Advanced Programming
Topic 2: Pointers and Arrays

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

21 January 2024

Outline

1. Variables vs. References
2. Pointers
3. Arrays and Dynamic memory allocation
4. Warnings on the use of pointers
5. Constness

Outline

1. Variables vs. References

Variables vs. References

1. A variable:
   - has a name and an associated data type, e.g., `int i, j, k; double x, y, z;`
   - can be defined, initialized, or assigned at the same time

```
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
```

Variables vs. References

1. A variable:
   - represents a memory location allocated to store a value of its data type
   - has a memory address which cannot be changed, and can be queried by the address operator `&i, &j, &k, &x, &y, &z`

```
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;`

```cpp
    double a(10.0);
    double &b = a;  // initializing reference b
```

Listing 1: references.cpp

- A reference creates a different name for already existing variables.
- Both the variable and its reference share the same memory address.

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

Listing 2: 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;
```

Vietnamese - German University

Variables vs. References

Note:

- Modifications of the reference also change the content of the variable to which it refers.

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

Note:

- Once assigned an address, both a variable and its reference cannot change to another address.
- In this sense, a reference can be considered as a `constant pointer`.

Vietnamese - German University

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;
```

Variables vs. References

Note:

- References of different data types cannot be assigned by one another.

Outline

2. Pointers

Pointers

- A pointer:
  - is defined by a data type and an `*`, e.g., `int *a; double *b;`

```
int   a = 12;  // variable of type int
int   *b = &a; // pointer b stores &a
```

  - whose type is the type of the variable it points to followed by `*`, e.g., `int*` for a pointer to `int`

```
int   a = 12;  // variable of type int
int   *b = &a; // pointer b stores &a
cout << "size of int pointer = "
   << sizeof(int*) << endl;
```

  - stores the memory address (not the value) of the variable or function it points to

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

Pointers

- A pointer:
  - One can access to the value of the variable a pointer points to by the *deference* operator `*`, e.g., `*b`

```
        int a = 12; // variable of type int
        int *b = &a; // pointer b stores &a
        int **c;   // c points to b
        c = &b;

        cout << "... Dereferencee b ..." << endl;
        cout << "*b = " << *b << endl;
        cout << "... Dereferencee c ..." << endl;
        cout << "*c = " << *c << "**c = " << **c << endl;
```

  - Using *deferencing*, one can also *modify* the value of a variable that the pointer points to.

```
        cout << "... Modifying value of a ..." << endl;
        *b = 120;
        cout << "a = " << a << ", &a = " << &a << endl;
```

Pointers I

- A pointer:
  - allows changing the memory address it stores, which is the address of the variable it points to.

```
        int a1 = 12;
        int a2 = 120;
        int *b;

        cout << "a1 = " << a1
            << ", &a1 = " << &a1 << endl;
        cout << "a2 = " << a2
            << ", &a2 = " << &a2 << endl;

        b = &a1; // b stores addr. of a1

        cout << "b = " << b
            << ", &b = " << &b << endl;
        cout << "... Dereferencee b ..." << endl;
        cout << "*b = " << *b << endl;
```

Pointers II

```
b = &a2; // b stores addr. of a2

cout << "b = " << b
    << ", &b = " << &b << endl;
cout << "... Dereferencee b ..." << endl;
cout << "*b = " << *b << endl;
```

Vietnamese - German University

Pointers

What is the output for the following code?

```cpp
int a = 12;
int *b = &a;
int **c;
c = &b;

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

Pointers

What is the output for the following code?

```
int a = 12;
int *b = &a;
int **c;
c = &b;

cout << "... Changing b ..." << endl;
b = b + 10; // What is changed here?
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;
```

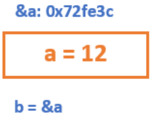
Pointers

Note:

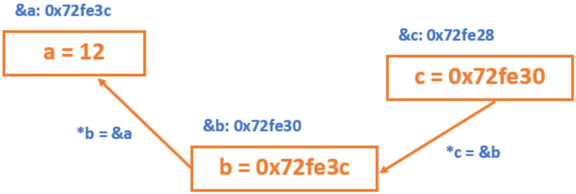
- Modifying the value of pointers is changing the object it points to.

References vs Pointers

- A reference = variable's alias (another name, nickname)
  - Both refer to the same memory addresses



- A pointer = a variable pointing to another variable
  - Both are allocated different memory addresses



Pointers

- *sum* =? What is wrong here?

```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;
```

Pointers

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

```
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;
```

Outline

3. Arrays and Dynamic memory allocation

Vietnamese - German University

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++?

Arrays with Fixed Sizes I

Question: How to store **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)
    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;
```

Arrays with Fixed Sizes II

```
    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;
  }
```

Vietnamese - German University

Arrays with Fixed Sizes I

Notes:

1. The size of the arrays must be *fixed*, and *known at compile time*.

```
VECTOR   v[4];         // 1D array
MATRIX   A[4][4];      // 2D array
```

2. Arrays can be directly 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.

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

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] << en
cout << "pv␣=␣" << pv << ",␣*pv␣=␣" << *pv << endl;
```

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

Arrays with Fixed Sizes III

```
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];
```

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

Arrays with Fixed Sizes IV

```cpp
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 V

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
```

Arrays with Fixed Sizes VI

```
    cout << "&v[2] = " << &v[2]
         << ", v[2] = " << v[2] << endl;
```

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?

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

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: (to be discussed later)*
     - ★ 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.

```
    double  a;        // scalar number
    double  v[4];     // 1D array
    double  A[4][4];  // 2D array
```

⇒ Both static and automatic memory allocation requires the specification of the variable size at **compile time**.

3. *Dynamic memory allocation:*
   - for pointers ONLY.
   - is manually allocated by the operator `new` and manually deleted by the operator `delete`.

```
#include <iostream>
using namespace std;
int main()
{
  double a(10.0);
  double *p;
  // manually 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;
```

Dynamic Memory Allocation II

```

  // manually delete
  delete p;
  p = NULL;

  return 0;
}
```

Notes:

- 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`.

Vietnamese - German University

Dynamic Memory Allocation

Question: What is the difference between `p1` and `p2`?

```
double a(10.0);
double *p1, *p2;

p1 = &a;
p2 = new double;
*p2 = a;

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

delete p2;
p2 = NULL;
```

Dynamic Memory Allocation

Question: What is the difference between p1 and p2?

- Try changing a, and observe the changes in p1 and p2

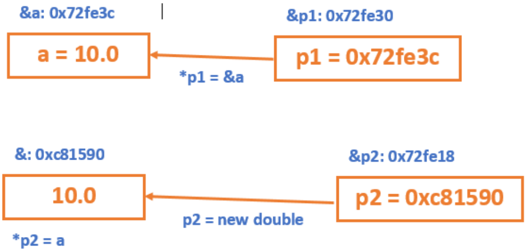
```
cout << "... CHANGING a ..." << endl;
a = 100.0;
cout << "p1 = " << p1 << ", &p1 = " << &p1 << endl;
cout << "*p1 = " << *p1 << endl;
cout << "p2 = " << p2 << ", &p2 = " << &p2 << endl;
cout << "*p2 = " << *p2 << endl;
```

⇒ *p1 is modified according to the change of a, but *p2 is NOT. Why is that?

⇒ Note that p2 DOES NOT store &a, i.e., the address of a. Instead, p2 points to a memory asked by new.

Vietnamese - German University

Question: What is the difference between `p1` and `p2`?



Notes:

- `p2` does not point to `a`, thus independent from each other.
- `p1` is *automatically* allocated at **compile time**.
- `p2` is *dynamically* allocated with `new` and `delete` at **run time**.

Task: Write a C++ code to create vector `v` and matrix `A` of random entries with vector's size (`size`), and number of matrix rows (`numRows`) and columns (`numCols`) are inputted from keyboard with `cin`.

Arrays with Dynamic Sizes I

- Vector and matrix sizes read from input keyboard:

```cpp
int size, numRows, numCols;
double  *v;
double  **A;

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

Arrays with Dynamic Sizes II

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

- Note: A is a 2D array, which is a pointer of pointers

- Initializing the vector and matrix with *random* entries

Vietnamese - German University

Arrays with Dynamic Sizes III

```
// 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 );
```

- Print the vector and matrix to the screen

Arrays with Dynamic Sizes IV

```
// print to the screen
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;
```

- De-allocation after use

Arrays with Dynamic Sizes V

```
// de-allocation
delete[] v;
for (int i = 0; i < numRows; ++i)
  delete[] A[i];
delete[] A;
```

- Note: 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.

Vietnamese - German University

Outline

4. Warnings on the use of pointers

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;
```

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.
  - Forgot to free a dynamically allocated memory after use

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

    cout << "... pa allocated" << endl;
    cout << "pa = " << pa << endl;
    cout << "&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 $\Rightarrow$ 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 << "\n";
    cout << "*pa = " << *pa << endl;

    pa = &a;  // old memory lost --> memory leak!
    cout << "... pa points to &a" << "\n";
    cout << "a = " << a << ", &a = " << &a << "\n";
    cout << "pa = " << pa << ", &pa = " << &pa << "\n";
    cout << "*pa = " << *pa << "\n";

    delete pa;
```

Warnings on the Use of Pointers

- **Dangling pointers**: are pointers pointing to deallocated memories $\Rightarrow$ 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 \n";
cout << "*pa = " << *pa << endl;

cout << "... delete a deleted pointer \n";
delete pa;
```

Warnings on the Use of Pointers I

- **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
```

Warnings on the Use of Pointers II

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

Outline

5. Constness

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;
}
```

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