C++ Programming
Lecture 2: Variables, References, Pointers, and Arrays

Nguyen, Thien Binh
Mechatronics and Sensor Technology
Vietnamese-German University

18 March 2019

Vietnamese - German University

Outline

1. Variables vs. References
2. Pointers
   - Pointers
   - Dynamic Memory Allocation
   - Warnings on the Use of Pointers
3. Constness
4. Arrays
   - Arrays with Fixed Sizes
   - Arrays with Dynamic Sizes
5. Matrix and Vector Operations: Coding 2

Vietnamese - German University

Outline

1. Variables vs. References

Vietnamese - German University

Variables vs. References

1. A variable:
   - has a name and an associated data type, e.g., `int i, j, k; double x, y, z;`
   - represents a memory location allocated to store a value of its data type
   - can be defined, initialized, or assigned at the same time
   - has a memory address which cannot be changed, and can be queried by the address operator `&i, &j, &k, &x, &y, &z`

```
double a, b;          // defining variables a, b
a = 100.0;            // assigning a
double c(10.0);       // defining and initializing c
double d = 1.0;       // defining and assigning d
cout << "a = " << a << ", addr of a = " << &a << endl;
cout << "b = " << b << ", addr of b = " << &b << endl;
cout << "c = " << c << ", addr of c = " << &c << endl;
cout << "d = " << d << ", addr of d = " << &d << endl;
```

Variables vs. References

2. A reference:
   - is defined by a data type and an `&`, e.g., `int &a; double &b;`
   - can be initialized only ONCE by assigning it to the variable it refers to, e.g., `int &a = i; double &b = x;`
   - creates a different name for already existing variables.
   - Both the variable and its reference share the same memory address.
   - Modifications of the reference also change the content of the variable to which it refers.

```cpp
    double a(10.0);
    double &b = a;

    cout << "a = " << a << ", &a = " << &a << endl;
    cout << "b = " << b << ", &b = " << &b << endl;
```

Listing 1: references.cpp

Variables vs. References

- What is the error?

```
double a(10.0);
double &b;
b = a;

cout << "a = " << a << ", &a = " << &a << endl;
cout << "b = " << b << ", &b = " << &b << endl;
```

Variables vs. References

Let a and b defined as in Listing 1. What are the outputs?

- Changing b:

```
    cout << "␣...␣Changing␣b␣..." << endl;
    b = 5.0;
    cout << "␣a␣=␣" << a << ";␣&a␣=␣" << &a << endl;
    cout << "␣b␣=␣" << b << ";␣&b␣=␣" << &b << endl;
```

- Changing a:

```
    cout << "␣...␣Changing␣a␣..." << endl;
    a = 50.0;
    cout << "␣a␣=␣" << a << ";␣&a␣=␣" << &a << endl;
    cout << "␣b␣=␣" << b << ";␣&b␣=␣" << &b << endl;
```

Variables vs. References

Let a and b defined as in Listing 1. What are the outputs?

- Changing the address of a:

```
    cout << " ... Changing address of a ... " << endl;
    &a = 50;
    cout << " a = " << a << "; &a = " << &a << endl;
    cout << " b = " << b << "; &b = " << &b << endl;
```

- Changing the address of b:

```
    cout << " ... Changing address of b ... " << endl;
    &b = 50;
    cout << " a = " << a << "; &a = " << &a << endl;
    cout << " b = " << b << "; &b = " << &b << endl;
```

Variables vs. References

Let a and b defined as in Listing 1. What are the outputs?

- Re-assigning b:

```
int &d = b;
double e = b;
b = 10;
cout << " a = " << a << "; &a = " << &a << endl;
cout << " b = " << b << "; &b = " << &b << endl;
cout << " e = " << e << "; &e = " << &e << endl;
```

Vietnamese - German University

Outline

2. Pointers

Pointers

- A pointer:
  - is defined by a data type and an `*`, e.g., `int *a; double *b;`
  - stores the memory address (not the value) of the variable or function it points to
  - whose type is the type of the variable it points to followed by `*`, e.g., `int*` for a pointer to `int`
  - allows changing the memory address it stores, which is the address of the variable it points to.
  - One can access to the value of the variable a pointer points to by the *dereference* operator `*`, e.g., `*a`
  - In order to query the data or methods of an object associated with a pointer, one can use the operator `->`, e.g., `(*a).function()` or `a->function()`

Vietnamese - German University

Pointers

```cpp
int    a = 12;        // a is a variable of type int
int    *b = &a;       // b is a pointer which stores &a
int    **c;           // c is a pointer pointing to pointer b
c = &b;               // assigning &b to c
double* d;            // pointer of type double
//d = &a;             // NOT ALLOWED due to difference of types

cout << "a = " << a << ", & a = " << &a << endl;
cout << "b = " << b << ", &b = " << &b << endl;
cout << "... Dereferencing b ..." << endl;
cout << "*b = " << *b << endl;
cout << "c = " << b << ", &c = " << &c << endl;
cout << "... Dereferencing c ..." << endl;
cout << "*c = " << *c  << ",  **c = " << **c << endl;
```

Listing 2: pointers.cpp

Fixing a, *b, and **c as in Listing 2. What are the outputs for the following codes?

```
cout << "... Changing a ..." << endl;
a = 100;
cout << "a = " << a << ", & a = " << &a << endl;
cout << "b = " << b << ", &b = " << &b << endl;
cout << "*b = " << *b << endl;
cout << "c = " << b << ", &c = " << &c << endl;
cout << "*c = " << *c  << ",  **c = " << **c << endl;
```

```
cout << "... Changing b ..." << endl;
b = b + 10;
cout << "a = " << a << ", & a = " << &a << endl;
cout << "b = " << b << ", &b = " << &b << endl;
cout << "*b = " << *b << endl;
cout << "c = " << b << ", &c = " << &c << endl;
cout << "*c = " << *c  << ",  **c = " << **c << endl;
```

Vietnamese - German University

Pointers

- *sum* $=$? What is wrong here?

```
int i, j, sum;
int *p_i, *p_j;

i = 10;  j = 20;
p_i = &i;

sum = *p_i + *p_j;

cout << "i = " << i << ", &i = " << &i << endl;
cout << "j = " << j << ", &j = " << &j << endl;
cout << "p_i = " << p_i << ", &p_i = " << &p_i << endl;
cout << ", *p_i = " << *p_i << endl;
cout << "p_j = " << p_j << ", &p_j = " << &p_i << endl;
cout << ", *p_j = " << *p_j << endl;
cout << "sum = " << sum << endl;
```

Vietnamese - German University

Pointers

- What is wrong here? $\Rightarrow$ `p_j is declared but does not point to any memory location, i.e., any variable!`

```cpp
int i, j, sum;
int *p_i, *p_j;

i = 10;  j = 20;
p_i = &i;

sum = *p_i + *p_j;

cout << "i = " << i << ", &i = " << &i << endl;
cout << "j = " << j << ", &j = " << &j << endl;
cout << "p_i = " << p_i << ", &p_i = " << &p_i << endl;
cout << ", *p_i = " << *p_i << endl;
cout << "p_j = " << p_j << ", &p_j = " << &p_i << endl;
cout << ", *p_j = " << *p_j << endl;
cout << "sum = " << sum << endl;
```

VGU
Vietnamese - German University

- A void* pointer can be used to store the address of any data type. The size of a void* pointer equals that of an integer.
- But arithmetic operations are not valid for a void* pointer.

```
    int    i(10);
    double  x(2.0);
    void    *p;    // void pointer

    cout << "i = " << i << ", &i = " << &i << endl;
    cout << "x = " << x << ", &x = " << &x << endl;

    p = &i;
    cout << "p = " << p << "sizeof(p) = " << sizeof(p) <<endl;

    p = &x;
    cout << "p = " << p << "sizeof(p) = " << sizeof(p) <<endl;

    //++p;        // NOT allowed
```

void Pointers

- A pointer of any type can be assigned to a `void*` pointer, but not vice versa.

```
int     *p_i  = &i;
double   *p_x  = &x;
double   *p_y;
void*    p;

p = p_i;     // OK
p = p_x;     // OK
//

p_y = p;     // NOT allowed
```

- Void pointers are useful for generic programming when a general pointer is used to point at different objects of various types. In C++, this mechanism is replaced by using *templates*.

Vietnamese - German University

`NULL` pointers

- In most operating systems, memory location of address 0 is preserved for the operating systems, and programs are not allowed to access it.
- A `NULL` pointer equals 0, thus does not point to any objects, e.g.,
  `double *p = NULL;`
- It is a good coding habit to initialized a pointer to `NULL` or if it is not used. A pointer accidentally points to some junk memory location is not a runtime error and *very difficult to debug*. The code will be compiled and executed without errors but the results might be totally incorrect.

Vietnamese - German University

Dynamic Memory Allocation

- Question: A variable is a name given to a memory chunk allocated. The question is, how does C++ allocate memory? ⇒ 3 types of allocations
  1. *Static memory allocation:*
     - for static and global variables
     - allocated when the program is run and remains until it ends
     - is automatically allocated and de-allocated
  2. *Automatic memory allocation:*
     - for local variables, pointers,
     - automatically allocated when the variable is declared, and de-allocated when it is out of scope.
     - The memory is allocated on the stack memory.
- Both static and automatic memory allocation requires the specification of the variable size at **compile time**.
- Question: How to have memory allocated at **run time**, e.g., to allocate an array with unknown size at compile time?

VGU Vietnamese - German University

Dynamic Memory Allocation

3. *Dynamic memory allocation:*
   - for pointers.
   - The program manually asks the operating system to allocate a memory to where the pointer points.
   - manually allocated by the operator `new` and manually deleted by the operator `delete`.
   - The memory is allocated on the heap memory.
   - The `delete` operator does not actually delete anything. It simply free the pointed memory and allows the operating system to get access into this memory to do whatever tasks.
   - It is a good programming habit to point a dynamically allocated pointer to `NULL` after `delete`.

Dynamic Memory Allocation I

```
#include <iostream>
using namespace std;
int main()
{
  double a(10.0);
  double *p;
  // dynamically allocate p with type double
  p = new double;
  // assigning value of a to *p, not &a to p
  *p = a;

  cout << "a␣=␣" << a << ",␣&a␣=␣" << &a << endl;
  cout << "p␣=␣" << p << ",␣&p␣=␣" << &p << endl;
  cout << "*p␣=␣" << *p << endl;

  cout << "...␣changing␣a␣..." << endl;
  a= 5.0;
  cout << "a␣=␣" << a << ",␣&a␣=␣" << &a << endl;
  cout << "p␣=␣" << p << ",␣&p␣=␣" << &p << endl;
```

Dynamic Memory Allocation II

```
  cout << "*p␣=␣" << *p << endl;

  cout << "...changing␣*p␣..." << endl;
  *p = 10.0;
  cout << "a␣=␣" << a << ",␣&a␣=␣" << &a << endl;
  cout << "p␣=␣" << p << ",␣&p␣=␣" << &p << endl;
  cout << "*p␣=␣" << *p << endl;

  // manually delete
  delete p;
  p = NULL;

  return 0;
}
```

- Note that a and p are allocated on two different memory locations, thus independent from each other. Pointer p *does not* point to a.

Warnings on the Use of Pointers

- Trying to assign a pointer with a value, not a memory address

```
    double a(100.0);
    double *pa;    // pa is declared but not assigned yet
    *pa = a;       // trying to store a
                   //at a random memory allocation
    cout << "a = " << a << ", &a = " << &a << endl;
    cout << "pa = " << pa << ", &pa = " << &pa << endl;
```

- Unintended change of a variable value through a pointer

```
    double y(3.0);
    double *py;
    py = &y;
    cout << "y = " << y << endl;
    *py = 1.0;              // y changed unintendedly
    cout << "y = " << y << endl;
```

Vietnamese - German University

Warnings on the Use of Pointers

- **Memory leaks:** happen when dynamically allocated memories are not properly deleted $\Rightarrow$ these memory addresses stay there in the memory untouchable.
  - Forgot to free a dynamically allocated memory after use

```
int main()
{
    double a(100.0);
    double *pa;
    pa = new double;
    pa = &a;

    cout << "... pa allocated" << endl;
    cout << "pa = " << pa << ", &pa = " << &pa << endl;
    cout << "*pa = " << *pa << endl;

    return 0;
}
```

Vietnamese - German University

Warnings on the Use of Pointers

- **Memory leaks:** happen when dynamically allocated memories are not properly deleted ⇒ these memory addresses stay there in the memory untouchable.
  - Using a dynamically allocated pointer with `new` and `delete` to point to an automatically allocated variable

```
    double a(100.0);
    double *pa;
    pa = new double;
    cout << "... pa allocated" << endl;
    cout << "pa = " << pa << ", &pa = " << &pa << endl;
    cout << "*pa = " << *pa << endl;

    pa = &a;  // old memory lost --> memory leak!

    cout << "... pa points to &a" << endl;
    cout << "a = " << a << ", &a = " << &a << endl;
    cout << "pa = " << pa << ", &pa = " << &pa << endl;
    cout << "*pa = " << *pa << endl;

    delete pa;
```

Warnings on the Use of Pointers

- **Dangling pointers**: are pointers pointing to deallocated memories ⇒ could lead to unexpected behaviors run by run!
  - when trying to dereference or delete a deleted memory address

```
    double a(100.0);
    double *pa;
    pa = new double;
    pa = &a;

    delete pa;

    cout << "... dereference a deleted pointer" << endl;
    cout << "*pa = " << *pa << endl;

    cout << "... delete a deleted pointer" << endl;
    delete pa;
```

Warnings on the Use of Pointers

- **Dangling pointers**: are pointers pointing to deallocated memories ⇒ could lead to unexpected behaviors run by run!
  - when multiple pointers pointing to the same memory dynamically allocated

```
double *p1, *p2;
p1 = new double;
*p1 = 10.0; p2 = p1;

cout << "p1 = " << p1 << ", &p1 = " << &p1 << endl;
cout << "*p1 = " << *p1 << endl;
cout << "p2 = " << p2 << ", &p2 = " << &p2 << endl;
cout << "*p2 = " << *p2 << endl;

delete p1; p1 = NULL;

// p2 is now dangling
// since it points to the deallocated memory
cout << "... p1 deleted ..." << endl;
cout << "p2 = " << p2 << ", &p2 = " << &p2 << endl;
cout << "*p2 = " << *p2 << endl;
```

Outline

3. Arrays

Arrays with Fixed Sizes I

- Consider the following vector and matrix

$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

Question:How to store $\mathbf{v}$ and A in C++? $\Rightarrow$ Using arrays.

```
#include <iostream>
using namespace std;
typedef double    VECTOR, MATRIX;
int main() {
  VECTOR  v[4];        // 1D array
  MATRIX  A[4][4];     // 2D array

  for (int i = 0; i < 4; ++i)
    v[i] = i + 1;

  for (int i = 0; i < 4; ++i)
```

```cpp
        for (int j = 0; j < 4; ++j)
            A[i][j] = i + j + 1;

    for (int i = 0; i < 4; ++i)
        cout << "v[" << i << "] = " << v[i] << ", ";
    cout << endl;

    for (int i = 0; i < 4; ++i)
    {
        for (int j = 0; j < 4; ++j)
        {
            cout << "A[" << i <<"][" << j  << "] = " ;
            cout << A[i][j] << ", ";
        }
        cout << endl;
    }
    cout << endl;
    return 0;
}
```

Arrays with Fixed Sizes

Notes:

1. The size of the arrays must be *fixed*, and *known at compile time*.
2. Arrays can be initialized with { } brackets when declaring

```
VECTOR v[4] = {1,2,3,4};
```

3. C++ is zero-based indexing, i.e., arrays starts from index 0.
4. When declaring an array, a *contiguous* memory chunk of the requested size is allocated.
5. 1D arrays, e.g., `v` are themselves pointers pointing to their first entry, i.e., `v[0]`.

```
double *pv;
pv = v;         // v itself is a pointer
//pv = &v[0]; // also OK.
cout << "v = " << v << ", *v = " << *v << endl;
cout << "&v[0] = " << &v[0] << ", v[0] = " << v[0] << endl;
cout << "pv = " << pv << ", *pv = " << *pv << endl;
```

Arrays with Fixed Sizes

Notes:

6. It is possible to define *arrays of pointers* in which each entry is a pointer, e.g.,

```
double x1(10.0), x2(1.0), x3(0.1);
double *px[3];
px[0] = &x1;
px[1] = &x2;
px[2] = &x3;
```

7. It is also possible to define a *pointer to an array* which stores only one address of the first entry

```
double (*p)[3];
```

Vietnamese - German University

Arrays with Fixed Sizes

Notes:

8. 2D arrays are pointers to an array in which each entry is a pointer pointing to a matrix row.

```
    double (*pA)[4];    // pointer to array
    pA = A;
    cout << "A = " << A << ", **A = " << **A << endl;
    cout << "&A[0][0] = " << &A[0][0]
         << ", A[0][0] = " << A[0][0] << endl;
    cout << "pA = " << pA << ", **pA = "
         << **pA << endl << endl;

    cout << "A[0] = " << A[0]
         << ", *A[0] = " << *A[0] << endl;
    cout << "&A[0][0] = " << &A[0][0]
         << ", A[0][0] = " << A[0][0] << endl;
    cout << "A[1] = " << A[1]
         << ", *A[1] = " << *A[1] << endl;
    cout << "&A[1][0] = " << &A[1][0]
         << ", A[1][0] = " << A[1][0] << endl;
    cout << "A[2] = " << A[2]
         << ", *A[2] = " << *A[2] << endl;
```

Arrays with Fixed Sizes

Notes:

9. Thanks to the contiguity of the memory chunk allocated for a fixed array, one can use pointer arithmetic to navigate through the entries of an array.

```
VECTOR v[4] = {1,2,3,4};     // 1D array
VECTOR *pv;
pv = v;      // points to v[0]
cout << "v = " << v << ", *v = " << *v << endl;
cout << "&v[0] = " << &v[0]
     << ", v[0] = " << v[0] << endl;
cout << "pv = " << pv
     << ", *pv = " << *pv << endl;
pv += 2;     // points to v[2]
cout << "&v[2] = " << &v[2]
     << ", v[2] = " << v[2] << endl;
*pv = 40;    // modifying v[2] = 40
cout << "&v[2] = " << &v[2]
     << ", v[2] = " << v[2] << endl;
```

Vietnamese - German University

Arrays with Dynamic Sizes

- Consider the following cases:
  1. Want to declare arrays whose size is not given at compile time, for example, the size of a vector is inputted from a keyboard with `cin`, or the size of a matrix varies for each run.
  2. Want to declare a real large array, e.g., $A[10000000][10000000]$. Since fixed arrays are allocated on the stack memory which is of limited size, it is sometimes not possible to declare such a big array.
- Question: How to handle the above cases?

Vietnamese - German University

Arrays with Dynamic Sizes

- A possible solution for case 1: to estimate a maximal size of the matrix for all runs, then declare it with that size $\Rightarrow$ waste of memory if the matrix size varies a lot!
- A better solution: to use dynamic memory allocation! Works perfectly for case 2 since the memory is allocated on the heap which is much larger than the stack memory.

Vietnamese - German University

Arrays with Dynamic Sizes I

- Declaration and dynamic memory allocation:

```
    int size, numRows, numCols;
    SCALAR *alpha;
    VECTOR  *v;
    MATRIX  **A;

    // read from the keyboard
    alpha = new SCALAR;
    cout << "... Input alpha..." << endl;
    cin >> *alpha;
    cout << "... Input the size of the vector" << endl;
    cin >> size;
    cout << "... Input the row number of the matrix" << en
    cin >> numRows;
    cout << "... Input the column number of the matrix" <<
    cin >> numCols;

    // dynamic memory allocation
```

Arrays with Dynamic Sizes II

```
18     v = new VECTOR [size];
19     A = new MATRIX* [numRows];
20     for (int i = 0; i < numRows; ++i)
21       A[i] = new MATRIX [numCols];
```

- Notes:
  - `size`, `numRows`, and `numCols` are declared but not assigned thus unknown at the compile time. They are directly inputted from a keyboard during run time.
  - One does not need to re-compile the code each run time.
  - Similar to fixed arrays, matrix A is a pointer pointing to a 1D array of pointers $A[i]$, $i = 0, \ldots, \texttt{numRows-1}$ (line 15).

Arrays with Dynamic Sizes

- De-allocation:

```
1    delete alpha;
2    delete[] v;
3    for (int i = 0; i < numRows; ++i)
4      delete[] A[i];
5    delete[] A;
```

- Notes:
  - Line 1 and 2: `delete` and `delete[]` are used to de-allocate single variables and arrays (pairing with `new` and `new []`).
  - Since A is a pointer to an array of pointers $A[i]$, each of these pointers in the array must be de-allocated first before de-allocating A.

Arrays with Dynamic Sizes I

- The whole code:

```
#include <iostream>
#include <cstdlib>        // for random generator rand()
using namespace std;
typedef double    SCALAR, VECTOR, MATRIX;
int main()
{
   int size, numRows, numCols;
   SCALAR *alpha;
   VECTOR  *v;
   MATRIX  **A;

   // read from the keyboard
   alpha = new SCALAR;
   cout << "...␣Input␣alpha..." << endl;
   cin >> *alpha;
   cout << "...␣Input␣the␣size␣of␣the␣vector" << endl;
   cin >> size;
```

Arrays with Dynamic Sizes II

```cpp
    cout << "... Input the row number of the matrix" << endl;
    cin >> numRows;
    cout << "... Input the column number of the matrix" << endl
    cin >> numCols;

    // dynamic memory allocation
    v = new VECTOR [size];
    A = new MATRIX* [numRows];
    for (int i = 0; i < numRows; ++i)
      A[i] = new MATRIX [numCols];

    // initialize v and A with random numbers
    for (int i = 0; i < size; ++i)
      v[i] = (double)( 1 + rand() % 10 );

    for (int i = 0; i < numRows; ++i)
      for (int j = 0; j < numCols; ++j)
        A[i][j] = (double)( 1 + rand() % 10 );

```

Arrays with Dynamic Sizes III

```cpp
    // print to the screen
    cout << "... scalar alpha..." << endl;
    cout << *alpha << endl << endl;

    cout << "... vector v..." << endl;
    for (int i = 0; i < size; ++i)
      cout << v[i] << ", ";
    cout << endl << endl;

    cout << "... matrix a..." << endl;
    for (int i = 0; i < numRows; ++i)
    {
      for (int j = 0; j < numCols; ++j)
        cout << A[i][j] << ", ";
      cout << endl;
    }
    cout << endl;

    // manually de-allocate the variables
```

Arrays with Dynamic Sizes IV

```
    delete alpha;
    delete[] v;
    for (int i = 0; i < numRows; ++i)
      delete[] A[i];
    delete[] A;

    return 0;
  }
```

Outline

4. Constness

Vietnamese - German University

Constness

- To define a constant in C++, either `const` (keyword) or `#define` (preprocessor directive, will be discussed in detail later) is used, although the latter is not recommended.
- Once assigned with `const`, a constant cannot be modified
- `#define` can be redefined anywhere in the program $\Rightarrow$ could be a source of bugging for constness!

Vietnamese - German University

Constness

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

#define PI  3.141592653589793        // double precision
const double SOUNDSPEED = 343;       // m/s

int main()
{
  cout.precision(16);      // double precision
  cout << fixed;
  cout << "PI = " << PI
       << ",  SOUNDSPEED = " << SOUNDSPEED << endl;
  #define PI  3.1415927    // single precision
  //SOUNDSPEED = 300;      // NOT allowed
  cout << "PI = " << PI
       << ",  SOUNDSPEED = " << SOUNDSPEED << endl;
  return 0;
}
```

Outline

5. Matrix and Vector Operations

Matrix and Vector Operations

In linear algebra, let $\mathbf{v}, \mathbf{w} \in \mathbb{R}^m$ and $A, B \in \mathbb{R}^{m \times n}$ where m and n is the number of rows (size of vectors) and number of columns, respectively, e.g.,

$$\mathbf{v} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_i \\ \vdots \\ v_{m-1} \end{bmatrix}, \quad A = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,j} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,j} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \cdots & \vdots & \ddots & \vdots \\ a_{i,0} & a_{i,1} & \cdots & a_{i,j} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,2} & \cdots & a_{n-1,n-1} \end{bmatrix},$$

or in short form,

$$\mathbf{v} = (v_i), \mathbf{w} = (w_i), A = (a_{ij}), B = (b_{ij})$$

Vietnamese - German University

Matrix and Vector Operations

- Vector/Matrix summation:

$$A + B = (a_{ij} + b_{ij}), \quad i = 0, \ldots, m-1,\ j = 0, \ldots, n-1 \tag{1}$$

- Matrix-Vector multiplication: possible if $\mathbf{v} \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m \times n}$

$$\mathbf{w} = A\mathbf{v} = (w_i) = \sum_{k=0}^{m-1} a_{ik} * v_k, \quad i = 0, \ldots, m-1 \tag{2}$$

- Matrix-Matrix multiplication: possible if $A \in \mathbb{R}^{m \times l}$ and $A \in \mathbb{R}^{l \times n}$. The product is $C \in \mathbb{R}^{m \times n}$

$$C = A * B = (c_{ij}) = \sum_{k=0}^{l-1} a_{ik} * b_{kj},\ i = 0, \ldots, m-1,\ j = 0, \ldots, n-1 \tag{3}$$

- Dot product: let $\alpha \in \mathbb{R}$ be a scalar. The dot product of 2 vectors of the same size is

$$\alpha = \mathbf{v} \cdot \mathbf{w} = \sum_{i=0}^{m-1} v_i * w_i \tag{4}$$

Vietnamese - German University

Matrix and Vector Operations

Coding 2: Write a C++ program to compute the following

1. Use dynamic memory to allocate $\mathbf{v}$, $\mathbf{w}$, $\mathbf{t}, \mathbf{u} \in \mathbb{R}^5$, and $A \in \mathbb{R}^{5\times4}$, $B \in \mathbb{R}^{5\times4}$, $C \in \mathbb{R}^{5\times4}$, $D \in \mathbb{R}^{4\times5}$, $E \in \mathbb{R}^{5\times5}$. All these are of type `double`
2. Initialize $\mathbf{v}, \mathbf{w}$, A, B, D with random numbers, and $\mathbf{t} = \mathbf{u} = 0$, $C = 0, E = 0$
3. Print them out to the screen
4. Compute $\mathbf{t} = \mathbf{v} + \mathbf{w}$, $C = A + B$, $u = C * t$, $E = C * D$, $\alpha = \mathbf{v} \cdot \mathbf{u}$. Use `assert` to check if the sizes of the vectors or matrices are valid for each computation
5. Print the results to the screen
6. Manually de-allocate using `delete`.

Vietnamese - German University

Matrix and Vector Operations I

```cpp
#include <iostream>
#include <cassert>       // for assert
#include <cstdlib>       // for random generator rand()
using namespace std;
typedef double SCALAR, VECTOR, MATRIX;
typedef int SIZE_VEC ,SIZE_MAT;
const double ZERO = 0.0000000000000000000000000;
const int m = 5;
const int n = 4;
int main()
{
   SCALAR alpha(ZERO);
   VECTOR *v, *w, *t, *u;
   MATRIX **A, **B, **C, **D, **E;

   SIZE_VEC size_v, size_w, size_t, size_u;
   SIZE_MAT numRows_A, numCols_A;
   SIZE_MAT numRows_B, numCols_B;
   SIZE_MAT numRows_C, numCols_C;
   SIZE_MAT numRows_D, numCols_D;
   SIZE_MAT numRows_E, numCols_E;
```

Matrix and Vector Operations II

```
// dynamic memory allocation with new

// initialize v, w, A, B, D with random number from 1 to 10
for (int i = 0; i < size_v; ++i)
  v[i] = (double)( 1 + rand() % 10 );

// set t = u = 0, C = 0, E = 0
for (int i = 0; i < size_t; ++i)
  t[i] = ZERO;

// print the initialized objects to the screen

// do the required computations with assert
// vector addition
assert(size_v == size_w);    // returns runtime error if not true
for (int i = 0; i < size_t; ++i)
  t[i] = v[i] + w[i];

// matrix addition

// matrix-vector multiplication

```

```cpp
    // matrix-matrix multiplication

    // dot product

    // print the results to the screen

    // manually deallocate the pointers

    return 0;
}
```

Listing 3: Coding1.cpp

Reading

1. Capper, *Introducing C++ for Scientists, Engineers, and Mathematicians*, **Chapters 6 - 7**
2. Pitt-Francis, and Whiteley, *Guide to Scientific Computing in C++*, **Chapter 4**

Vietnamese - German University