

Revision

Song Liu (song.liu@bristol.ac.uk)
GA 18, Fry Building,
Microsoft Teams (search "song liu").

Lecture 1, Introduction of Computing

Key Objectives

Upon completion of this unit you should:

1. *Understand* the workflow of computer programming and appreciate computer as a data processing tool.
2. *Program, debug, document* and *test* basic algorithms in C(++) and R, with appropriate coding paradigms.
3. *Decide* which programming language to use when faced with a computing task.

Assessed Coursework 1

Deadline: Friday, Week 13.

Task: Given a data set ([MNIST](#)) containing images of handwritten digits, implement a simple classification algorithm ([k-nearest neighbor](#)), which labels a test image with "0", "1", "2".... "9". Details will be announced in a few weeks.

Dataset:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

Input:



Output:

"4"

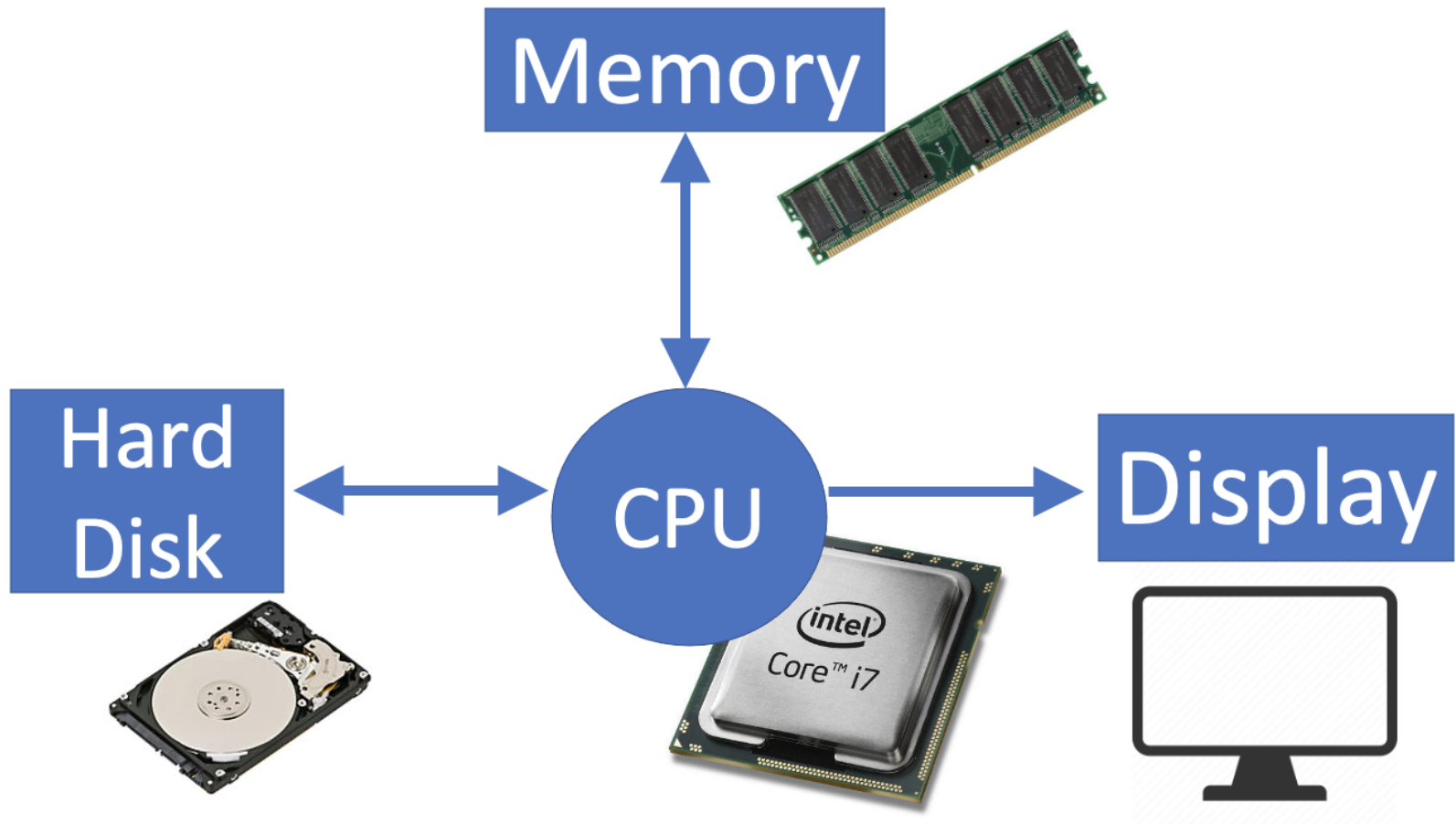
Assessed CW Marking Criteria:

- **5%:** You have submitted a C file.
- **10%:** Your code can compile and run **without any error**.
 - It will be tested using `gcc` in the lab pack.
 - Erratic behavior includes **crash, infinite loop**.
- **10-30%:** You have attempted the coursework "toward the right direction". However, your code does not produce any correct output given specific inputs within reasonable amount of runtime.
- **30-50%:** Your code produces partially correct outcome. Though it may suffer from some issues such as memory leak.

Assessed CW Marking Criteria:

- **50%-60%:** Your code produces mostly correct output and uses functions/classes dividing functionalities of your program.
- **60%-70%:** Your code produces the correct output and code is nicely formatted and comments are made to improve the readability of your code.
- **70%+:** Not only your code produces the correct output, but it does so using highly readable and efficient algorithms. Your code is nicely formatted and comments to each function/class. No memory leak or other unpredictable runtime issues.

How does Computer Work?



von Neumann Architecture

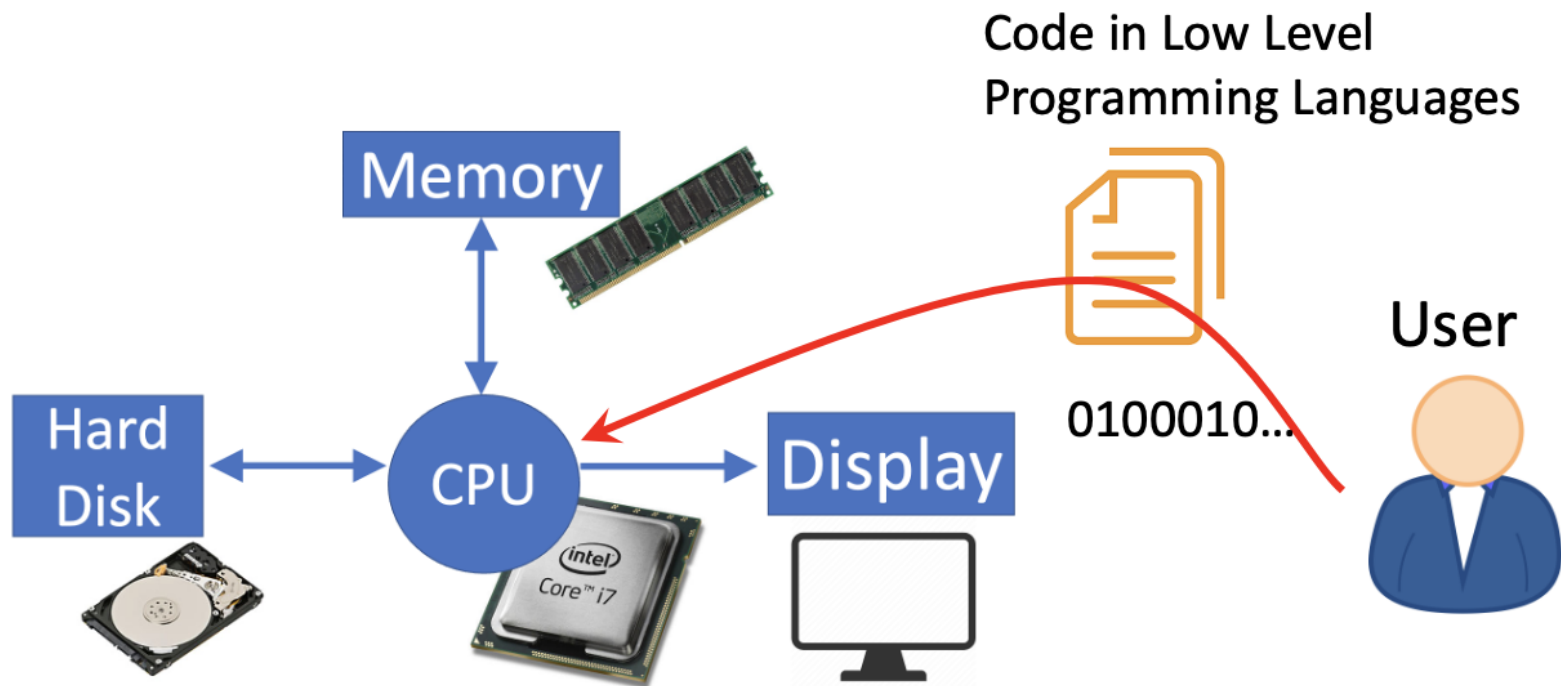
- **Central Processing Unit (CPU)**
 - Performs computational tasks.
 - Controls Input/Output (IO) devices.
 - Maintains data stored in the memory.
- **Memory**
 - Stores program/data being used by CPU temporarily.
- **IO Devices**
 - Hard disk
 - Display
 - Camera
 - Touch Screen, etc.

What is Programming?

- Programming = **writing a list of instructions to be executed on the CPU.**
- The list of instructions is called the "code".
- The language used to write the code is called **programming language.**

Low-level Programming Language

- Coder can program in **machine code**.
- Then the code can be **directly executed** on the CPU requiring no (or very little) translation.
- Machine code (and its more human friendly variants) are referred to as "Low-Level Programming Languages".



Low-level Programming Language

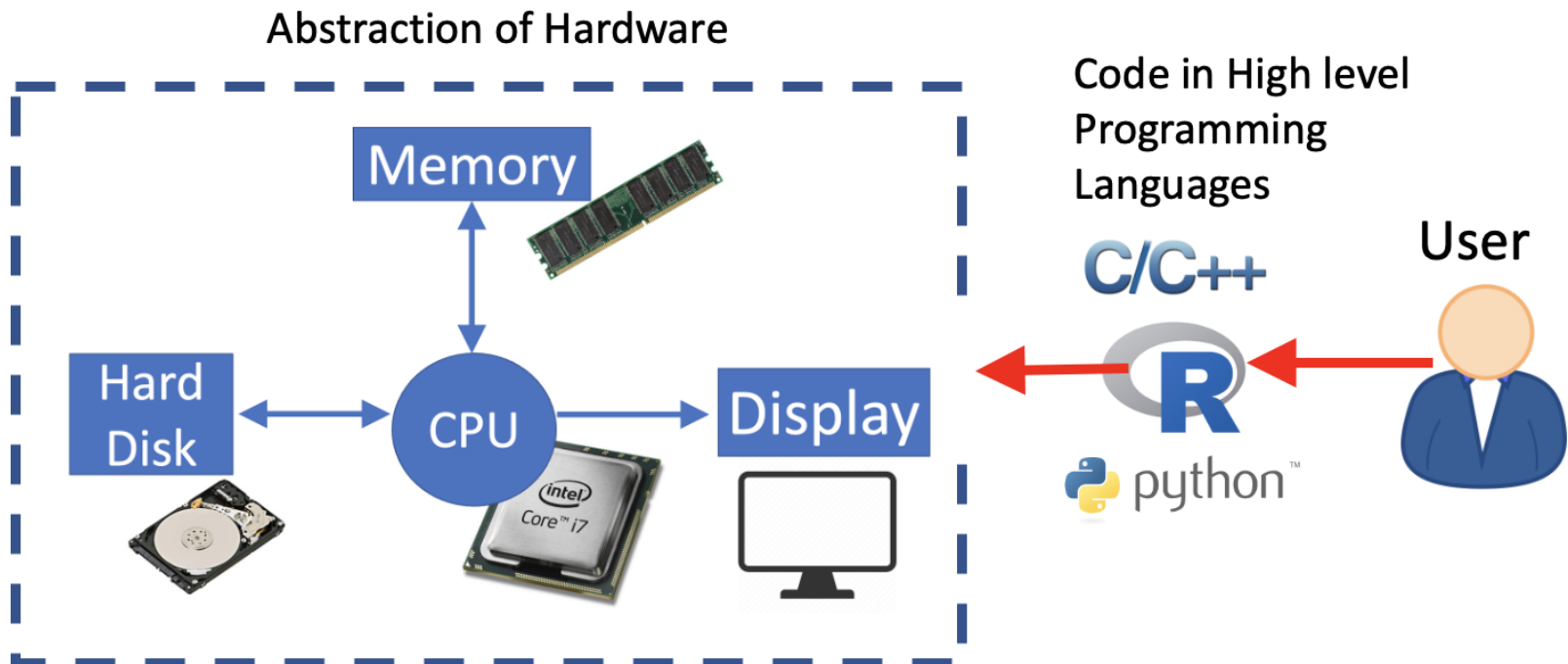
- **Advantages** of Low-level Programming Language:
 - total control of hardware
 - efficient
- **Disadvantages** of Low-level Programming Language:
 - can damage hardware
 - hard to read/learn
 - not portable.

High-level Programming Language

- Coder can program in a more natural language, which is then **translated** into machine code.
- This translation process is called "compilation" and the software that performs this translation is called "**complier**".

Abstraction of Hardware

- High-level language provides an **abstraction of hardware**.
- Coder interfaces with this abstraction rather than directly program for the underlying CPU and hardware.



Abstraction of Hardware

- Writing code for the abstraction has many advantages:
 - Code is portable.
 - Coding is easier since less hardware maintenance.
 - Enhancing the security of the system.

High-level Programming Language

- **Advantages** of High-level Programming Languages
 - Easy to learn/read.
 - Code is "Portable".
- **Disadvantages** of High-level Programming Languages
 - Less efficient.
 - Cannot directly communicate with hardware.

Example Code: C

Example C code for printing "Hello World!" on almost all CPUs.

```
//filename: main.c
#include <stdio.h>

void main(){
    printf("Hello World!\n");
}
```

Compilation:

```
gcc main.c -o main.out
```

Execution:

```
./main.out
Hello World!
```


printf function

- Know the basic usage of `printf` function and is able to use it in your C code.
- What are specifiers (`%`) for different types of data. Know at least specifiers for
 - integer
 - float point numbers
 - string
 - character

Lecture 2, Function and Recursion

What is a Function in Programming?

- Functions are individual building blocks of your program that accomplish specific tasks.
 - Function helps you divide your code into smaller, more **manageable and readable** pieces.
 - Code in a function ******is only executed when its host function is "called". ******
- Some functions take *input arguments*.
- Some functions return *an output value*.
- Some functions do not have input or output.

How to Write a Function Definition?

1. A function definition starts by indicating the **return type** (`void` means no return value will be produced.).
2. Followed by the **function name**.
3. The **Input arguments** come after that, inside `(` and `)`.
4. The **body of the function** is enclosed by `{` and `}`.

```
...  
return_type function_name(input argument deceleration){  
    function body  
}
```

Read Example Code Carefully.

PS: You **CANNOT** write a function inside another function!

You should **NOT** do:

```
void main(){  
    int cal_length(...){  
        ...  
    }  
}
```

Instead do

```
int cal_length(...){  
    ...  
}  
void main(){  
    ...  
}
```

main with Inputs and an Output

The `main` function can also take inputs and return output.

```
#include <stdio.h>
int main(int nargs, char* args[]){
    printf("%s\n", args[0]);
    return 0;
}
```

- `main` takes two inputs: `nargs` and `args`. These are passed on to `main` **from** the OS.
- In this example, `main` returns an integer value to OS. It returns 0 if succeeded, otherwise, returns a non-zero value.
- `main` is the only entrance of your program!
 - DO NOT define more than one `main` in your code!!!

Function Body

```
... function_name(...){  
    declaration of variables  
    ...  
  
    other statements  
}
```

Data Types in C

- Some data types are:
 - `int` or `long` : integers
 - `float` or `double` : float point numbers
 - `char` : characters
- Each declaration tells the compiler: "reserve __ bytes of memory for this variable when function is running!".
- On modern PCs (and most smartphones):
 - `int` and `float` occupies 4 bytes of memory.
 - `long` and `double` occupies 8 bytes of memory.
 - `char` occupies 1 byte of memory.

Declarations

Once a variable is declared, **the variable is assigned a memory location by the compiler**. If the variable is uninitialized, **the variable can contain whatever (rubbish) value** that is already in that memory location!

Expressions

- Computing operations are done using **expressions**:
 - `G*m1*m2/dist/dist`
 - Each expression has a value. The value of `G*m1*m2/dist/dist` is its computation outcome.
- `m1 = 1.0, m2 = 2.0, gravity = G*m1*m2/dist/dist` are all **assignment expressions**.
 - It assigns the value of expression on the RHS to the variable on the LHS.
 - **The equality sign does not represent equality.**

Relational and Logical Expressions

- Relational Expression

- expressions like `a > b`, `a != 1` and `a == 1 && b = 2` are relational expressions. They have values 0 or 1.
- `2 > 1` has value 1,
- `2 >= 3` has value 0.
- `1 == 1` has value 1, where `==` means equal.
- `1 != 1` has value 0, where `!=` means not equal.

- Logical Expression

- `1 == 1 && 1 > 2` has value 0, where `&&` means logical AND.
- `1 == 1 || 1 > 2` has value 1, where `||` means logical OR.

Relational and Logical Expressions

- In C, logical FALSE is expressed by an integer 0.
- Logical TRUE is expressed by any non-zero integer.
- Read this tutorial [Operators in C](#).
- [Precedence of Operators](#)

Calling a Function

- You can call a function and obtain its returned value **after its definition.**
- Calls are made using the function name with all input values **in the same order they are declared!**
- Read the code example in the original slides.

Calling a Function: Recursion

- A function can call itself!
 - This is called **recursive call**.

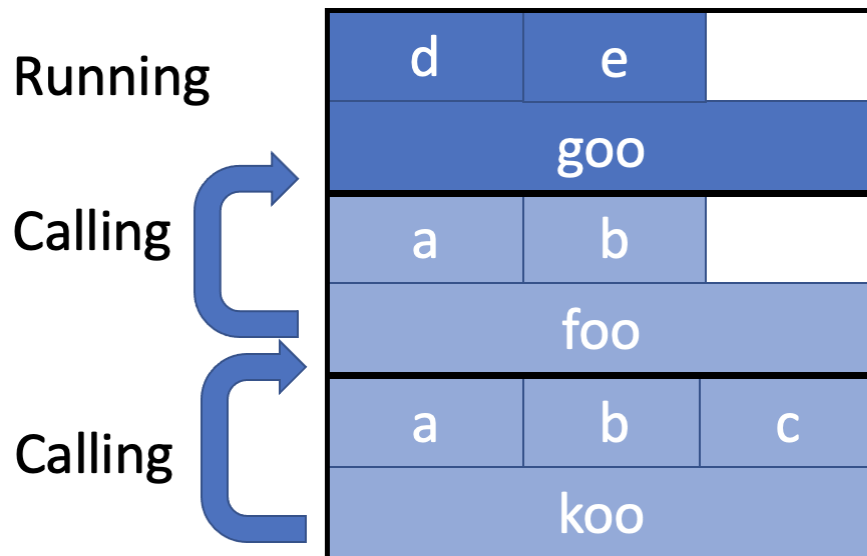
```
#include <stdio.h>
void countdown_to_1(int n){
    if (n<1) {;}
    else{
        printf("%d\n",n);
        countdown_to_1(n-1);
    }
}
void main(){
    //It prints 10, 9, 8 ... 1
    countdown_to_1(10);
}
```

Stack Memory Allocation

- When the function is being executed on the CPU, its data (such as variables declared in the function) are temporarily stored in the memory.
- The memory region for storing function data in the current program is called "stack".
- When a function is called, its data is added to the top of the stack. Your program can access them.
- When a function finishes its execution, its data is removed from the stack and the space it occupies is freed for future calls of functions.

Stack Memory Allocation

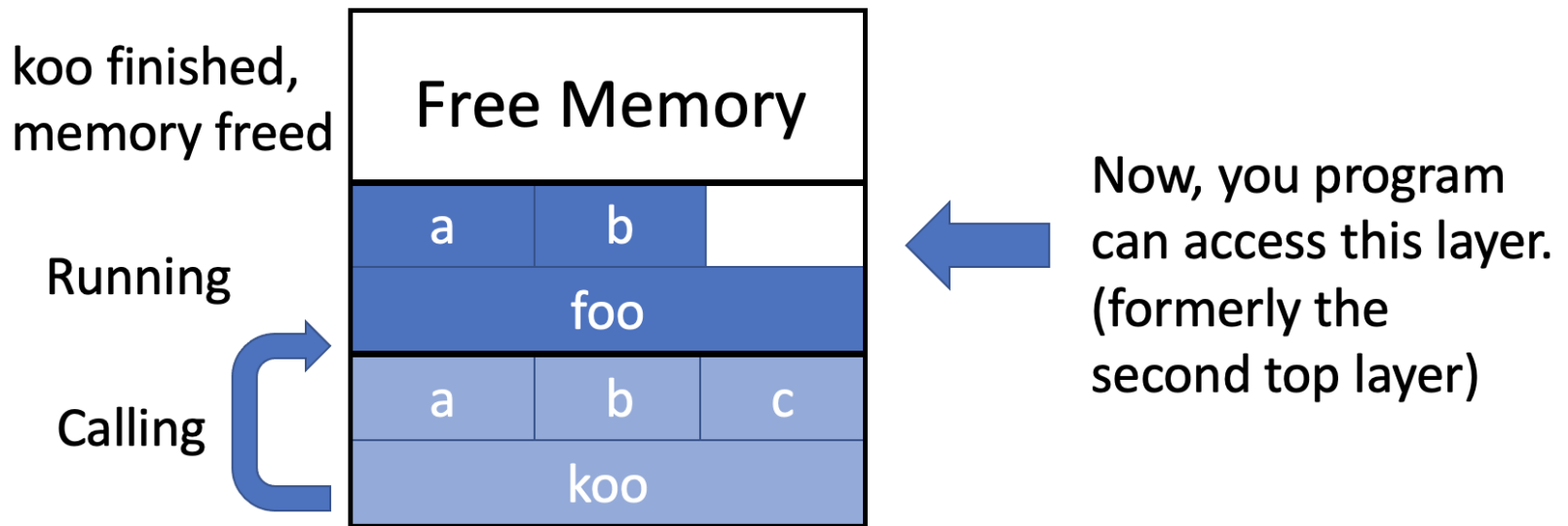
Consider a situation where function `koo` calls `foo` and `foo` calls `goo`. Below is the stack while `goo` is running.



Your program can only access the top layer of the stack while `goo` is running!

Stack Memory Allocation

When `goo` finishes running, its memory is freed.



When `foo` finishes running, its memory will be freed too and only variables in `koo` will be accessible.

Local Variables

Variables declared inside the function are called **local variables** (This includes all input argument variables!).

- They can only be accessed by the function where it is defined.
 - They cannot be accessed by other functions.
- **Why?** The program can only access the top layer of the stack, which stores variables of the function that is **currently running**.
- In the "koo-foo-goo" example, your program cannot access `a` and `b` defined in `koo` while `foo` is running.

Stack Memory Allocation

Stack is a highly efficient memory allocation/release mechanism.

- The memory allocation and release are all automatically handled by the OS.
- However, there is only a limited stack space for each program (determined by the OS). If a single function occupies a large memory space, or the call stack gets too "tall", we may run out of stack memory and an execution error will be raised by the OS.
 - This out-of-memory error is called "**Stack Overflow**".

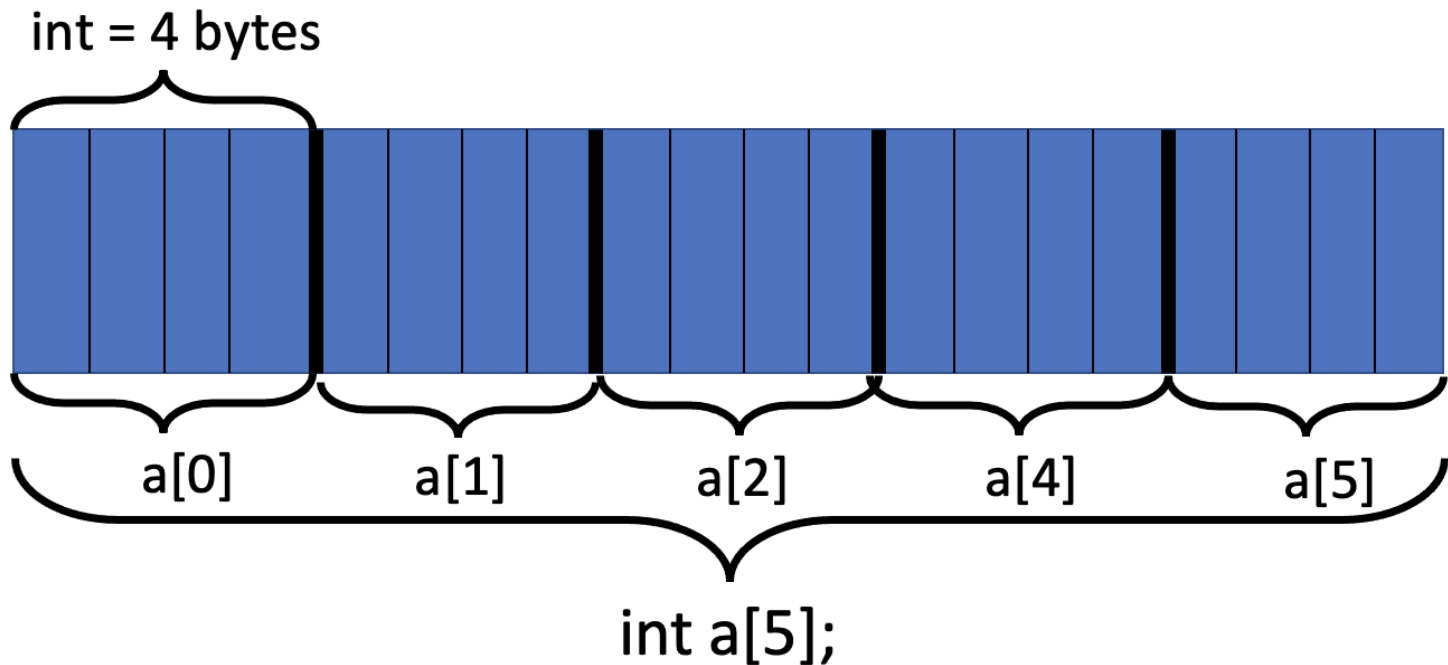
Lecture 3: Array and Loops

Array

- Array is a fundamental **data structure** in C programming language that stores a sequence of elements.
- You can declare an array using the syntax:
 - `data_type variable_name[array_size];` .
 - e.g., `int a[100];` declares an 100 int elements array.
- You can initialize it using the syntax:
 - `data_type variable_name[] = {elements};` .
 - e.g., `int a[] = {1,2,3};` . No need to specify the array size.

Array

- You refer to the first element in the array as `a[0]`, the second element as `a[1]`, and so on.
 - e.g., `a[2] = 3;` assigns 3 to the third element of `a`.
- Array is stored in the memory as a continuous chunk of bytes.



Loops

- When encounter loops, the CPU will continue to execute a block of code, until certain exit conditions are met.
 - If the exit conditions are not set properly, CPU may stuck in a loop and will not exit, wasting computational resources.
 - This situation is called **infinite loop**.

While Loop

The simplest loop is while-loop and its syntax is:

```
while(expression){  
    statements  
}
```

The statements inside of the brackets will only run if `expression` is true. 3D vector addition can be written as

```
int i = 0;  
while(i<3){  
    c[i] = a[i] + b[i];  
    i = i + 1;  
}
```


For Loop

- In previous examples, we all maintained a variable `i`, which increases by one at each iteration.
- The initialization of `i`, exit condition check and increment of `i` are all scattered in the code and are difficult to spot.
- for loop would gather these three pieces all in one place.

```
for(init; exit_condition_check; increment){  
    ...  
}
```

Vector Addition, Revisited 2

```
#include <stdio.h>
void main(){
    double a[] = {1.0, 2.0, 3.0},
           b[] = {2.0, 3.0, 4.0};
    double c[3];
    //addition
    for(int i = 0; i < 3; i=i+1){
        c[i] = a[i] + b[i];
    }
    //display each element in the array c
    for(int i = 0; i < 3; i=i+1){
        printf("%f\n", c[i]);
    }
}
```

- This code does the exactly the same thing as the previous `while` loop, but is arguably more compact.
- Notice `i` is only accessible inside each for loop!

Early Loop Exit

Using `break` statement to exit the loop immediately.

```
// finding the smallest number <= 1000000 that
// can be divided by 32 and 23.
int i = 1;
while(i <= 1000000){
    if(i%32 == 0 && i%23== 0){ // & is logical "AND"
        break;
    }
    i = i + 1;
}
printf("%d\n", i);
//displays "736"
```

Early Loop Restart

Use the `continue` to restart the next loop immediately.

```
// print numbers < 1000 that can be divided by 39.
for(int i = 1; i < 1000; i++){
    if(i % 39 != 0){
        continue;
        // this statement immediately
        // finishes the current iteration
        // and move on to the next iteration.
    }
    printf("%d\n", i);
}
```

You can rewrite this program without using `continue`. How?

Array as Input Argument

You can pass array as input variables of a function:

```
//compute dot product between a and b.  
double dot(double a[], double b[]){  
    double s = 0;  
    for(int i = 0; i < 3; i++){  
        s+ = a[i]*b[i];  
    }  
    return s;  
}
```

If you specify the size of array,

```
double dot(double a[3], double b[3]){  
    ...  
}
```

The size will be ignored by the compiler.

Array as Input Argument

If I do not know the length of the array, what can I do?

- Pass another input argument, specifying the array length.

```
//compute dot product between a and b.  
double dot(double a[], double b[], int len){  
    double s = 0;  
    for(int i = 0; i < len; i++){  
        s += a[i]*b[i];  
    }  
    return s;  
}
```

Pass by Value

When you pass an input argument to a function, you are passing by value: The program will copy the value to the input variable.

```
double square(double a){
    a = a*a;
    return a;
}
void main(){
    double in = 2;
    double out = square(in);
    printf("%f %f\n", out, in);
    //display 4 2
    //square function does not change the value of
    //its input variable.
}
```

The value of `in` is copied to the input argument `a`, thus operations on `a` has no effect on `in`.

Pass by Reference

- However, comparing to ordinary variables, the array occupies a much bigger memory space, thus pass by value can be inefficient.
- In C, array is passed by reference.
 - If callee changes the array, caller's array will also be changed.

Pass by Reference, Example

```
//add all elements in an array by 1
void addone(double a[], int len){
    for(int i = 0; i < len; i++){
        a[i] += 1;
    }
}

void main(){
    double a[] = {1.0, 2.0};
    addone(a, 2);
    printf("%f %f\n", a[0], a[1]);
    //display 2 3, NOT 1, 2!!
}
```

Return an Array

- Array cannot be returned by a function.
- However, since a function can make changes to caller's array, you can pass an array as input argument, and store results in that array.

```
//compute a+b and store the result in c
void add(double a[], double b[], double c[], int len){
    for(int i = 0; i < len; i++){
        c[i] = a[i] + b[i];
    }
}
void main(){
    double a[] = {1.0, 2.0}, b[] = {2.0, 3.0};
    double c[2];
    add(a,b,c,2);
    printf("%f %f\n", c[0], c[1]);
    //display 3 5
}
```

Multidim. Array

Matrix is a rectangle of numbers, arranged in rows and columns.

$$\mathbf{A} = \begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 6 \\ 7, & 8, & 9 \end{bmatrix}.$$

We can use multidimensional array to store a matrix.

```
//an integer 2D array used to store a 3 by 3 matrix.  
//The initialization is row-first.  
int A[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
```

Multidim. Array Example

We can use multidimensional array to store a matrix. The following function `trace` computes the [trace of a matrix](#).

```
#include<stdio.h>
int trace(int nrow, int ncol, int A[nrow][ncol]){
    if (nrow != ncol) {
        printf("not a square matrix!\n");
        return 0;
    }
    int tr = 0;
    for(int i =0; i<nrow; i++){
        tr += A[i][i];
    }
    return tr;
}
void main(){
    int A[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
    printf("%d\n", trace(3, 3, A));
}
```

Multidim. Array Example

- It is important to perform a sanity check at your code!
 - Trace operation is only defined on square matrices!
- The compiler does need to know all dimensions of the array!
 - If you want to use other arguments (`nrow` and `ncol` in the example above) to specify the dimensions, make sure they are declared **before** the array.
 - `int trace(int A[nrow][ncol], int nrow, int ncol)` will not work!
- Multidimensional array is also passed by reference.

Week 4 Lab,

1. Write a function that prints out elements in a matrix A .

```
void print_matrix(int nr, int nc, int A[nr][nc])
```

- Remember to add a space between elements in the same row and add line breaks between rows.
- For example, a matrix

$$A = \begin{bmatrix} 1, & 20, & 3 \\ 4, & 5, & 6 \end{bmatrix}$$

Should be printed out as

```
1.00 20.00 3.00
4.00 5.00 6.00
```

Week 4 Lab,

2. Write a function performs the **matrix multiplication** using multi-dimensional arrays

```
void mult(int nrowA, int ncolA, int nrowB, int ncolB,  
          double A[nrowA][ncolA],  
          double B[nrowB][ncolB],  
          double C[nrowA][ncolB])
```

- Use the skeleton function that is already provided in the lab pack.
- Test your function with dummy inputs, using the function you just wrote in Q1 to print out the outcome and check the correctness of your function.

Week 4 Lab,

3. (submit) Write a function, that "flattens" a 2D array into 1D array. For example, 2D array `{{1,2},{3,4}}` is flattened into `{1,2,3,4}` .

i. `void flatten(int nrow, int ncol, int a[nrow][ncol], int a_f[]);`

ii. After the execution, `a_f` stores the flattened array.

Week 4 Lab

4. Write a function that reads an element from the flattened array as if it is reading from the corresponding 2D array.

- i. `double get_elem(int nc, double a_f[], int i, int j)`, where `nc` is the number of columns of the unflattened 2D array and `i, j` are indices of the unflattened 2D array.
- ii. For example, given a 2D array `a`, and its flattened version `a_f`, `get_elem(nc, a_f, i, j)` should output the value `a[i][j]`.
- iii. Write no more than one line of code in your `get_elem` function.