j/i);
    printf("kronecker_delta= %d\n", kronecker_delta);
    printf("a+b= %f\n", sum);

    return 0;
}
```
Blocks and Compound Statements

- A simple statement ends in a semicolon ";":
  
  
  \[
  \text{area} = 2.0 \times \text{pi} \times \text{rad} \times \text{rad};
  \]

- Use curly braces to combine simple statements into compound statement/block, no semicolon at end

- Variables can be defined inside block, example:

  ```
  
  \{
  \hspace{10pt} \text{double area;}
  \hspace{10pt} \text{double rad=1.0;}
  \hspace{10pt} \text{double pi=3.1415926;}
  \hspace{10pt} \text{area}=2.0*\text{pi}*\text{rad}*\text{rad};
  \}
  
  ```

- Block can be empty {}

- Usage? See next few slides
Control Flow
Control flow

- Conditional Statements (decision making/selection)
  - if...else if...else
  - switch
- Loops
  - for
  - while
  - do while
The if...else if...else statement

- An if statement can be followed by an optional else if...else statement.
  - Evaluate condition
  - If true, evaluate inner statement
  - Otherwise, do nothing

```c
if(boolean_expression 1) {
    /* executes when the boolean expression 1 is true */
} else if( boolean_expression 2){
    /* optional, executes when the boolean expression 2 is true */
} else if( boolean_expression 3){
    /* optional, executes when the boolean expression 3 is true */
} else{
    /* optional, executes when the none of the above condition is true */
}
```
The if statement example

```c
#include <stdio.h>

int main () {
    /* local variable definition */
    int a = 100;

    /* check the boolean condition */
    if ( a == 10 ) {
        printf("Value of a is 10\n");
    }
    else if ( a == 20 ) {
        printf("Value of a is 20\n");
    }
    else {
        /* if none of the conditions is true */
        printf("None of the values is matching\n");
    }
    printf("Exact value of a is: %d\n", a);

    return 0;
}
```
The switch statement

- A switch statement allows a variable to be tested against a list of values. Each value is called a case.
- *Without break, the program continues to evaluate the next case*

```c
switch(expression){
    case constant-expression :
        statement(s);
        break; /* optional */
    case constant-expression :
        statement(s);
        break; /* optional */
    /* you can have any number of case statements */
    default : /* Optional */
        statement(s);
}
```
The switch statement example

What is the expected output of this code?

```c
/* switch_statement_grade.c */
#include <stdio.h>
int main (){
    /* local variable definition */
    char grade = 'A';
    /* what is the expected output? */
    switch(grade) {
        case 'A' : printf("Excellent!\n");
        case 'B' : printf("Well done\n");
        case 'C' : printf("You passed\n");
        case 'F' : printf("You failed\n");
        default : printf("Invalid grade\n");
    }
    printf("Your grade is  %c\n", grade);
    return 0;
}
```
Nested conditional statements

- Conditional statements can be nested as they do not overlap:

```c
if( boolean_expression 1 ) {
  if(boolean_expression 2) {
    /* Executes when the boolean expression 2 is true */
    /* nested switch statement */
    switch(expression){
      case constant-expression :
        statement(s);
        break; /* optional */
      case constant-expression :
        statement(s);
        break; /* optional */
      /* you can have any number of case statements */
      default : /* Optional */
        statement(s);
    }
  }
}
```
For loops

For loops in C:

– The init step is executed first and only once.
– the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute, the loop exits.
– the increment statement executes after the loop body.
– The loop continues until the condition becomes false

```c
for ( init; condition; increment ) {
    loop body;
}
```
while and do...while loops

- while loops are similar to for loops
- A while loop continues executing the code block as long as the condition in the while holds.

```c
while (condition) {
    statement(s);
}
```

- do...while loop is guaranteed to execute at least one time.

```c
do {
    statement(s);
} while (condition);
```
#include <stdio.h>

int main ()
{
    int i;
    /* for loop execution */
    for(i = 0; i < 5; i++) {
        printf("for loop i= %d\n", i);
    }
    i=0;
    /* while loop execution */
    while( i < 5 ) {
        printf("while loop i: %d\n", i);
        i+=1;
    }
    i=1;
    /* do-while loop execution */
    do {
        printf("do while loop i: %d\n", i);
        i=i+1;
    }while( i < 0 );

    return 0;
}
Nested loops in C

➢ All loops can be nested as long as they do not overlap:

```c
/* nested for loops*/
for (init; condition; increment) {
    for (init; condition; increment) {
        statement(s);
    }
    statement(s);
}

/* nested while loops*/
while(condition) {
    while(condition) {
        statement(s);
    }
    statement(s);
}

/* nested do while loops*/
do {
    while(condition) {
        statement(s);
    }
    statement(s);
} while( condition );

/* mixed type loops*/
while(condition) {
    for (init; condition; increment) {
        statement(s);
    }
    do {
        statement(s);
    } while( condition );
    }
    statement(s);
} while( condition );
```
#include <stdio.h>

int main()
{
    int i, j, k;
    printf("i j k\n");
    /* examples for nested for loops */
    for (i=0; i<2; i++)
        for (j=0; j<2; j++)
            for (k=0; k<2; k++)
                printf("%d %d %d\n", i, j, k);
    return 0;
}
Loop Control Statements

Loop control statements change execution from its normal sequence:

- **break** statement
  - terminates the *entire loop* or switch statement
- **continue** statement
  - causes the loop to skip the remainder of the loop body for the *current iteration*.
- **goto** statement
  - **Avoid** using this in your program
#include <stdio.h>

int main () {
    /* local variable definition */
    int a = 0;

    /* while loop execution */
    while( a < 10 ) {
        if( a > 5 ) {
            /* terminate the loop using break statement */
            break;
        }
        if (a==3) {
            a++;
            /* terminate the current iteration using continue statement */
            continue;
        }
        printf("value of a: %d\n", a);
        a++;
    }
    return 0;
}
Derived Data Types
Arrays in C

- Arrays are special variables which can hold more than one value using the same name with an index.
- Declaring Arrays: `type arrayName[arraySize];`

```c
/* simply define the arrays */
double balance[10];
float atom[1000];
int index[5];
```

- C array starts its index from 0
- Initialize the array with values:

```c
/* initialize the array with values*/
int index[5]={5, 4, 6, 3, 1};
double value[]={5.3, 2.4, 0.6, 1.3, 1.9};
```

- Access array values via index:

```c
/* access the array values*/
int current_index=index[i];
double current_value=value[current_cell_index];
```
Be careful in accessing C array

- C arrays are a sequence of elements with contiguous addresses.
- There is no bounds checking in C.
- Be careful when accessing your arrays
- Compiler will not give you error, you will have *undefined* runtime behavior:

```c
#include <stdio.h>
int main() {
    int index[5]={5, 4, 6, 3, 1};
    int a=3;
    /* undefined behavior */
    printf("%d\n",index[5]);
}
```
# Multidimensional Arrays

- **General form of a multidimensional array declaration in C:**
  
  ```c
  datatype name[size1][size2]...[sizeN];
  ```

- **Declaring 2D and 3D arrays:**

  ```c
  float array2d[4][5];
  double array3d[2][3][4];
  ```

- **Initialize multidimensional arrays**

  ```c
  int a[3][4] = { /* 2D array is composed of 1D arrays*/
      {0, 1, 2, 3}, /* initialize row 0 */
      {4, 5, 6, 7}, /* initialize row 1 */
      {8, 9, 10, 11}}; /* initialize row 2 */
  ```

<table>
<thead>
<tr>
<th>row 0</th>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
</tr>
</thead>
</table>
  | int a[3][4] = { /* 2D array is composed of 1D arrays*/
  |      |      |      |      |
  |      | {0, 1, 2, 3}, /* initialize row 0 */
  |      | {4, 5, 6, 7}, /* initialize row 1 */
  |      | {8, 9, 10, 11}}; /* initialize row 2 */
  | a[0][0]=0 | a[0][1]=1 | a[0][2]=2 | a[0][3]=3 |
  | a[1][0]=4 | a[1][1]=5 | a[1][2]=6 | a[1][3]=7 |
Something to remember for C arrays

- *Row-major* order and *Column-major* order describe methods for storing multidimensional arrays in *linear memory*.
- In C/C++ programming language, *Row-major* order is used.
- Consider the below array:

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
\end{bmatrix}
\]

- Declared in C as:

```c
int A[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

- In C the array is laid out contiguously in linear memory as:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- Fortran is Column-major order.
Arrays-Example

```c
#include <stdio.h>
#define N 10

int main() {
    /* TODO: find the max, min, sum of the 10 values */
    double sum, max, min;
    int i=0;
    int a[N]={13, 14, 15, 16, 17, 16, 15, 14, 13, 11};

    sum=min=max=a[0];
    for (i=1;i<N;i++) {
        if (max<a[i]) max=a[i];
        if (min>a[i]) min=a[i];
        sum += a[i];
    }
    printf("The max value is: %f\n" max);
    printf("The min value is: %f\n", min);
    printf("The sum value is: %f\n", sum);
    return 0;
}
```
Strings in C

- Strings in C are a special type of array: *array of characters* terminated by a null character `\0`.

```c
/* define a string */
char str[7]={'H','E','L','L','O','!','\0'};
char str1[]="HELLO!";
```

- Memory presentation of above defined string in C/C++:

<table>
<thead>
<tr>
<th>str[]</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‘H’</td>
<td>‘E’</td>
<td>‘L’</td>
<td>‘L’</td>
<td>‘O’</td>
<td>‘!’</td>
<td>‘\0’</td>
</tr>
</tbody>
</table>

- C uses built-in functions to manipulate strings:

```c
/* C sample string functions */
strcpy(s1, s2); /* Copies string s2 into string s1. */
strcat(s1, s2); /* Concatenates string s2 onto the end of string s1. */
strlen(s1);    /* Returns the length of string s1. */
strcmp(s1, s2); /* Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. */
```
#include <stdio.h>
#include <string.h>
#define N 30

int main ()
{
    char str1[N] = "C program ";
    char str2[N] = "is great!";
    char str3[N];
    int len;

    /* copy str1 into str3 */
    strcpy(str3, str1);
    printf("strcpy( str3, str1) : %s\n", str3);

    /* concatenates str1 and str2 */
    strcat(str1, str2);
    printf("strcat( str1, str2): %s\n", str1);

    /* total length of str1 after concatenation */
    len = strlen(str1);
    printf("strlen(str1) : %d\n", len);

    return 0;
}
Structures

- User-defined type in C: **struct**, union and enum
- A C **struct** is an aggregate of elements of (nearly) arbitrary types.
- Structures are the basic foundation for objects and classes in C++.
- Structures are used for:
  - Passing multiple arguments in and out of functions through a single parameter
  - Data structures such as linked lists, binary trees, graph, and more
- Syntax for defining structure:

```c
/* syntax for defining structure */
struct [structure tag] /* tag is optional */
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```
How to use struct

- Example of defining a “Point” struct

  ```c
  /* define a structure “Point” */
  struct Point {
    int index;
    char tag;
    double x;
    double y;
  };
  
  /* define two struct Point variables */
  struct Point p1, p2, p3;
  
  Here is how we access the struct Point type variables, using the ".":
  p1.index=0; /* access members of p1 with dot “.” operator */
  p1.tag = 'a';
  p1.x = 0.0;
  p1.y = 0.0;
  p3 = p1; /* assign struct variable p1 to variable p3 */
  ```
Using typedef to define new variables

- C provides a keyword called `typedef` to name a new variable type (note that `typedef` does not create new types).

```c
typedef existing_type new_type_name;
```

- Use `typedef` with `struct` in the previous example:

```c
typedef struct Point point;
typedef struct Point { /* alternative way to define Point*/
  int index;
  char tag;
  double x;
  double y;
} Point;
```

- `typedef` can also be used to give alias to existing variable types:

```c
typedef double real; /* typedef float real; easy switch between precisions*/
```

- Use the newly defined type to define your variables, e.g.:

```c
real x; /* x is actually double defined above */
Point p1, p2, p3; /* p1, p2 and p3 are struct Point */
```
Functions
Functions

- A function is a group of statements that together perform a task.
- Every C program has at least one function, which is `main()`.
- Functions receive either a fixed or variable amount of arguments.
- Functions can only return one value, or return no value (void).
- In C, arguments are **passed by value** to functions.
- How to pass by reference? - **Pointers** (we will detail it in Part 2).
- Functions are defined using the following syntax:

```c
return_type function_name(type0 param0, type1 param1, ..., typeN paramN)
{
    function body
}
```

- Function **declaration**: declare function's name, return type, and parameters.
- Function **definition**: provides the actual body of the function.
Function definition

- **Return Type**: Function’s return type is the data type of the value the function returns. When there is no return value, return **void**.

- **Function Name**: This is the actual name of the function.

- **Parameter/argument list (optional)**: The parameter list refers to the type, order, and number of the parameters of a function. A function may contain no parameters.

- **Function Body**: The function body contains a collection of statements that define the function behavior.

- **Example for function definition**:

```c
/* find the max between two numbers */
int find_max(int a, int b)
{
    /* function body */
    int t;
    if (a > b) t = a;
    else t = b;
    return t;
}
```
Functions-Example

#include <stdio.h>

/* function declaration */
int max(int a, int b);

int main() {
    /* local variable definition */
    int a = 100, b = 200, ret;
    /* calling a function to get max value */
    ret = find_max(a, b);
    printf( "Max value is : %d\n", ret );
    return 0;
}

/* function returning the max between two numbers */
int find_max(int a, int b) {
    /* function body */
    int t;
    if (a > b) t = a;
    else t = b;
    /* rewrite the above using the ternary operator */
    /* t=(a>b)?a:b; */
    return t;
}
Input and Output
Input/Output

- Input means to feed data into program.
  - This can be given in the form from
    - screen (stdin)
    - file
  - C uses **built-in functions** to read given input and direct it to the program

- Output means to display data to:
  - file (C treats all devices as files):
    - screen (stdout, stderr)
    - printer
    - file
  - C uses a set of **built-in functions** to output the data on the computer screen as well as you can save that data in text or binary files.
**C Input/Output built-in functions -1**

- **getchar() and putchar()**

  /*reads the next available character from screen and returns the same character*/
  int getchar(void);
  /* puts the character "c" on the screen and returns the same character. */
  int putchar(int c);

- **gets() and puts()**

  /* reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF. */
  char *gets(char *s);
  /* writes the string s and a trailing newline to stdout. */
  int puts(const char *s);
C Input/Output built-in functions -2

- **scanf() and printf()**

```c
/* reads input from the standard input stream stdin and scans that input according to the format string. */
int scanf(const char *format, ...);
/* writes output to the standard output stream stdout and produces output according to the format string. */
int printf(const char *format, ...);
```

- **fscanf() and fprintf() - file operations, see next few slides**

```c
/* reads input from file fp and stores them according to the format string. */
int fscanf(FILE * fp, const char * format, ... );
/* writes output to file fp according to the format string. */
int fprintf(FILE * fp, const char * format, ... );
```
C Input/Output example

/* io_example */
#include <stdio.h>

int main() {
    char str[100];
    int i;
    float a;
    double b;
    printf( "Enter string i(int) a(float) b(double):
" );
    /*
    1. note the & sign (get the address of the variable) before i,a,b
    2. question, why there is no address sign before the str?
    3. note the %lf when reading double
    */
    scanf("%s %d %f %lf", str, &i, &a, &b);
    printf( "
You entered: %s, %d, %f, %lf
", str, i, a, b);
    return 0;
}
File Input/Output

- Two types of files:
  - text file (ASCII) /*we will only talk about text file in this training*/
  - binary file
- Similar to standard I/O, C uses built-in functions for File I/O
- Opening a file

```c
/* use fopen() function to create a new file or to open an existing file,
   the call will initialize a FILE object */
FILE *fopen( const char * filename, const char * mode );
/* filename: string for the file name
   mode: controls the file access mode */
```

- Closing a file:

```c
/* closing a file, *NEVER* forget to close a file after opening */
int fclose( FILE *fp );
```
More on file access mode:

```c
/* file access modes*/
"r", "w", "a", "w+", "r+", "a+
```

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Read only. The file pointer is placed at the beginning of the file.</td>
</tr>
<tr>
<td>w</td>
<td>Write only. The file pointer will be at the beginning of the file.</td>
</tr>
<tr>
<td>a</td>
<td>Append, The file pointer is at the end of the file if the file exists.</td>
</tr>
<tr>
<td>t</td>
<td>text mode</td>
</tr>
<tr>
<td>b</td>
<td>binary mode</td>
</tr>
<tr>
<td>+</td>
<td>read and write</td>
</tr>
</tbody>
</table>
File Input/Output Example

- The “file_io.c” example reads a series of vectors from “vector.in”, calculates the length of each vector and then outputs the length results to a file named “vector.out”
- Compile your program with “gcc file_io.c -lm” with math library

```c
/* open the "vector.in" in read mode */
if ((fp=fopen("vector.in", "r"))==NULL) exit(1);
 fscanf(fp,"%d",&num_vec);
if (num_vec>N) {
    fprintf(stderr, "out of bound error");
    exit(1);
}
for (i=0;i<num_vec;i++) {
    /* read the vectors */
    fscanf(fp,"%f %f %f",&vx,&vy,&vz);
    v_length[i]=sqrt(vx*vx+vy*vy+vz*vz);
}
fclose(fp);
...```
Part 2 Outline (proposed)

- Pointers in C/C++
- User defined types
- C++ basics and Objected Oriented Programming concepts
- Standard Template Library (STL)
Exercise 0

1. Complete the C code for \( \pi \) value evaluation. \texttt{calc_pi.c}

2. Write a function to calculate \( \pi \), then call this function from main()

Hint: We can use the following equation to calculate the value of pi:

\[
\int_{0}^{1} \frac{4}{1 + x^2} dx = 4 \cdot \arctan(x) \bigg|_{0}^{1} = \pi
\]

The numerical integration:

\[
\pi \approx \sum_{i=1}^{N} \frac{4}{1 + x_i^2} \Delta x
\]
Exercise 1

- Finding the root of a polynomial equation using the bisection method, you can refer to the details from the wiki page:
  - Create a function that calculates: \( f(x) = x^3 - x - 2 \)
  - Find the solution for \( x = [1.0, 2.0] \)

- Source code: `bisection.c`
Exercise 2

- Calculate the result of a constant times a vector plus a vector:
  where \( a \) is a constant, \( \vec{x} \) and \( \vec{y} \) are one dimensional vectors

\[
\vec{y} \leftarrow a\vec{x} + \vec{y}
\]
Exercise 3

3. Complete the C code for matrix multiplication

\[ A \cdot B = C \]

where:

\[ a_{i,j} = i + j \]

\[ b_{i,j} = i \cdot j \]

\[ c_{i,j} = \sum_{k} a_{i,k} \cdot b_{k,j} \]