PIC 10A 2B

TA: Bumsu Kim

Today…

- Structure
- HW Hints
- Review of Materials Covered after the Midterm
  - Arrays and Vectors
  - Classes
  - Pointers

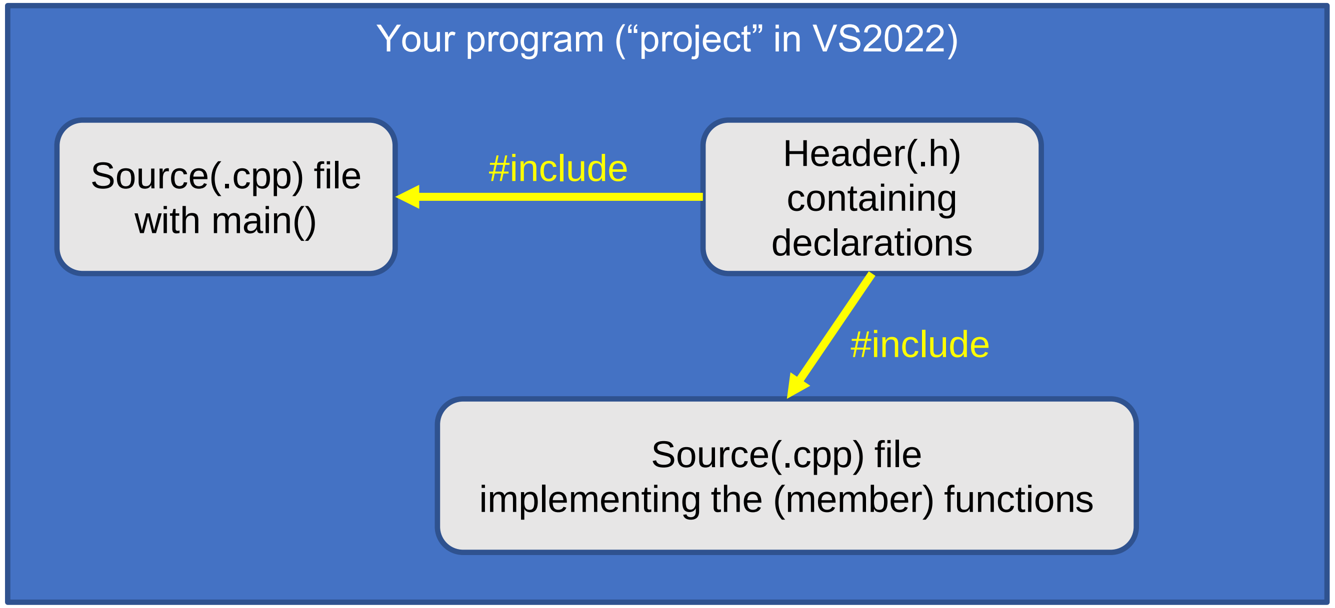
Structure

- A C++ structure is basically same with a class, whose members are public by default, instead of private
- Everything else is exactly the same with a class
- Now you can use "." operator to directly access the member
  - which was prohibited for classes, in terms of coding practice
- Pointer + "." operation = "→" operator (works for classes, too)

```
struct myStruct {
        int i;
        string str;
};
int main() {
        myStruct* sPtr;
        sPtr->i = 1; // (*sPtr).i = 1;
        cout << sPtr->str.length(); // (*sPtr).str.length();
```

HW7 Questions?

- Separate the header(.h) and the source(.cpp) files
  - One and only one "main()" must exist in each program
  - Declare the class and functions in the header
    - Class interface
    - Function signatures
  - Define (implement) the (member) functions in the source file



HW8 Questions?

- HW8: Replace Number
  - 1. store the "old value"
  - 2. change the value that p refers to if necessary
  - 3. return the old value

- HW8: Find Number
  - 1. Use a for loop to check if we find the match
  - 2. Return the pointer (using the pointer arithmetic), or nullptr if not found

Review

Array, Vector, Class, and Pointer

Arrays

Arrays

- Consider a string variable. It consists of a *sequence of characters*
- Likewise, we can also think of a *sequence of “some other data type”*
  - For instance, a sequence of 6 integers

123	456	789	10	11	12

- In fact, the most “basic” object in C++ for a sequence of data is “Array”
  - Unlike a string, it doesn’t have additional features (e.g. member functions) like length(), substr(), etc.
  - In fact, strings are indeed a “class” defined using C++ arrays (to be covered later)

Arrays

- Array declaration/definition

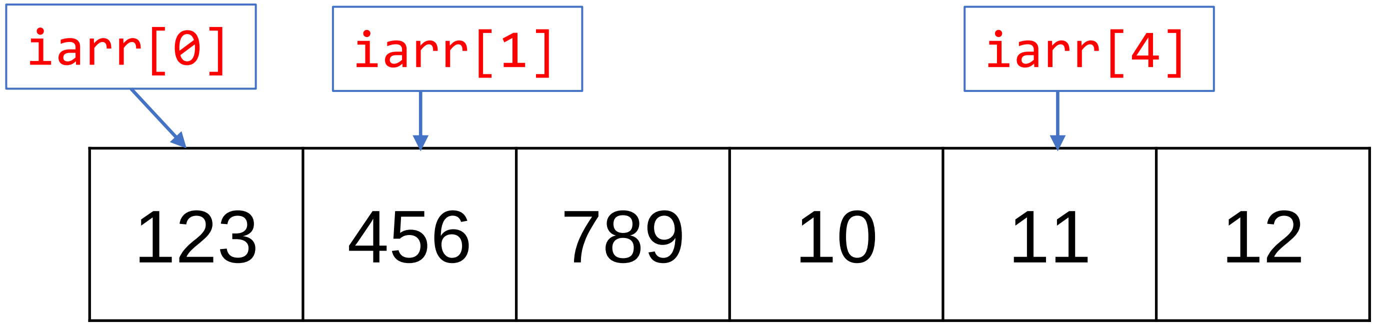
```
int arr[7] =
       { 0, 1, 2, 3, 4, 5, 6 }; // initialization
double darr[10] = { 0.1, 0.2 }; // legal?
char    uninitialized_carr[10]; // legal?
```

- Does not need to be initialized (*legal* in terms of syntax)
- However, ALWAYS initialize your variable (*better* coding practice!)
- Brace initialization
  - #elts in the braces can be smaller or equal to the size of the array
  - (larger → compilation error)
  - If #elts is strictly smaller, it fills the array from the front, and the rest will be *"empty-initialized"*
    - All numeric types becomes 0 (i.e. null character for `char`, false for `bool`, etc.)

UCLA

Arrays

- An int array of size 6 may look like (in the memory):



- “`iarr`” is the name of the array object
- This “`iarr`” actually “points to” the address of the part of the memory that stores the value 123 (or, `iarr`[0])
- Accessing the k-th element of `iarr`: `iarr`[k] (subscript operator)

`sizeof` Operator

- `sizeof` operator measures the size of the type in bytes
  - e.g. What is the `sizeof(char)`?

    A. 1 B. 2 C. 4 D. 8

  - c.f. 1 byte = 8 bits, and can express $2^8 = 256$ different values (i.e. characters)

- Sizes of fundamental types
  - C++ standard does NOT specify the size of integral types in bytes, but it specifies minimum ranges (e.g. [-32767, 32767] for int)
  - In VS 2022, we have:

```
size of int: 4
size of unsigned int: 4
size of short: 2
size of long: 4
size of long long: 8
size of char: 1
size of bool: 1
size of float: 4
size of double: 8
```

Arrays

- Arrays cannot change its size "dynamically"
  - Its size should be set in compile-time, and cannot be changed
- For instance, you can't do:
  - The int variable `sz` is not a compile-time constant

```
int sz = 5;
int iarr[sz] = {};
```

- On the other hand, you can do:
  - `sz` is now a compile-time constant

```
const int sz = 5;
int iarr[sz] = {};
```

- However, it doesn't mean that `const int` is always a compile-time constant
  - You can't do:
  - To make it clear, use the `constexpr` keyword

```
int get_num() {
    int n;
    cin >> n;
    return n;
}
int main() {
    const int sz = get_num();
    int iarr[sz] = {};
```

Array Exercises

- Week7_1_Array_Exercises.cpp on Github
  - `PrintArr`
  - `maxArr`
  - `sumRange`

Vectors

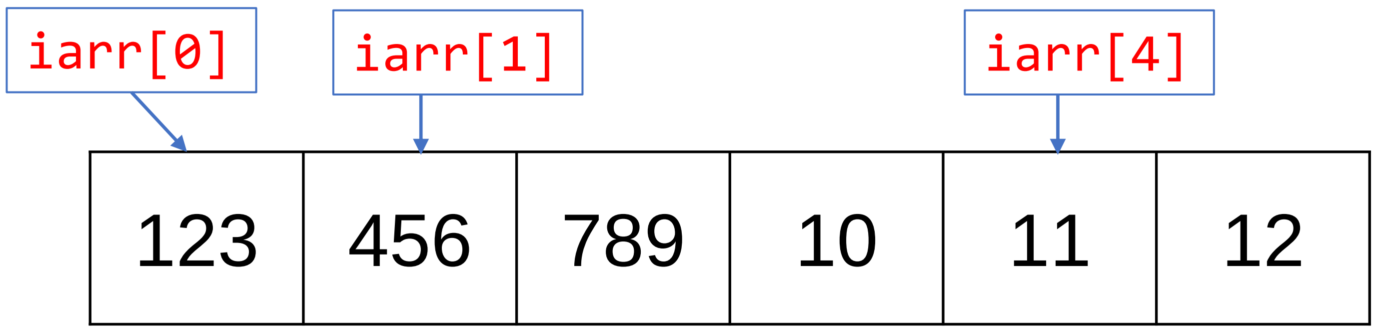
Vectors

- Vector is a special type of a class (“templated class”) that can be considered an **array** of some other class type, with useful member functions
  - Recall that a “string” can be thought as an array of “char” variables, with useful member functions such as length, substr, etc.
- Must include `<vector>` library to use vectors
- Ex) Vector of `int`s
  - `vector<int> vint = { 2022, 11, 10 }; // contains 2022, 11, 10 in this order`
  - Access: subscript operator []
    - `cout << vint[0] << ", " << vint[1] << ", " << vint[2] << endl;`
- The vector class has many useful constructors
  - Creating a size “N” vector: `vector<int> vint2(N);`
  - Creating a size “N” vector and initialize all with “val”: `vector<int> vint3(N, val);`
  - Creating a copy of other vector(copy constructor): `vector<int> vint4(vint);`

UCLA

Vectors and Arrays

- Vectors are like arrays, but with *special features* in addition
- As mentioned last time, (static) arrays are the *most basic* “array-like” objects
- Recall that an int array of size 6 may look like (in the memory):



- Accessing the k-th element of iarr → iarr[k] (subscript operator)
- A vector (internally) has an array to store the data, and support the same subscript operator
- A vector not only stores the data, but also has useful member functions for it
  - push_back, insert, erase, etc.

Vectors and Arrays

- But (static) arrays cannot change in its size, and there are no member functions for array objects
  - It’s why I called it the *most basic* “array-like” type
  - No push_back, insert, erase, etc.

- There is another type of arrays, “dynamic arrays” whose size can be changed dynamically
  - But not covered in this course, you will see them again in PIC 10B
  - Also this object is pretty difficult to deal with

- Vectors are much easier to handle, so in most cases you can just use vectors
  - Vectors = Arrays + Useful Features

Vector Member Functions

- Some useful member functions of std::vector<T>
  - size() Returns the size (#elements) of the vector
  - front(), back() Returns the element in the front and back, respectively
  - push_back(val) Adds val at the end (and thus increases the size by 1)
  - insert(pos, val) Inserts val at the position pos (it also ++size)
  - pop_back() Removes the element at the end (--size)
  - erase(pos) Removes the element at the position pos

- Here pos must be “iterators” (covered later in this course, or PIC 10B)

Vector Algorithms

- Implement the following functions for vectors
  - copy_vec — Gets a vector `from`, and copies it to another vector `to`.
  - find_vec — Gets a vector `v` and an input `p`, and finds the first position of `p` in `v` (if doesn't exists, return -1)
  - remove — Gets a vector `v` and a position `pos`, and removes the data at `pos`
  - insert — Gets a vector `v`, position `pos`, and an input `p`, and inserts `p` at `pos`

UCLA

2-D Vectors

- A vector of vectors is called a 2-D vectors, because it looks like a 2-D array if you visualize it

v is a vector of vectors, where

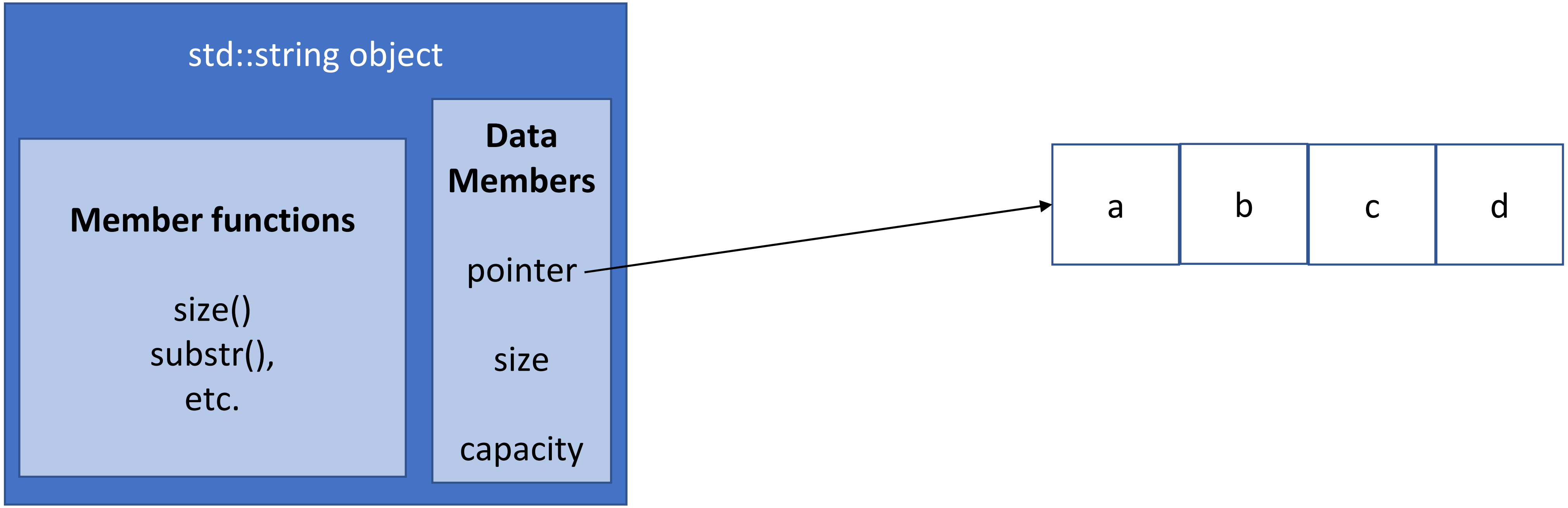
- v[0] is a vector of length 3
- v[1] has length 5
- v[2] has length 4
- v[3] has length 2
- …

C++ Classes

Concepts, Examples, and Exercise Problems

Classes

- Roughly speaking, a class is a user-defined type that contains a collection of data members with other features (methods)
- e.g. “std::string” has data members like
  - the pointer to the char array (actual string data)
  - size and capacity variables, and some other members
  - useful member functions like size(), substr(), and operators([], ==, <, +, etc.)



Classes

- Default accessibility of data members/functions is *private* for classes
  - On the other hand, a very similar data structure, “struct” has *public* members by default

- To access the members, use the “.”(dot) operator

e.g.
```
string str = "ABCDE";
int len = str.length();
```

- Initialization is done by a “constructor”

```
class B {
public:
    void b() const;
    B();
};
```

```
int main() {
    B b_object;
}
```

Calling the default constructor here

Classes

- If a class has several data members, a proper initialization may be important

```
class B {
public:
    string name;
    double salary;
    int age;
    void b() const;
    B();
    //(another c'tor)
};
```

```
int main() {
    B b_object;
    B John("John Doe", 60000, 25);
}
```

Calling the default constructor here.
Setting the name = “”, salary = 0.0 and age = 0 in the default constructor can be a good initialization

Calling another type of constructor here
(not declared in the class interface yet)

What’s the correct signature of this constructor?

```
B(string _name, double _salary, int _age);
```

Classes

- Always use the ***constructor initializer list*** for initialization

```
class B {
public:
    string name;
    double salary;
    int age;
    void b() const;
    B();
};
```

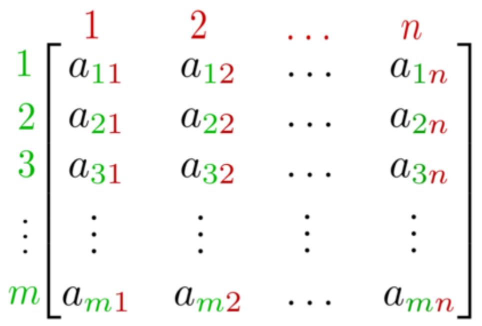
```
B(string _name, double _salary, int _age)
      : name(_name), salary(_salary), age(_age) {}
```

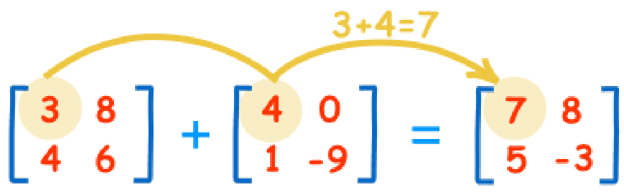
```
int main() {
    B b_object;
    B John("John Doe", 60000, 25);
}
```

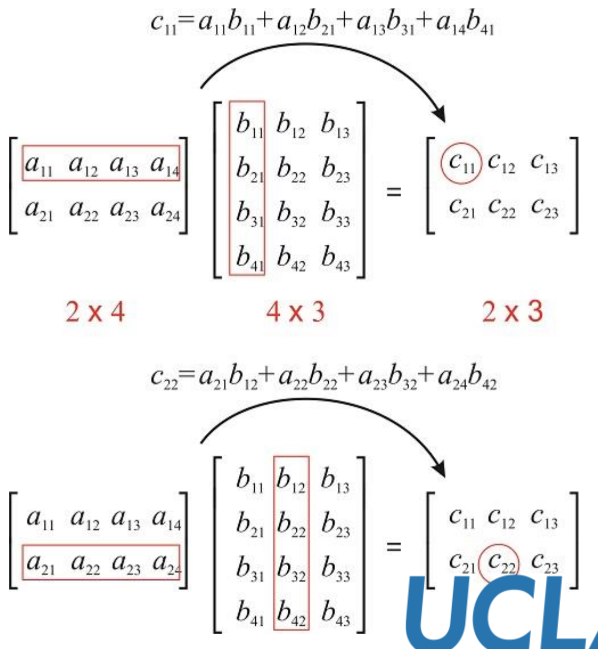
The constructor's **body** is empty in this case

“Matrix” Class

- We implement a class “Matrix” to handle matrices (mathematical objects)
- Each matrix has the following member variables
  - number of rows
  - number of columns
  - 2-D vector for storing the entries
- And the following member functions
  - A.size(): returns the dimension as a vector of length two (nRows, nCols)
  - A.at(i, j): returns the reference of the element at (i-th row and j-th column)
  - A.isEqualDim(B): check if the dimensions of A and B are the same
  - A.add(B): adds A and B and return the sum A+B
  - A.subtract(B): returns A-B
  - A.scalarMultiplication(c): returns cA, where c is a scalar and A is a matrix
  - A.multiplication(B): returns AB (matrix multiplication)
  - A.transpose(): transposes the matrix (does not return anything)
  - A.print(): prints the matrix on the console
- Constructors accepting the dimension (default: 1x1, filled with 0)







Classes (Review)

- Access Specifiers
  - Private – Members (data and methods) cannot be accessed **outside** the class
    - Inside the class interface, everything is accessible!
  - Public – Members can be accessed everywhere (e.g. in the **main** function)

- Constructor
  - A special member function used to initialize objects of its class type
  - Called only once, when an object of a certain class is created (defined)

- Accessing Members
  - Outside the class, use the dot "." operator to access the **public** data members/methods

- Accessors and Mutators
  - Any method that can modify the class object is called a mutator
  - An accessor provides the access to the protected (private) members
    - should be marked as `const`!

Classes – Implicit and Explicit Parameters

- Consider a function "`length()`" of the `string` class

```
string str = "PIC 10A";
str.length();
```

- It doesn't get any *explicit* parameters, but it has an *implicit* parameter
  - `str1.length()`, `str2.length()`, `str3.length()` can return different values
  - The class object upon which the function called is called the *implicit* parameter

- Recall the "add" function in the **Matrix** class
  - at(i, j) is a call upon the implicit parameter
  - e.g. A.add(B) → it's A.at(i,j)

```
Matrix Matrix::add(const Matrix& B) const {
        if (isEqualDim(B)) {
                Matrix C(num_rows, num_cols);
                for (int i = 0; i < num_rows; ++i) {
                        for (int j = 0; j < num_cols; ++j) {
                                C.at(i, j) = at(i, j) + B.at(i, j);
                        }
                }
        return C;
        }
        else {
                cout << "Error: Dimensions must match!" << endl;
                return Matrix{};
        }
}
```

Classes – Constructor Initializer List

- Always use the ***constructor initializer list*** for initialization

```
class B {
public:
    string name;
    double salary;
    int age;
    void b() const;
    B();
};
```

```
B(string _name, double _salary, int _age)
     : name(_name), salary(_salary), age(_age) {}
```

```
int main() {
    B b_object;
    B John("John Doe", 60000, 25);
}
```

The constructor's **body** is empty in this case

Classes – Constructor Initializer List

- Default values can be set using the initializer list

```
class B {
public:
    string name;
    double salary;
    int age;
    void b() const;
    B();
};
```

```
B() : name("default_name"), salary(0.0), age(21) {}
```

```
int main() {
    B b_object;
    B John("John Doe", 60000, 25);
}
```

The constructor's **body** is empty in this case

Classes – Constructor Initializer List

- Or, you can further do this:

```
class B {
public:
    string name;
    double salary;
    int age;
    void b() const;
    B(string _name = "default_name", double salary = 0.0, int age = 21);
};
```

with

```
B(string _name, double _salary, int _age)
      : name(_name), salary(_salary), age(_age) {}
```

This can replace the default constructor and works for both

```
int main() {
    B b_object;
    B John("John Doe", 60000, 25);
}
```

Function Overloading

- Writing functions with the same name, but with different parameters is called “function overloading”
- Here the length function is overloaded on `string`, `vector<int>`, and `vector<double>`

```
// Function Overloading
int length(const string& str) {
	return str.length();
}

int length(const vector<int>& v) {
	return v.size();
}

int length(const vector<double>& v) {
	return v.size();
}
// *******************************************
int main() {
	vector<int> vint{ 1, 2, 3, 10, 20, 30 };
	vector<double> vdouble{ -1.0, 0.0, 1.0 };
	string str("PIC 10A");

	cout << "length of vint = " << length(vint) << endl;
	cout << "length of vdouble = " << length(vdouble) << endl;
	cout << "length of str = " << length(str) << endl;
	return 0;
}
```

Function Overloading

- Operators are also functions, the only difference is that they *can* be used with special forms (i.e. with symbols) instead of “function_name(parameters)”
- e.g.

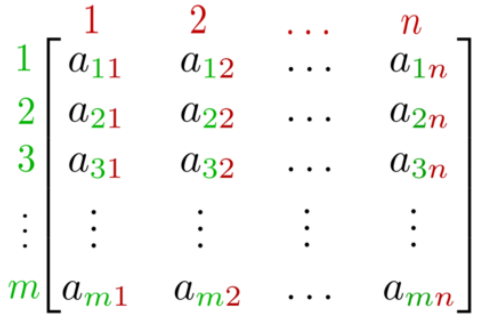
```
string a = "a", b = "bb";
cout << "a+b = " << a + b << ", or " << operator+(a, b) << endl;
```

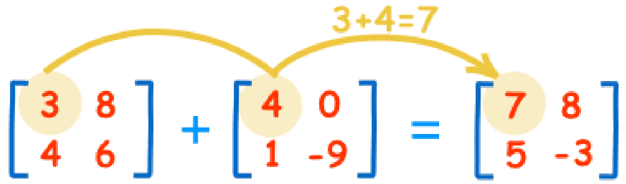
- Thus we can also overload the operators on the user-defined classes
  - Operators +, – (binary), and – (unary)
  - Operator<<
  - See the **Matrix** class example!

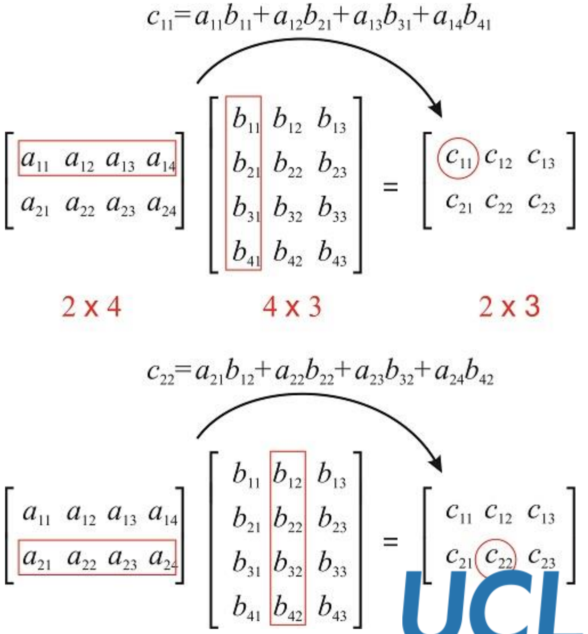
“Matrix” Class with Operator Overloading

- We overload the operators + and – (binary), and operator – (unary) and operator<< for the output
- Compare with the previous version:

```
// Printing
cout << "A: " << A << endl;
// A.print();

// Arithmetic Operations (and printing)
cout << "A + C: " << A+C << endl;
// (A.add(C)).print();

cout << "A * B: " << A*B << endl;
// (A.multiply(B)).print();

cout << "1/4 * A * B: " << A*0.25*B << endl;
// (A.scalarMultiplication(0.25).multiply(B)).print();
```







Pointers

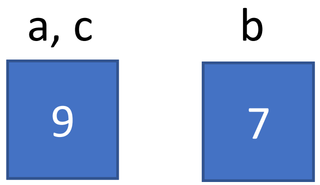
Pointers

- Recall that a reference to a variable is just “another name” of the other

```
class T {
        // ...
};
T tvar;                  // variable of type T
T& ref_tvar = tvar;             // a reference to a type T variable
```

- And they both refer to the same “address” in the memory

```
int a = 5;
int b = a;
int& c = a;
cout << a << ' ' << b << ' ' << c << endl;
b = 7;
cout << a << ' ' << b << ' ' << c << endl;
c = 9;
cout << a << ' ' << b << ' ' << c << endl;
```

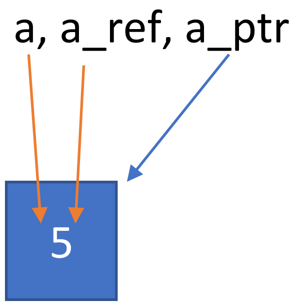


Pointers

- So, a reference, in fact, refers to the value in some address
- Pointers are used to “point” to those address

```cpp
int a = 5;
int& a_ref = a;
int* a_ptr = &a; // address of a
cout << a << ' ' << a_ref << ' ' << *a_ptr << endl;
a_ref = 7;
cout << a << ' ' << a_ref << ' ' << *a_ptr << endl;
*a_ptr = 9;
cout << a << ' ' << a_ref << ' ' << *a_ptr << endl;
```

```
5 5 5
7 7 7
9 9 9
```



- Note that, they store the “address” so you can’t do something like

```cpp
a_ptr = a;
```

a value of type "int" cannot be assigned to an entity of type "int *"

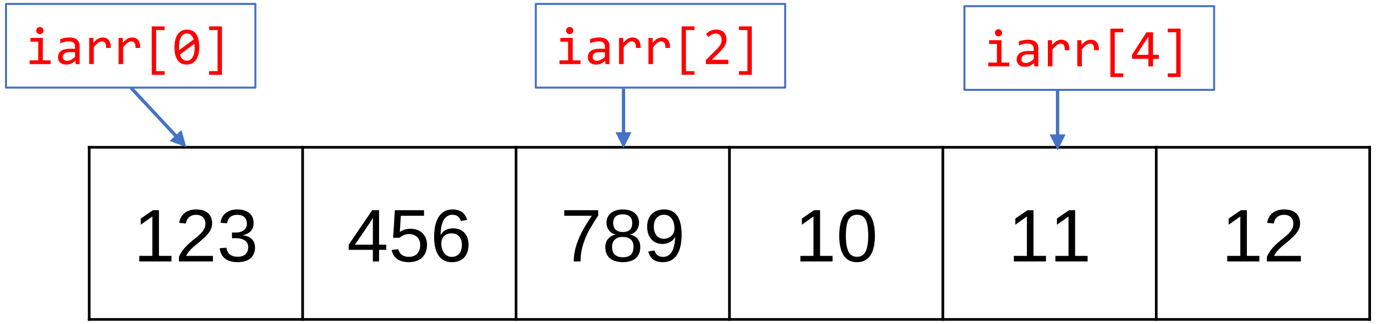
Search Online

Pointer Arithmetic

- To access the value stored in the address where your pointer points, use the ***dereference operator*** (*)

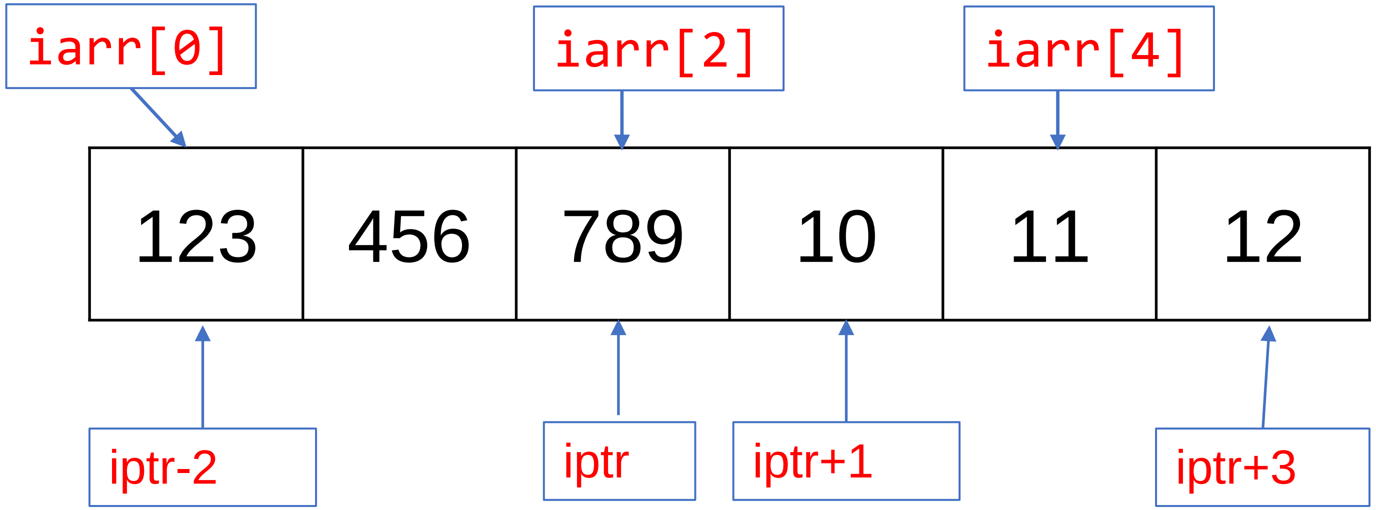
```
int a = 5;
int* a_ptr = &a; // address of a
*a_ptr = 9;           // dereference to access the value
cout << a << ' ' << *a_ptr << endl; // again, dereference to access the value
```

- You can also “move around” the address using “pointer arithmetic”
- Let’s say you have an `int` array of size 6:



Pointer Arithmetic

- An `int` array of size 6 may look like (in the memory):




```
int* iptr = &(iarr[2]);
// &(2nd elt of iarr)
```

- Let iptr be a pointer pointing to the 2nd element of the array, then you can move around by adding/subtracting numbers from its address
- This is called "pointer arithmetic" (just like "integer arithmetic" but no *,/,%, etc.)
  - +, -, and ++, -- are allowed
  - Additionally, comparisons (==, <, >) are also available

Arrays vs Pointers

```
size of a: 20
size of aptr: 8
```

- Arrays “decay” to pointers if it needs to be
- But still, they are DIFFERENT!

```
int a[] = { 1,2,3,4,5 };
int* aptr = a;
cout << "size of a: " << sizeof(a) << endl
<< "size of aptr: " << sizeof(aptr) << endl;
```

- You can compute the length of the array using “`sizeof(a)/sizeof(int)`”
- But then why do you still need to pass the array size to a function?
  - e.g. consider the following exercise from the textbook:

(This is essentially Exercise E6.8 of the book)
Write a program equals.cpp that implements the function

```
bool equals(int a[], int a_size, int b[], int b_size)
```

Arrays vs Pointers

- It is because an array ***decays*** to a pointer when it is passed to a function
- You can never pass an array (by value), nor return an array
  - Passing arrays is not an error, but it decays to a pointer
  - Returning an array is an error!

- *Remark*: In fact, arrays can be passed by reference

```
int f(int(&a)[3]) {
    cout << "*size of this array: " << sizeof(a) << endl;
    return 0;
}
```

  - This function can get an array of size 3, and the output is `*size of this array: 12`

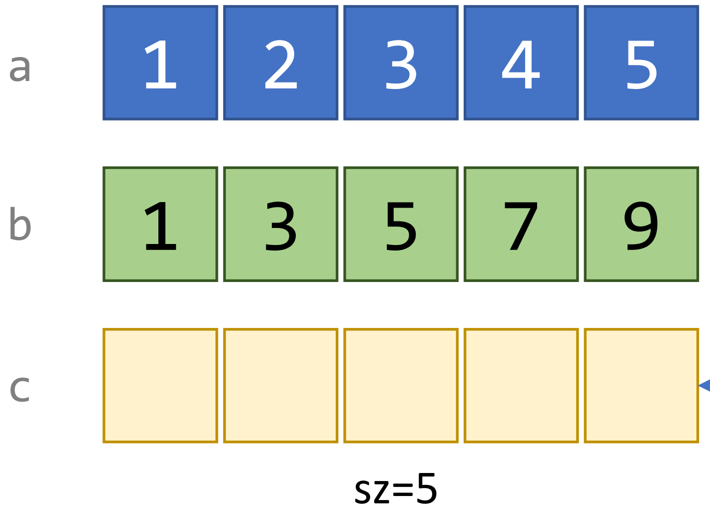
Arrays vs Pointers

- Since the implicit conversion [array → pointer] is allowed, you can use arrays in every expression where pointers are expected

```
void add_arrays(int a[], int b[], size_t sz, int *c) {
    for (size_t i = 0; i < sz; ++i) {
        *(c++) = *(a++) + *(b++);
    }
}
```

- How does the function work?

Arrays vs Pointers




```
void add_arrays(int a[], int b[], size_t sz, int *c) {
    for (size_t i = 0; i < sz; ++i) {
        *(c++) = *(a++) + *(b++);
    }
}
```

At least "sz" places must be reserved!
(otherwise it may corrupt your data and result in undefined behavior)

- Pointers move forward by ++
- The post-increment operator returns the original value
- Access the value stored in an address via the dereference operator *