Arrays

Arrays

- Suppose we are dealing with a vector of 1000 numbers of type `int`. We could define 1000 variables: `int x0, x1, x2, ...`, but that would not be convenient!
- Arrays are the solution
- Arrays are *collections of values of the same type*
- The number of entries of an array is called its *size* or its *length*
- Arrays have a fixed size (i.e. the size must be stated at compile time and cannot be changed during the execution of the program).

Arrays

```
type arrayName[size];
```

Explanations:

- The statement above creates an array ***arrayName*** which contains ***size*** values of type ***type***
- The ***size*** must be a constant value (not a variable, except constant variables)
- The i-th entry is accessed by ***arrayName***[i]
- The indices run from 0 to ***size***-1, not from 1 to ***size*** *!*
- Using forbidden indices (negative or greater than ***size***-1) is one of the most common programming errors

Arrays: Example 1

Task:

- Declare an array with 3 entries of type `int`
- Initialize all entries with some values
- Print all entries of the array to screen

Solution

```
int A[3];   // integer array A with 3 entries
A[0]=4;
A[1]=5;
A[2]=3;   // all entries initialized now
for(int i=0;i<3;i++)
    cout << A[i] << endl;
// for operations with all entries we
// usually use a for-loop
```

Arrays: Example 2

Task:

- Check what happens if forbidden indices of arrays are used
- Check what happens if entries of arrays are not initialized

Solution

```
int A[3];
A[100]=4;        // forbidden index, runtime error!
cout << A[100];
// no runtime error,  but unpredictable  result
cout << A[0];
// uninitialized entry, unpredictable  result
```

Problem

- Declare an array with 100 entries of type `double`
- Initialize the entries with random numbers
- Compute minimum, maximum and average of the entries
- Print the results to the screen

Problem

- Find all prime numbers less than 1,000,000
- Store the primes in a array of length 80,000 with entries of type `int`
- Use the following fact: an odd positive integer `n>1` is a prime if and only if it has no **prime divisor** `p` with $2 < p \leq \sqrt{n}$

Solution

```
 int main()
{
    unsigned primes[80000]; // over-dimensioned

    unsigned nPrimesFound = 0;
    unsigned upperBound = 100;

    for (unsigned i = 2; i < upperBound; ++i) {
        unsigned largest = static_cast<unsigned>(sqrt(static_cast<double>(i)));
        bool isPrime = true;
        for (unsigned j = 0; j < nPrimesFound; ++j) {
            if (primes[j] > largest)  // unnecessary to checks larger primes?
                break;  // exit from the loop in j
            if (i % primes[j] == 0) { // divisible by primes[j]?
                isPrime = false;
                break;  // exit from the loop in j
            }
        }
        if (isPrime)
            primes[nPrimesFound++] = i;
    }

    for (unsigned i = 0; i < nPrimesFound; ++i)
        cout << primes[i] << ", ";
    cout << endl;
    return 0;
}
```

Problem

- Find all prime numbers less than 1,000,000 using the sieve of Erastothenes

https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

- Hint: use an array of bool of size 1,000,000 and update it progressively as appropriate. Note: you do not need any other array: see the pseudocode in wikipedia!

Initializer Lists for Arrays

Syntax:

```
type arrayName[] = {values};
```

Explanations:

- The statement above creates an array *arrayName* with *values* as entries
- *values* is a comma separated list of values of the given *type*
- Examples

```
int x[] = {1, 7, 5};  // size 3: auto-inferred
int y[2] = {2, 5};    // size 2: explicitly stated
int z[4] = {2};       // size 4: last 3 initialized to 0
int p[2] = {1, 2, 3}; // size 2: compile error
```

Initializer Lists: Example

Task:

- Create an integer array containing the numbers 5,10,100,1000 using an initializer list.
- Print all entries of the array on the screen.

Solution

```
int A[] = {5,10,100,1000};
for(int i=0;i<4;i++)
    cout << A[i] << endl;
```

Arrays are Dangerous!

```
int A[10];
cout << A[2] << endl;  // error!
```

Result unpredictable since `A[2]` not initialized and thus undefined

---

```
int A[10];
A[10]=10;  // error!
```

Result unpredictable since `A[10]` is no valid array entry

This will not be flagged out as syntax errors. In debug mode, some debugger may detect the issue

Arrays are Dangerous

```
double A[1000000];  // error!
```

Runtime error, array too big for memory **stack.**
Don't use arrays in this way with more than 100,000 elements!

```
int n=5;
double A[n];  // error!
```

Compiler error. Size of an array must be an integer constant.

for on ranges

- In **C++ 11** a new variation of for-loop has been extended to work on ranges
- A range is anything on which we can iterate on
- For instance, it works with arrays

for (for-range-declaration : expression)

statement

Example:

```
int x[3] = {1,2,3};
for (int i : x)
   cout << i << endl;
```

for on ranges

- Often used with the **auto** keyword
- Very convenient with the STL where type names are very long, as we will see later

Example:

```
int x[3] = {1,2,3};
for (auto i : x)
   cout << i << endl;
```

Char Sequences

- Simply an array of type char
- There are lot of library functions which work with strings in this format
- By convention, they assume the sequence is terminated by the character 0
- See: http://www.cplusplus.com/doc/tutorial/ntcs/

How to Store a Matrix in an Array

The entry in row i and column j of a matrix A is denoted by $A_{i,j}$

$$
A_{1,1} \quad A_{2,4} \quad
\begin{pmatrix}
2.3 & -23 & -5 & 5 \\
66 & 345 & 5 & 45 \\
0 & 0 & 0 & 6 \\
3 & 5 & -56.8 & 4
\end{pmatrix}
\quad A_{4,2} \quad A_{3,3}
$$

We need to choose a mapping between the position of the elements in the matrix and a one dimensional array

How to Store a Matrix in an Array

- Suppose the matrix has **m** rows and **n** columns. We need a storage space large enough for m x n elements.

  ```
  double A[m*n];
  ```

- We need to choose a mapping between the position of the elements in the matrix and a one dimensional array
- An obvious choice is to map either by row or by column.
- Example, mapping `i=1..m, j=1..n` to the position in the array:
  - by row (i.e. **row major**): 2.3, -23, -5, 5, 66, 345, 5, 45, 0, 0, 0, 6, 3, 5, -56.8, 4

    i.e. $M_{i,j}$ corresponds to `A[(i-1)*n+j-1]`

    inner dimension is *j*, i.e. the dimension contiguous in memory
  - by column (i.e. **column major**) : 2.3, 66, 0, 3, -23, 345, 0, -5, -5, 5, 5, 0 -568, 5, 45, 6, 4

    i.e. $M_{i,j}$ corresponds to `A[(j-1)*m+i-1]`

    inner dimension is *i*, i.e. the dimension contiguous in memory

Problem

$$\text{Task :}$$

$$\text{Store the matrix } \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \text{ in an 1D array with}$$

$$\text{row - wise mapping using an initializer list,}$$

$$\text{then print the entries to the screen with a nested loop.}$$

High Dimensional Arrays

- Arrays can have more than one dimension
- For instance, a matrix whose dimensions m and n are known at compile time could be stored in the 2-D array

  ```
  double A[m][n]
  ```

  In this case the elements would be accessed with two indices

  ```
  cout << A[i][j] << endl;
  ```
- Dimension could be higher than 2
- The memory storage scheme is such that the right-most dimension is the inner storage dimension
- Not commonly used, as they require knowledge of all dimensions at compile time

Example – Matrix Multiplication

```
double A[3][2] = { { 1.2, 2.3 }
                 , { 2.3, 1.0 }
                 , { 1.0, 5.0 }
                 };
double B[2][3] = { { 1.0, 2.0, 3.0 }
                 , { 4.0, 5.0, 6.0 }
                 };
double C[3][3];

// nested loops
for (unsigned n = 0; n<3; ++n)
    for (unsigned m = 0; m<3; ++m) {
        C[n][m] = 0;
        for (unsigned k = 0; k<2; ++k)
            C[n][m] += A[n][k] * B[k][m];
    }
```

In this matrix multiplication example all matrices have constant size, are known at compile time, and are stored as multidimensional arrays

Note the nested loop

High Dimensional Array

- Like mono-dimensional arrays, they require that all dimensions are stated in advance.
- This is in general not possible, and hence they are rarely used
- There are cases where it may be appropriate. For example, if we are dealing with rotation matrices in 3D plan, then all matrices have constant size 3x3

Pointers to Data

Pointers to Data

- A pointer is a variable whose value is the memory address of another variable. We say it points to that address.
- Pointers can be used to access the values of variables indirectly and sometimes more efficiently
- Assign `NULL` to pointers for which a suitable address in not yet available (`NULL` pointer). NULL is a constant treated by convention as an **invalid** memory address. Same as zero.
- Pointers are often used in libraries which are written in C (not C++). To use these libraries, we need pointers.

•See
http://www.cplusplus.com/doc/tutorial/pointers/

Topics:
Pointer declaration
Memory address (& operator)
Dereferencing (* operator)
Pointer arithmetic
* precedence
const pointers

Pointers to Data

- See http://www.cplusplus.com/doc/tutorial/pointers/
- Topics:
  - Pointer declaration
  - Memory address (& operator)
  - Dereferencing (* operator)
  - Dereferencing with offset ([] operator)
  - Pointer arithmetic
  - Precedence of operators * and ++
  - const pointers
  - Pointers and arrays
  - Pointers to pointers

Problem

- Initialize the entries of an array A of size 100 to random numbers
- Compute the sum of entries of A in two ways:

1. As usual, sum up the A[i]

2. Define a pointer p to the first element of A, let p run over the addresses of all elements of A using pointer arithmetic, and sum up the values *p

High Dimensional Arrays Pointers

- In a muti-dimensional array the internal memory storage is equivalent to 1-D row-major scheme. If we assign a pointer to the first element of the matrix:

  ```
  double *p = &A[0][0];
  ```

  then

  `A[i][j]` is the equivalent to `p[i*m+j]`

Dynamic Memory Allocation

Arrays of Variable Size

- "Variable size" means that the size can be changed while the program is running ("at runtime") or that the size is not known at compile time
- Often we can use STL **vectors** instead of arrays if we need variable size, but sometimes the use of arrays cannot be avoided or is more efficient
- We need to know how to create and delete arrays of variable size ("dynamic memory allocation")

How Dynamic Memory Allocation cannot be done

```
int n;
cin >> n;
int A[n];
```

- This is an attempt to let the user determine the size of A at runtime
- Compiler error! (usually)
- sizes of arrays must be known at compile time
- size of array must be a literal or a const expression computable at compile time
- We need to use dynamic memory allocation.
- We'll see libraries as STL will provide us with easy-to-use and safe containers (e.g. vector)

new and delete

```
int *A = NULL;        // init to null pointer
int n;
cin >> n;             // input desired array size
A=new int[n];         // allocate memory and assign to A
...                   // do something with A
delete [] A;          // deallocate memory
A = NULL;             // reset to null pointer
```

- This is a correct way to create an array of variable size
- The delete command is necessary, otherwise the memory assigned to A is not released until the program terminates (“memory leak”). **This is a very common mistake and tricky to handle correctly. We’ll say more about this later.**
- Note: memory allocation can fail, if there is not enough memory left. It throws an exception (we’ll see exceptions later).

Problem

- Read a sequence of positive integers from the console in a loop and store them in a dynamic array resized on the fly.
- If an entry is invalid, ignore it
- Any negative number terminates the input sequence and cause the entered sequence to be printed to the screen.
- Hints
  - Allocate space for a dynamic array of size currentSize=10.
  - Keep track of the number of elements in the array (initially zero)
  - Prompt the user to enter an integer number
  - If the entry is a valid integer, store it at the at the next available empty slot
  - If the user doesn’t enter an integer, the expression cin becomes false, so we can use a condition if(!cin). To restore cin so that it can read input again, use cin.clear(); cin.ignore(10,’\n’); (the ignore part makes sure the previous incorrect input is ignored).
  - If there are no empty slots, create a new array with size 2*currentSize, copy the old array into the new array, deallocate the old array and swap the pointers

Solution (skeleton)

- We proceed top down, i.e. we write the skeleton of our application first
- The parts in red we leave for later implementation
- This allows us to see the big picture

```
int main()
{
    // initialize necessary variables (note the use of unsigned)
    unsigned int capacity = 10;   // storage capacity
    unsigned int size = 0;        // how many numbers have been stored so far
    unsigned int *numbers = new unsigned int[capacity];  // allocate memory storage

    // we start an infinite loop, which we will stop with 'break'
    // we keep asking the user to enter a number until a negative number is entered
    while (true) {
        // populate the array and, at the first negative input, terminate the loop with break
        // if necessary we increase the storage capacity
    }

    // the input loop is completed.

    // print the numbers
    for (auto i = 0u; i < size; ++i) // note the use of 'auto' and of suffix 'u'
        cout << numbers[i] << " ";
    cout << endl;

    // release memory
    delete[] numbers;

    return 0;
}
```

Solution (input loop)

```
// we start an infinite loop, which we will stop with 'break'
// we keep asking the user to enter a number until a negative number is entered
while (true) {

    int userInput;

    // obtain a valid integer from the console

    // if the input is negative, we terminate the input loop
    if (userInput < 0)
        break;

    // At this point we are sure have a valid integer input and we know it is positive

    // do we have enough space to store it? If not, we increase the storage space
    if (size == capacity) {
        // increase storage space
    }

    // At this point we are sure have a valid integer input, we know it is positive
    // and we know we have enough storage space to add it
    // So we add the new number to the existing storage
    // Note that in the following instruction 'size' is post-incremented
    // and ‘userInput’ is automatically casted from signed to unsigned
    numbers[size++] = userInput;
}
```

Solution (obtaining a valid input)

```
// obtain a valid integer from the console

bool invalidInput;  // we assume the input will be valid

do {  // this loop is executed at least once
    invalidInput = false;  // we assume the input will be valid

    // we prompt the user to enter a number
    cout << "Enter a non-negative integer or a negative to terminate: "; // (note: no end of line)
    cin >> userInput;

    if (!cin) {  // was the input invalid?
        cout << "Invalid input\n";
        // discard the bad input and clean up buffers
        cin.clear();
        cin.ignore(10, '\n');
        invalidInput = true; // this will cause the loop to continue
        }
} while (invalidInput);

// At this point we are sure that 'userInput' is assigned to a valid integer input
```

Solution (increase storage space)

```
// do we have enough space to store it? If not, we increase the storage space
if (size == capacity) {

    // increase capacity variable
    capacity *= 2;

    // allocate new storage space with the new larger capacity
    // note that the pointer variable 'tmp' is local to this statement block
    unsigned int *tmp = new unsigned int[capacity];

    // copy from previous small storage space to new large storage space
    for (auto i = 0u; i < size; ++i) // note the use of 'auto' and of suffix 'u'
        tmp[i] = numbers[i];

    // release old storage space
    delete[] numbers;

    // point the 'numbers' pointer to the new storage space
    numbers = tmp;
}
```

Full Solution

See file: InputNumberArray.cpp

Heap vs Stack

- We have seen two ways of storing vectors:

```
int a[100];
int *a = new int[100];
```

- In both cases, we are reserving memory
- What is the difference and what are the pros and cons?
- Arrays have fixed pre-defined size and are allocated on the **stack**
- Dynamic memory have variable size and are allocated on the **heap**

Stack

- The stack is like a scratch pad used by the program while it runs.
- It is **small** (we cannot store large arrays)
- Local variables are allocated there (so large arrays will cause runtime errors).
- Allocation and deallocation is automatic, we do not need to manage it explicitly
- Allocation and deallocation is always from the top of the stack. Therefore it is **very fast**.

Heap

- A **large** tank of memory.
- Allocation and deallocation happens via a formal request and need to be **explicitly managed** by the programmer.
- Allocation requests are satisfied by the heap allocator, which searches for a block of free memory with sufficient size and reserves it.
- Allocation is **very slow**

Heap vs Stack

- Memory management is a language specific concept
- This way of managing heap and stack are the C, C++ solution to it
- Often criticised because requires low level memory management and error prone. At the same time it enables nice memory / performance optimizations.
- Detailed understanding of the difference between heap and stack is beyond the scope of this course
- See: http://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap

struct

(http://www.cplusplus.com/doc/tutorial/structures/)

struct

- A composite type which aggregate multiple fundamental types into a single structure
- It is a new type, defined by the user

```
struct [type_name]
{
  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
  .
  .
} [object_names];
```

struct

- To access a member of a struct we use the ‘.’ operator

```
object_name.member_name;
```

struct (example 1)

```
// define the new type ComplexNumber
struct ComplexNumber
{
  double realPart;
  double imagPart;
};

// declare a variable of type ComplexNumber
ComplexNumber x;
// initialize the data members of x
x.realPart = 3.0;
x.imagPart = -2.2;
```

struct (example 2)

```
// define the new type ComplexNumber
struct ComplexNumber
{
  double realPart;
  double imagPart;
};

// declare a variable of type ComplexNumber
// and initialize using an initializer list
ComplexNumber x = {1.0, 2.3};
```

struct (example 3)

```
// define the new type ComplexNumber and declare
// two variables x and y of type ComplexNumber
struct ComplexNumber
{
  double realPart;
  double imagPart;
} x, y;

// initialize x and y
x.realPart = 3.0;
x.imagPart = -2.2;
y = x;  // copy the entire structure y into x
```

struct (example 4)

```
// define a new anonymous type and declare
// a variable x with this type
struct
{
  double realPart;
  double imagPart;
} x;

// initialize x and y
x.realPart = 3.0;
x.imagPart = -2.2;
```

pointers to struct

- Like any other type, structures can be pointed to by its own type of pointers
- To access a member of a pointer to struct we use the ‘->’ operator

```
object_pointer_name->member_name;
```

pointers to struct (example 1)

```
// define the new type ComplexNumber
struct ComplexNumber
{
  double realPart;
  double imagPart;
};

// declare a variable of type complexNumber
ComplexNumber x = {1.0, 1.0 };
ComplexNumber *ptr = &x;  // a pointer to x

// print the data members of y
cout << ptr->realPart << "," << ptr->imagPart << endl;
```

struct dynamic allocation

- struct variables can be allocated dynamically with new and delete
- we can also create arrays of struct

struct dynamic allocation (example 1)

```
// define the new type ComplexNumber
struct ComplexNumber
{
   double realPart;
   double imagPart;
};

// declare a pointer variable of type ComplexNumber
ComplexNumber *ptr = NULL;
ptr = new ComplexNumber;  // allocate memory (note no square brackets)

// initialize members
ptr->realPart = 3.0;
ptr->imagPart = -2.2;

// print the data members of y
cout << ptr->realPart << "," << ptr->imagPart << endl;

delete ptr;  // de-allocate memory (note no square brackets)
```

struct dynamic allocation (example 2)

```
// define the new type ComplexNumber
struct ComplexNumber
{
   double realPart;
   double imagPart;
};

// declare a pointer variable of type complexNumber
ComplexNumber *ptr = NULL;
ptr = new ComplexNumber[2];  // allocate memory for an array of size 2

// initialize members
ptr[0].realPart = 3.0;  // note we are using '.' not  '->'
ptr[0].imagPart = -2.2;
ptr[1] = ptr[0];  // copy ptr[0] to ptr[1]

// do something with ptr
...

delete [] ptr;  // de-allocate memory (note the square brackets)
```