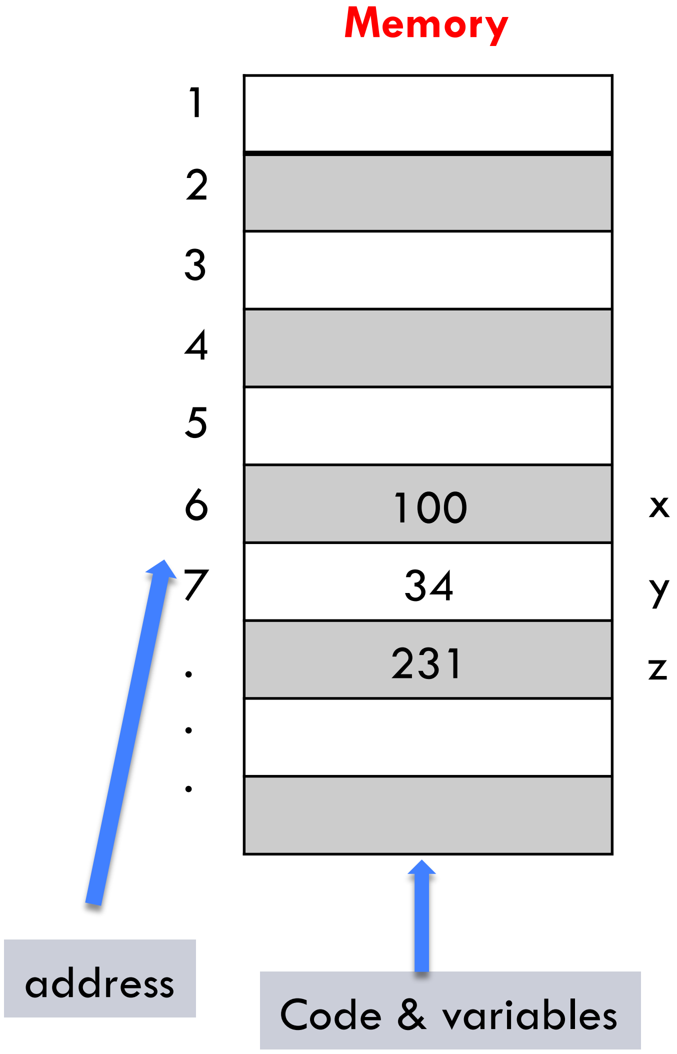
CSC230

Intro to C++ Lecture 3

What is Pointers?

Pointer is a special variable that stores the address of another variable

Pointers



x is an int variable.
Q: Where is x in memory ?

A: &x

Q: How to save x's memory address?

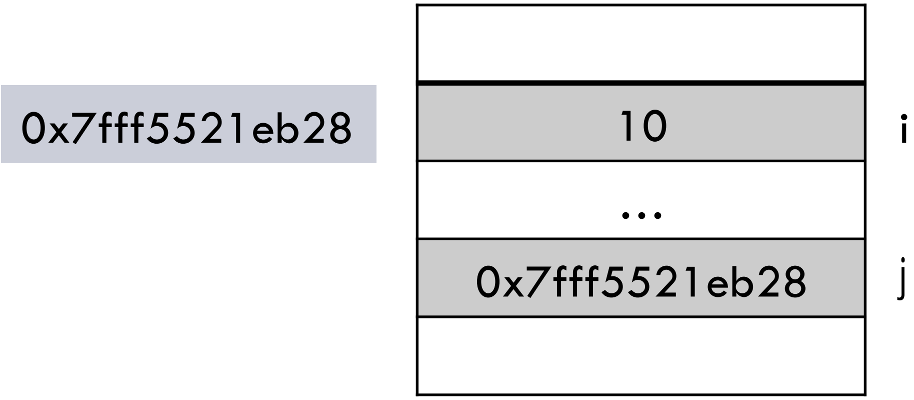
A: int *addr;
addr = &x;

A **pointer** is a variable that contains the address of another variable.

- addr is a pointer to int.
- &x is NOT a pointer, it is an address

Pointer example

```
#include <iostream>
using namespace std;

int main ()
{

  int i = 10;
  int *j = &i;
  cout << i << "\t" << &i << "\t" << j << "\t" << *j << endl;

  return 0;
}
```



```
$ ./a.out
10      0x7fff5521eb28  0x7fff5521eb28  10
```

The unary operator * is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to.

More pointer examples

```
int x = 1, y = 2, z[10];
int *ip;       // what is ip?
// ip is a pointer to int

  ip = x // correct ?

ip = &x;         // ip now points to x
y = *ip;         // what is the value of y?
                  // y = 1

*ip = 0;         // what is the value of ip?
                  // address of x
                 // what is the value of x?
                  // 0

ip = &z[0];      // what is the value of ip?
                 // what is the value of x?
```

Why pointer?

Want to use a function to swap two values.

```
void swap(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y =temp;
}
```

```
void swap(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y =temp;
}
```

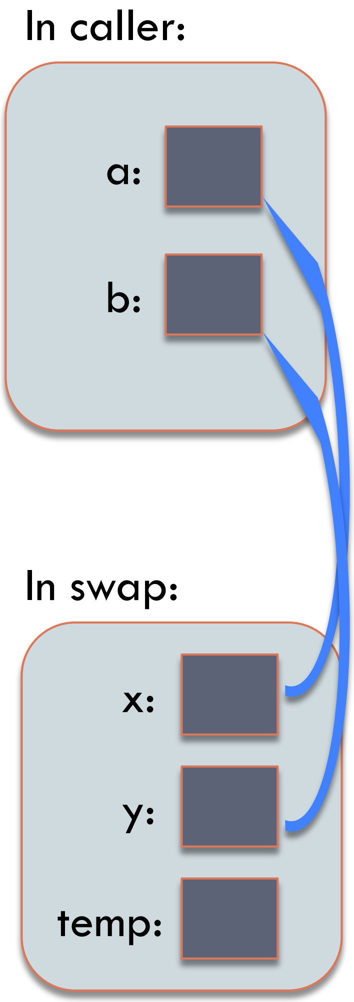
```
swap(a, b);
```

✗

```
swap(&a, &b);
```

✔

How it works inside?



In caller:

a:

b:

In swap:

x:

y:

temp:

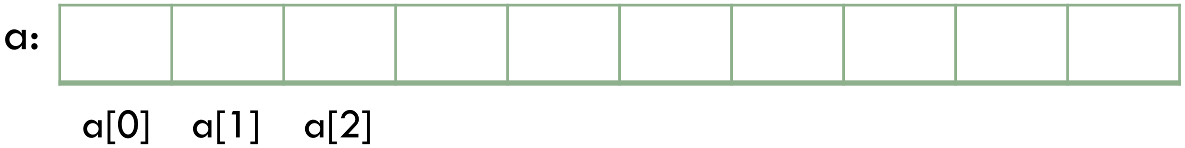
```
void swap(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y =temp;
}
```

```
swap(&a, &b);
```

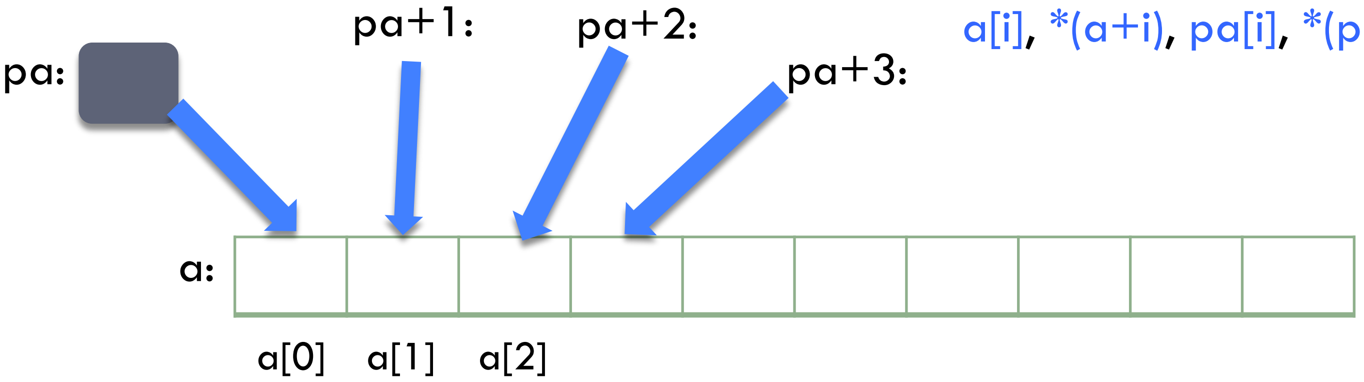
Pointers and Arrays

Pointer has a strong relationship with array.

```
int a[10];
```



```
int *pa;
pa = &a[0];
```



In fact, a[0] == a, variable a has the starting address of the whole array.

pa = a; and pa = a[0]; are equivalent.

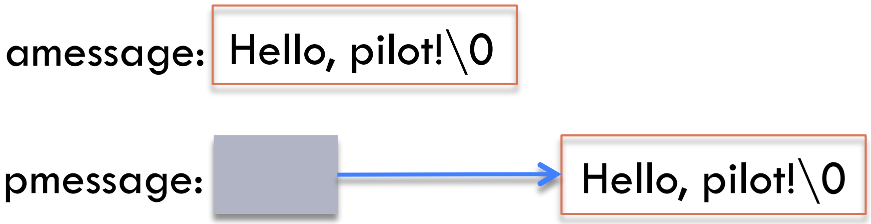
a[i], *(a+i), pa[i], *(pa+i) are equivalent.

Pointers and Arrays Example

```c
/* strlen: return length of string s */
int strlen(char *s)
{
  int n;
  for(n=0; *s != '\0'; s++)
    n++;
  return n;
}
```

Characters Pointers

```
char amessage[]= "Hello, pilot!";
char *pmessage = "Hello, pilot!”;
```

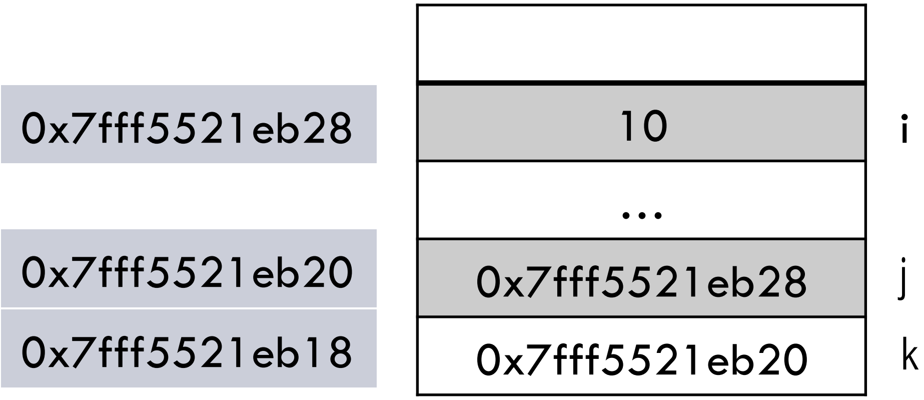


amessage: Hello, pilot!\0

pmessage: Hello, pilot!\0

Pointer to pointer

```
#include <iostream>
using namespace std;

int main ()
{

   int i = 10;
   int *j = &i;
   int **k = &j;
   cout << &i << "\t" <<k << "\t" << **k << endl;

   return 0;
}
```



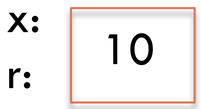
0x7fff5521eb28	10	i
	...	
0x7fff5521eb20	0x7fff5521eb28	j
0x7fff5521eb18	0x7fff5521eb20	k

C++ Reference

IMPORTANT

A **reference** variable is an alias, another name of an existing variable.

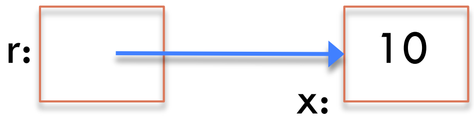
```
  int x = 10;
  int &r = x;
```



x == r
&x == &r

vs. **Pointer**

```
  int x = 10;
  int *r = &x;
```



x != r
&x != &r
*r == x

Reference vs. Pointer

```
void swap(int &x, int &y)
{
   int temp;
   temp = x;
   x = y;
   y =temp;
}
```

```
Swap(a, b);
```

✔

Looks simpler?

```
void swap(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y =temp;
}
```

```
swap(&a, &b);
```

✔

Reference (alias) vs. Pointer (memory address)

- Pointer can be reassigned, reference cannot.
- Pointer can point NULL (nowhere), reference cannot.
- Pointer has “arithematics” operators, reference does not have.

When should I use reference? When pointer ?

- Use references as function parameters and return types in interfaces
- Use pointers to implement algorithms and data structures

Example: Reference / Pointer

cin & cout Example

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string greeting = "Hello, ";
  string name;
  cin >> name;
  cout << greeting << name << endl;
  return 0;
}
```

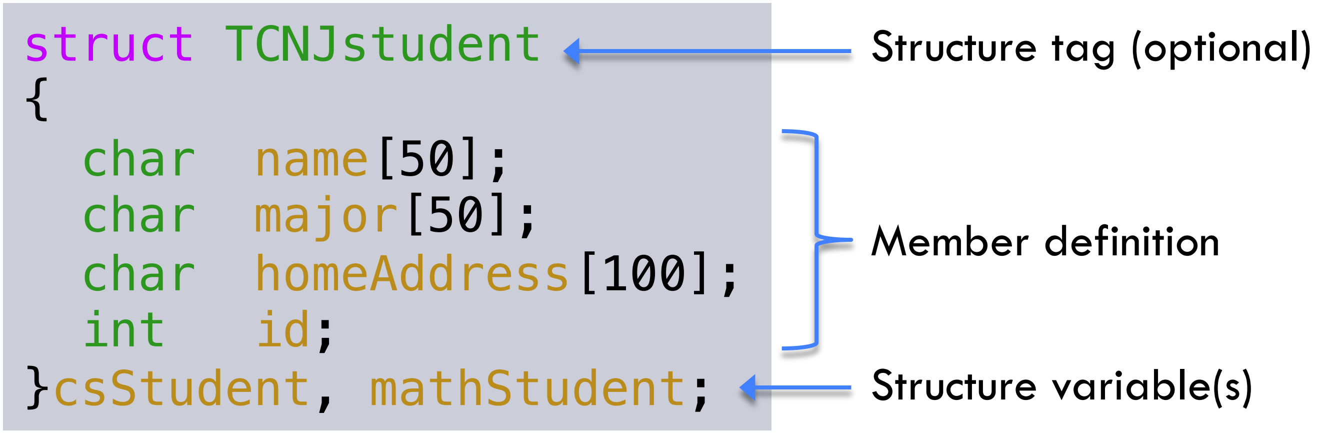
Structure

array:

- User defines
- Combine multiple data items of same type

structure:

- User defines
- Combine multiple data items of different types



```
struct TCNJstudent
{
   char  name[50];
   char  major[50];
   char  homeAddress[100];
   int   id;
}csStudent, mathStudent;
```

Structure tag (optional)

Member definition

Structure variable(s)

Access members of a structure

```
include <iostream>
#include <string>
using namespace std;

struct TCNJstudent
{
   char  name[50];
   char  major[50];
   char  homeAddress[100];
   int   id;
};

int main ()
{
   struct TCNJstudent csStudent, mathStudent;
   csStudent.id = 1000;
   mathStudent.id = 2000;
   strcpy(csStudent.name, "Mike Lee");
   strcpy(csStudent.major, "CS");
   strcpy(csStudent.homeAddress, "Earth");
   cout << csStudent.name << " " << csStudent.homeAddress <<endl;
   return 0;
}
```

Structure variables

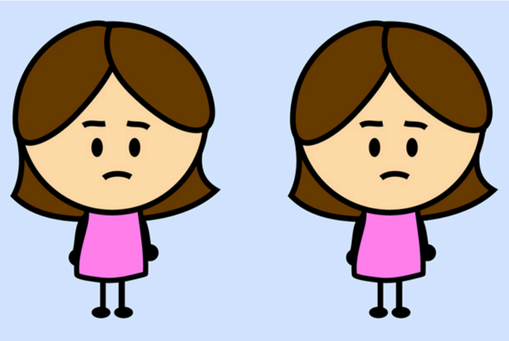
Member access

Structure as a function parameter

```
struct TCNJstudent
{
  char  name[50];
  char  major[50];
  char  homeAddress[100];
  int   id;
};

void infoCheck(struct TCNJstudent student)
{
  cout << student.name << endl;
}
```

```
infoCheck(csStudent);
```

Pointers to structures



Can’t you see the difference?

```
struct TCNJstudent
{
  char  name[50];
  char  major[50];
  char  homeAddress[100];
  int   id;
};

void infoCheck(struct TCNJstudent *student)
{
  cout << student->name << endl;
}
```

```
infoCheck(&csStudent);
```

Class and Object

```
class Base {
  public:
  // public members go here
  protected:
  // protected members go here
  private:
  // private members go here
};
```

- Access specifiers: public, private, protected
- Each class may have multiple sections
- Each section remains effective until either another section or the end of the class body
- The default access is private

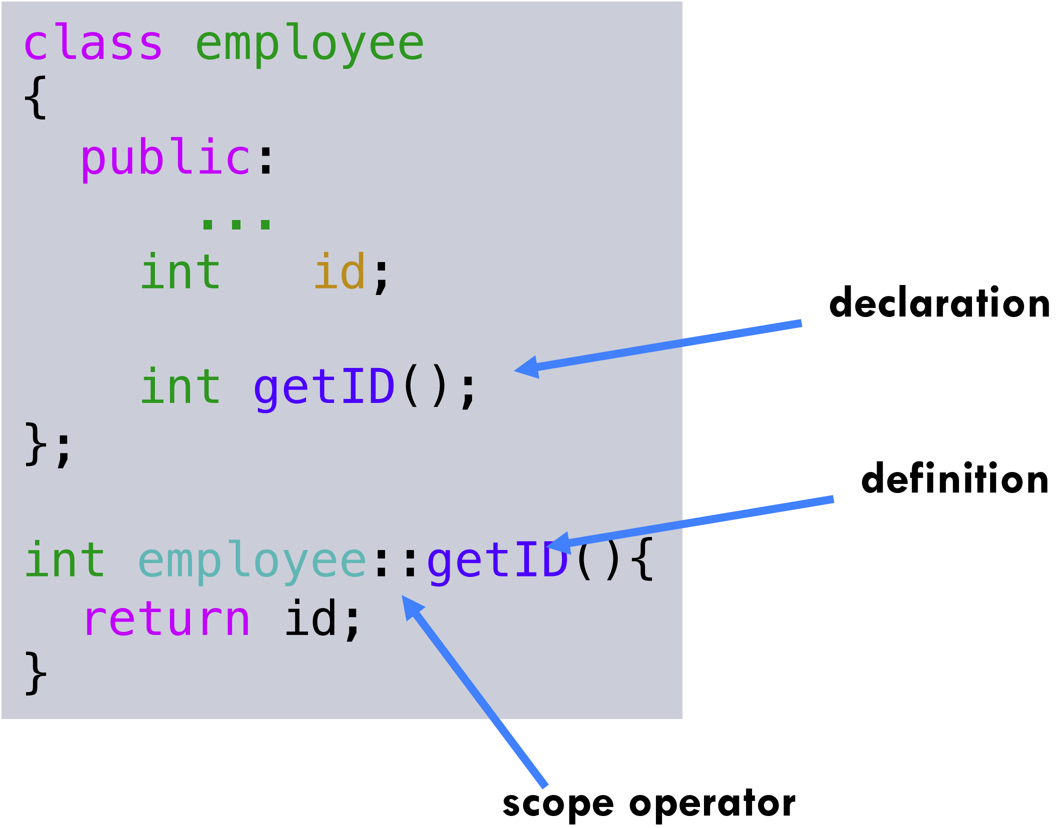
Class and object example

```
#include <iostream>
#include <string>
using namespace std;

class student
{
  public:
    char  name[50];
    char  major[50];
    char  homeAddress[100];
};

int main ()
{
  student csStudent, mathStudent;
  strcpy(csStudent.name, "Mike Lee");
  strcpy(csStudent.major, "CS");
  strcpy(csStudent.homeAddress, "Earth");
  cout << csStudent.name << " " << csStudent.homeAddress <<endl;
  return 0;
}
```

Method definition

```
class employee
{
  public:
      ...
    int   id;

    int getID(){
      return id;
    }
};
```

✔



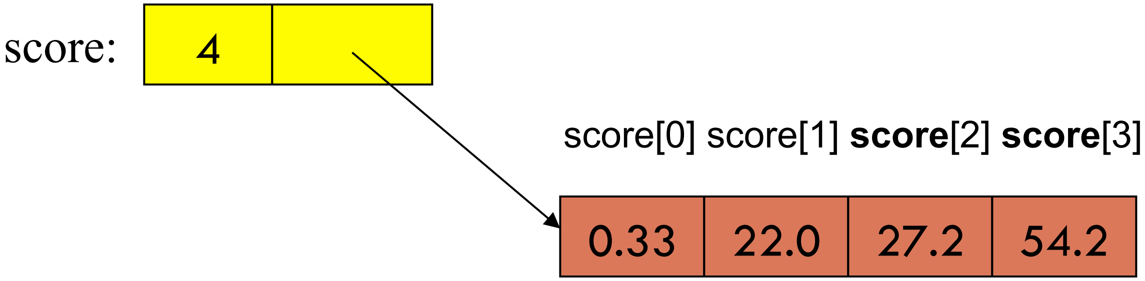
```
class employee
{
  public:
      ...
    int   id;

    int getID();
};

int employee::getID(){
  return id;
}
```

✔

Vector

- A **vector**
  - Can hold an **arbitrary** number of elements
    - Up to whatever physical memory and the operating system can handle
  - That number can **vary** over time
    - E.g. by using **push_back()**
  - Example

```
vector<double> score(4);
score[0]=.33;    score[1]=22.0;    score[2]=27.2;    score[3]=54.2;
```



Array vs. vector

Array	Vector
Provides contiguous, indexable sequence of elements	Provides contiguous, indexable sequence of elements
Once created, the size cannot be changed	Size change be changed, grow or shrink dynamically
If dynamically allocated, user got a pointer, the user can use *sizeof(arr)/ sizeof(*arr)* to figure out the array size. But it is error-prone.	When a vector is created, one object is created. A vector object is not a pointer, but *&vec[0]*returns the starting address of the data
If the array is dynamically allocated, user need to de-allocate it.	Vector automatically manages memory, including allocation and de-allocation.
Usually when passed to a function, it is passed as a pointer with separate parameters for its size. Cannot be returned from a function.	Can be passed to/returned from function
Can’t be copied/assigned directly	Can be copied/assigned directly

Revisit a 2D array parameter example

address of element (r, c) = base address of array
+ r*(number of elements in a row)*(size of an element)
+ c*(size of an element)

What if we do not know the col value?

- col value must be determined when we define the function
- row value can be passed when we call the function

```
void init(int twoD[][col], const int row) {
  for (int i = 0; i < row; i++) {
    for (int j = 0; j < col; j++)
      twoD[i][j] = -1;
  }
}
```

Pass a 2D vector as parameter

```
void init(vector< vector<char> > &twoD) {
  for (int i = 0; i < twoD.size(); i++) {
    for (int j = 0; j < twoD[0].size(); j++)
      twoD[i][j] = -1;
  }
}
```

twoD is a reference to 2D vector of char

twoD element is accessed like 2D array

```
vector< vector<char> > searchMatrix;
searchMatrix.resize(x);
for (int i=0; i<x; i++) {
    searchMatrix[i].resize(y);
}
….
init(searchMatrix);
```

Declare 2D vector

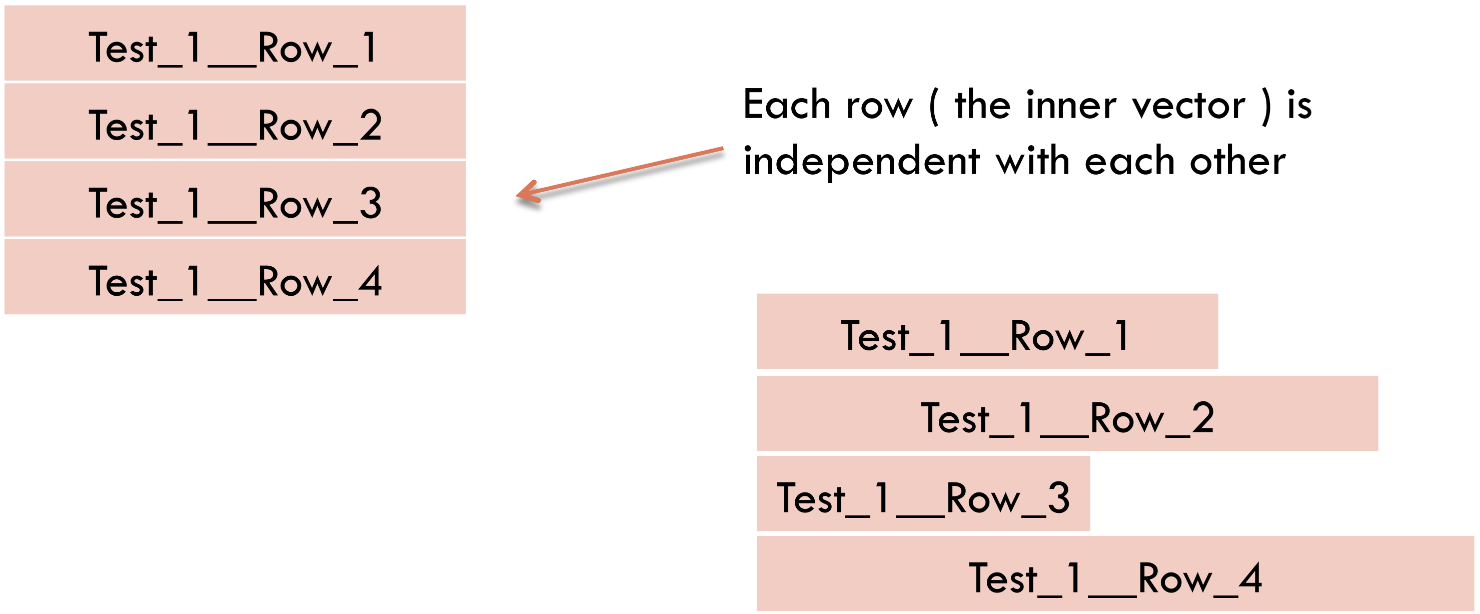
First dimension size is x

The element of the first dimension is a vector with size y

Lab 2 discussion: Vector

Vector of a vector

- 2D Vector Declaration : vector < vector < char> > Test_1



Each row (the inner vector) is independent with each other

Lab 2 discussion: Vector

Vector of a vector

- 2D Vector initialization:

```
int m, n;

cin>>m>>n;

vector<vector<int> > v;

for(int i=0; i<m; i++)
{
    for(int j=0; j<n; j++)
    {
        int a;
        cin>>a;
        v[i].push_back(a);
    }
}
```

Access element through index need to be initialize first

```
for (int i=0; i<m; i++)
{
    v.push_back(vector<int>());
    for (int j=0; j<n; j++)
    {
        int a;
        cin >> a;
        v[i].push_back(a);
    }
}
```

```
vector<vector<int> > v(m);

for(int i=0; i<m; i++)
{
    for(int j=0; j<n; j++)
    {
        int a;
        cin>>a;

        v[i].push_back(a);
    }
}
```

Initialize v with m rows

Lab 2 discussion: Vector

Vector of a vector

- index vs push_back

index : the element has to be there (initialize it before you use it)
push_back : append a value at the end of the vector

- size of the 2d vector

    vector<vector<char> > Test_1;

- what is the Test_1.size() ?
- what is the Test_1[0].size()?
- Example -- test_vector.cpp

Lab 2 discussion: Vector

Traverse the 2D vector

- for loop in 2D vector

```
for(int i=0; i<ROW; i++)
   {
       rowvector.clear();

       for(int j=0; j<COL; j++) {

            cin >> current;

rowvector.push_back(current);

                        }

array2.push_back(rowvector);
   }
```

What is the starting point of the loop?

What if we do not know the total row number or how many elements in each row?

How to change the starting point to x and y?

Lab 2 discussion: arguments to main

Pass arguments to main function

- main (int argc, char *argv[])

- Examples – test_main.cpp
    - Test_1.cpp