Week 1 Introduction

Monday, January 19, 2026 10:39 PM

0-
Introduction

CSCI 3110 An Introduction to "Algorithms and Data Structures"

Algorithms are step-by-step procedures or sets of rules for solving a problem or performing a task. In computer science, they provide the precise instructions a computer needs to process input, perform calculations, and produce output. Algorithms can be simple, like finding the maximum number in a list, or highly complex, like routing data across the internet or training a machine learning model.

Some classic simple algorithms:

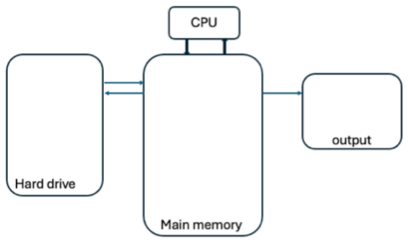
- **Linear Search** – Scan through a list one element at a time until the target is found.
- **Binary Search** – Repeatedly divide a sorted list in half to quickly locate a target.
- **Bubble Sort** – Repeatedly swap adjacent elements if they are in the wrong order.
- **Selection Sort** – Repeatedly find the smallest (or largest) element and place it in its correct position.
- **Greatest Common Divisor (Euclid's Algorithm)** – Repeatedly take remainders to find the GCD of two numbers.
- **Fibonacci Sequence (recursive or iterative)** – Compute terms of the sequence by summing the previous two.

Data Structures: To organize, manage, and store data in a format in a computer that is efficient in accessing data for the application.

When studied alongside **data structures**, algorithms allow us to design solutions that are not only correct but also efficient in terms of time and memory. Together, algorithms and data structures form the foundation of computer science and are essential for understanding how software systems achieve speed, scalability, and reliability.

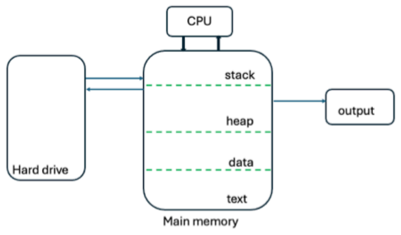
Main memory vs. secondary memory

- Program and data reside in secondary memory, i.e., hard disk drive, when not in use. Program and Data are stored permanently on secondary memory.
- Main memory, aka primary memory, stores data temporarily. Data stored in main memory can be directly accessed by the CPU
- When we run a program, the program and the data will be fetched into the main memory, CPU will execute the commands, perform tasks.

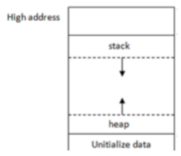


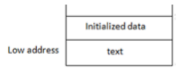
Stack and Heap
Each running program has its own memory layout, separated from other programs. The layout consists of a lot of segments, including:

- stack: stores local variables
- heap: dynamic memory for programmer to allocate
- data: stores global variables, separated into initialized and uninitialized
- text: stores the code being executed



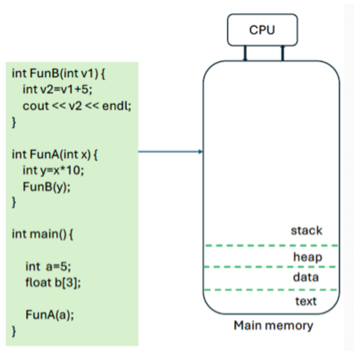
The addresses go from 0 all the way to the largest possible address, depending on the machine. As the figure below, the text, data, and heap segments have low address numbers, while the stack memory has higher addresses.





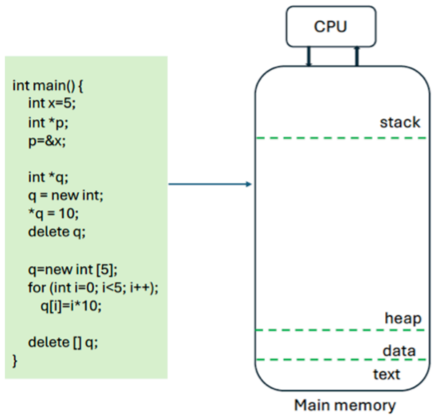
Memory layout of a c++ program

Stack
The stack segment is near the top of memory with high address. Every time a function is called, the machine allocates some stack memory for it. When a new local variable is declared, more stack memory is allocated for that function to store the variable. Such allocations make the stack grow downwards. After the function returns, the stack memory of this function is deallocated, which means all local variables become invalid. **The allocation and deallocation for stack memory is automatically done**. The variables allocated on the stack are called *stack variables*, or *automatic variables*.

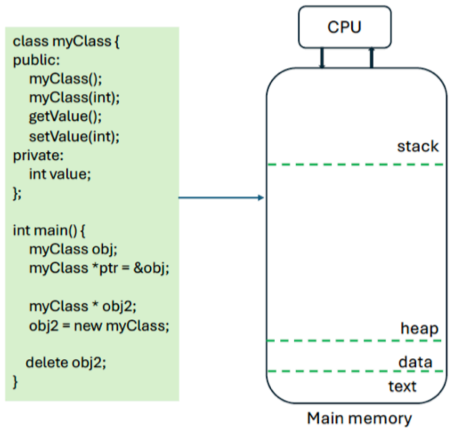


```
int FunB(int v1) {
   int v2=v1+5;
   cout << v2 << endl;
}

int FunA(int x) {
   int y=x*10;
   FunB(y);
}

int main() {

    int  a=5;
    float b[3];

    FunA(a);
}
```

Heap: Unlike stack memory, heap memory is allocated explicitly by programmers, and it won't be deallocated until it is explicitly freed.



```
int main() {
    int x=5;
    int *p;
    p=&x;

    int *q;
    q = new int;
    *q = 10;
    delete q;

    q=new int [5];
    for (int i=0; i<5; i++);
       q[i]=i*10;

    delete [] q;
}
```



```
class myClass {
public:
    myClass();
    myClass(int);
    getValue();
    setValue(int);
private:
    int value;
};

int main() {
    myClass obj;
    myClass *ptr = &obj;

    myClass * obj2;
    obj2 = new myClass;

   delete obj2;
}
```

Case Study - Developing an online dating app

- What data is used?
- What is the task?
- How are the data stored during program execution?
- Are there other ways this data can be stored?
- Are there other tasks?
- Which one is a good data structure that supports the task of the application?
    - How do we objectively evaluate “goodness”?