Introduction to C++

Georgia Koutsandria

Internet of Things A.Y. 19-20

Prof. Chiara Petrioli
Dept. of Computer Science
Sapienza University of Rome

How to compile a C++ program

- **Windows**: Install an Integrated Development Interface (IDE).
  - Dev-C++ http://www.bloodshed.net/dev/index.html

- **Mac:** Install Xcode with the gcc/clang compilers.

```
g++ -std=c++11 example.cpp -o example_program OR
clang++ -std=c++11 -stdlib=libc++ example.cpp -o example_program
```

- **Linux:** Compile your code directly from the terminal using the following commnad `g++ -std=c++0x example.cpp -o example_program`

Pointers

Pointers

- Variables: Locations in the computer's memory which can be accessed by their identifier (their name).
- The address of a variable can be obtained by using the ampersand sign(&).
- **Pointer**: The variable/object whose value is the address in memory of another variable.

Pointers

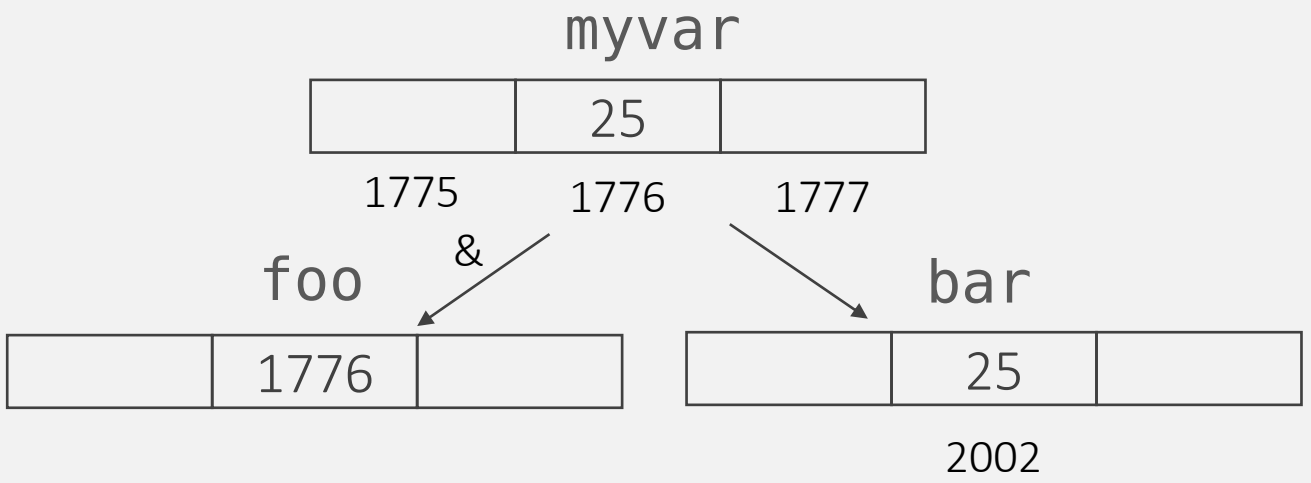
- **Pointer**: The variable/object whose value is the address in memory of another variable.

- *Dereferencing*: Acessing an object to which a pointer refers
  - Use the indirection operator, i.e., " * "
  - E.g., if p is a pointer, *p is the object to which the pointer refers

    - **Null pointer:** a special pointer value that does not refer to any valid memory location (*nullptr* keyword)

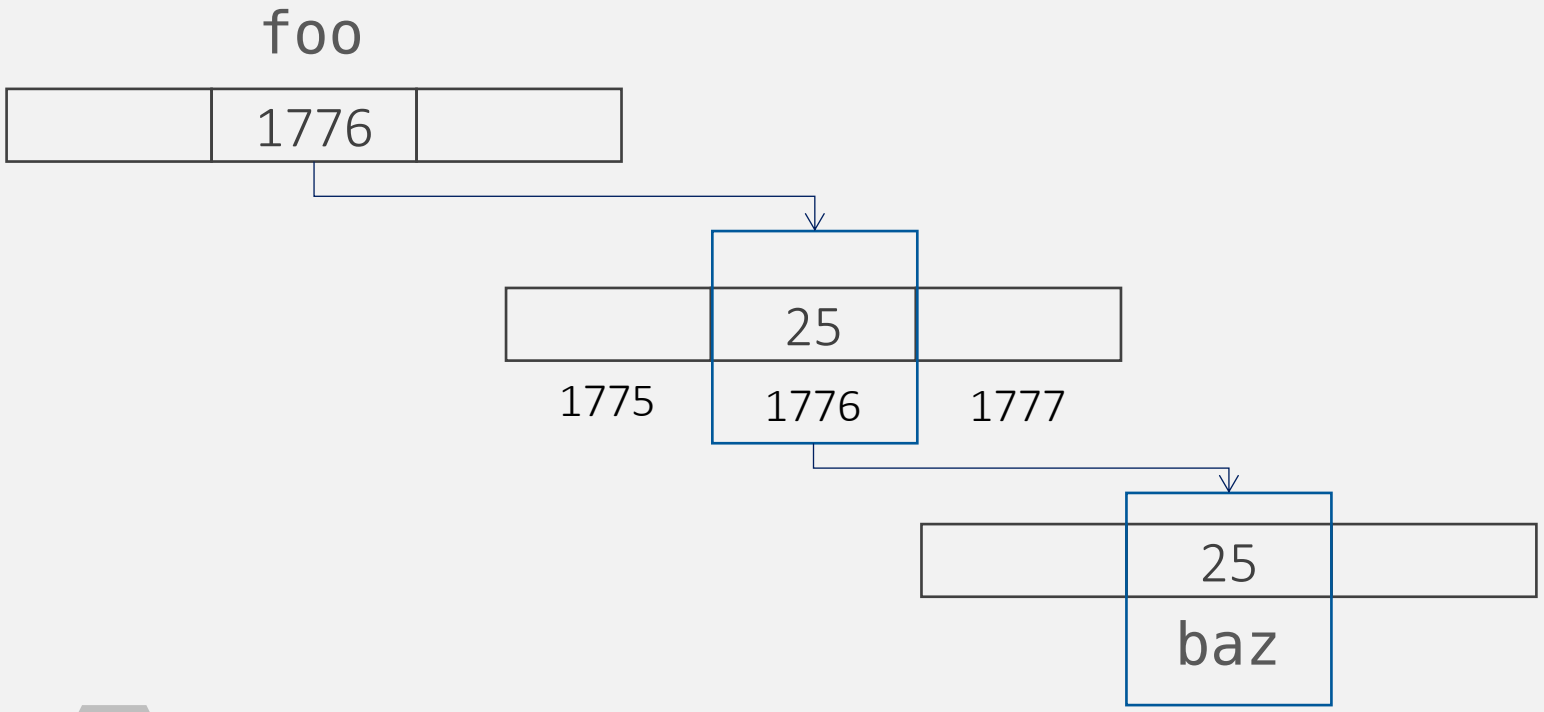
Pointers

- **Pointer**: The variable/object whose value is the address in memory of another variable.
- *Dereferencing*: Acessing an object to which a pointer refers
  - Use the indirection operator, i.e., " * "
  - E.g., if foo is a pointer, *foo is the object to which the pointer refers
-




```
myvar = 25;
foo = &myvar;
bar = myvar;
```

Pointers




```
myvar = 25;
foo = &myvar;
baz = *foo;
```

Declaring Pointers

- They have different properties when they point to a `char` than when they point to an `int` or `float`.
- Their declaration needs to include the data type are going to point to.
- Syntax: `type * name;`
- The asterisk means that a pointer is declared which should not be confused with the dereference operator.

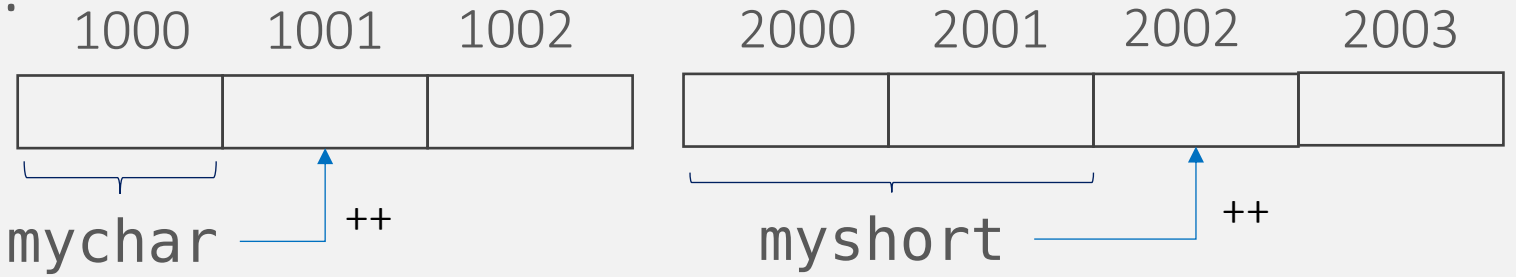
Pointers- An example

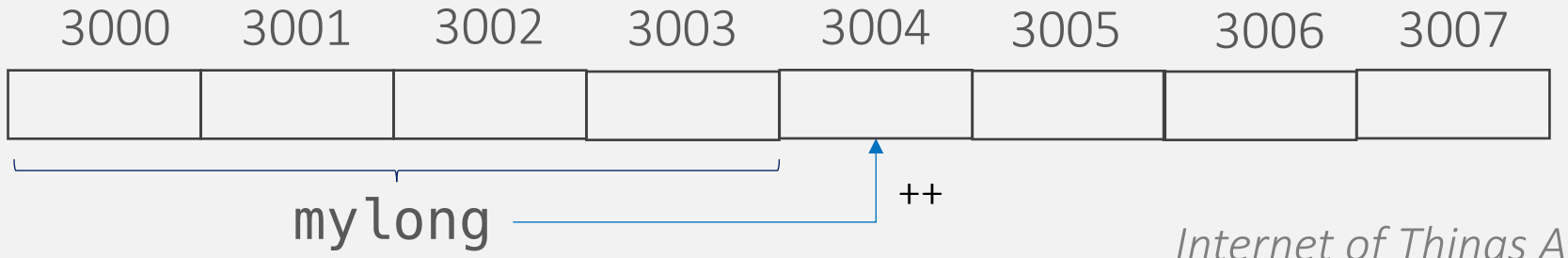
```
#include <iostream>
using namespace std;
int main(){
	int firstvalue = 0;

	int * mypointer;

	mypointer = &firstvalue;
	cout << "mypointer is " << mypointer << endl;
	cout << "firstvalue is " << *mypointer << endl;
	*mypointer = 10;
	cout << "firstvalue is " << firstvalue << endl;

	return 0;
}
```

Pointers arithmetics

- Only addition/subtraction operations are allowed.
- Operations depend on the size of the data type to which they point.
- E.g.: In a given system, a `char` takes 1 byte, a `short` takes 2 bytes, and `long` takes 4 bytes. 3 pointers that point to memory locations 1000, 2000, and 3000.

```
char * mychar;
short * myshort;
long * mylong;
```





Pointers arithmetics

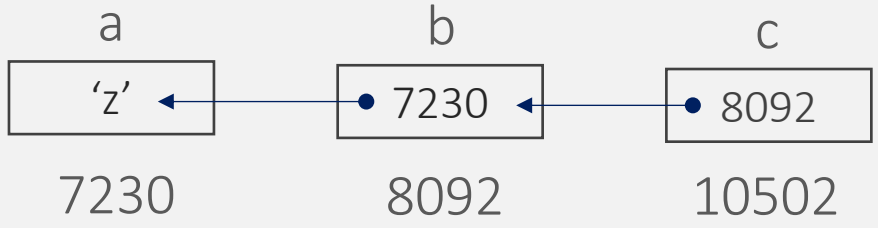
- The increment/decrement operators can be used as either prefix or suffix of an expression.
- The increment/decrement operator has a higher precedence than the *.

```
//incremement pointer, and dereference unincremented address
*p++;//same as *(p++);
//incremement pointer, and dereference incremented address
*++p; //same as *(++p);
//dereference pointer, and increment the value it points to
++*p; //same as ++(*p);
//dereference pointer, and post-increment the value it
points to
(*p)++;
```

Pointers to Pointers

- The syntax requires an asterisk (*) for each level of indirection in the declaration of the pointer.

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```



- Variable c can be used in three different levels of indirection
  1. c is of type `char**` and has a value of 8092.
  2. `*c` is of type `char*` and has a value of 7230.
  3. `**c` is of type `char` and has a value of `'z'`.

Pointers and Arrays

- An array can always be implicity converted to a pointer of a proper typer.
- Pointers and arrays support the same set of operations.
- **Exception**: Pointers can be assigned a new address, while arrays cannot.
- The name of an array can be used like a pointer to its first element.

Pointers and Arrays-An example

```
#include <iostream>
using namespace std;
int main(){
	int numbers[5];
	int * p;
	p = numbers; *p = 10;
	p++; *p = 20;
	p = &numbers[2]; *p = 30;
	p = numbers + 3; *p = 40;
	p = numbers; *(p+4) = 50;
	for (int n=0;n<5;n++)
		cout << numbers[n] << "\n";
	return 0;
}
```

Pointers and Functions

- C++ allows to pass a pointer to a function
- The function parameter(s) should be declared as a pointer
- Changes on the value of the pointer inside the function reflect back in the calling function.

Pointers and Functions– An example

```
#include <iostream>
using namespace std;
double calcAverage(int *arr, int size);
int main(){
        int numbers[5] = {2, 8, 10, 20};
        double avg;
        avg = calcAverage(numbers, 4);
        cout << "Average is: " << avg << endl;
        return 0;
}

double calcAverage(int *arr, int size){
         int sum = 0;
         double k;
        for (int n=0;n<size;n++)
                sum +=arr[n];
        k = double(sum)/size;
        return k;

}
```

Exercise 1

What is the exact output of the following program?

```
#include <iostream>
using namespace std;
int main(){
	int array[3]={3, 5, 10};
	int * p = array;
	cout << "Print a: " << endl;
	for (int n=0;n<3;n++)
		cout << *(p+n)+2 << endl;
	cout << "Print b: " << endl;
	for (int k=2;k>=0;k--)
		cout << *(p+k)- 2 << endl;

	return 0;
}
```

Exercise 1--Solution

```
#include <iostream>
using namespace std;
int main(){
	int array[3]={3, 5, 10};
	int * p = array;
	cout << "Print a: " << endl;
	for (int n=0;n<3;n++)
		cout << *(p+n)+2 << endl;
	cout << "Print b: " << endl;
	for (int k=2;k>=0;k--)
		cout << *(p+k)- 2 << endl;

	return 0;
}
```

```
Print a:
5
7
12
Print b:
8
3
1
```

Exercise 2

Write a program to print the elements of an array in reverse order using pointers. Print the elements of the array before and after reversing it.

Exercise 2-Solution

```
#include <iostream>
using namespace std;
int main(){
	int array[4] = {1, 2, 3, 4};
	int *p = array;
	cout << "Array before reversing: " << endl;
	for (int i=0;i<4;i++)
		cout << *(p+i) << endl;
	cout << "These reversed array is: " << endl;
	for (int j=3;j>=0;j--)
		cout << *(p+j) << endl;
	return 0;
}
```

Exercise 3

Write a program with a function that swaps (exchanges the values of) two integer numbers. You should pass the arguments to the function using pointers. Display the values of the two numbers before and after swapping them.

Exercise 3-Solution

```
#include <iostream>
using namespace std;
void swap(int* n1, int* n2) {
	int temp;
	temp = *n1;
	*n1 = *n2;
	*n2 = temp;
}
int main(){
	int a = 10, b = 20;
	cout << "Before swapping: ";
	cout << "a = " << a << ", b = " << b << endl;
	swap(&a, &b);
	cout << "After swapping: ";
	cout << "a = " << a << ", b = " << b << endl;
	return 0;
}
```

Containers in the C++ standard library

Containers

- Container: stores a collection of other objects (elements).
- Containers library: a collection of class templates and algorithms; allows flexibility to the programmer.
- Two main categories of containers
  - Sequential
  - Associative: Ordered; Unordered (c++11)

- Q: Which container to choose?
  - A: - Functionality offered by the container.
    - Efficiency/complexity of its members.

Sequential Containers in the C++ standard library

Sequential Containers

- Standard library includes several container types
  - E.g., `array(c++11)`, `vector`, `list`, `forward_list(c++11)`, `deque`.
- The order of the elements corresponds to the positions in which the elements are added to the container (they can be accessed sequentially).
- Built-in functions, e.g., sorting and ordering.

Sequential Containers

- `Array`: A fixed-size container.
- Other containers: We can add or remove elements, growing and shrinking the size of the container.
- Containers offer different performance trade-offs relative to
  - The cost to add or delete elements
  - The cost to perform nonsequential access to the elements.

Sequential Containers

- `Array`:
  - Fixed-size
  - Fast random access
  - Cannot add/delete elements.

- `Vectors`:
  - Hold their elements in contiguous memory
  - Fast random access
  - Fast insert/delete at the back
  - Insert/delete other than the back may be slow.

Sequential Containers

- `Deque`:
  - Supports fast random access
  - Adding/removing elements in the middle of a deque is a (potentially) expesnsive operation.
  - Adding/removing elements at either end is a fast operation.
- `List` and `forward_list`:
  - Fast insert/delete at any point of the list
  - Do not support random access to elements; access is done by iterating through the container
  - Substantial memory overhead.

Which sequential container to use?

- Unless you have a reason to use another container, use a `vector`.
- Lots of small elements and space overhead matters, don't use `list` or `forward_list`.
- Random access to elements: `vector` or `deque`.
- Insert/delete elements in the middle of the container: `list` or `forward_list`.
- Insert/delete elements at the front and the back (not in the middle): `deque`.

Which sequential container to use?

The predominant operation of the application (whether it does more access or more insertion or deletion) will determine the choice of the container type.

Array

- A fixed-size sequence container; No memory management.
- Holds a specific number of elements ordered in a strict linear sequence.
- Appropriate header: `#include <array>`

```
//array holds 2 objects of type int; initialized
array<int, 2> myarray = {2, 8};
//array holds 2 objects of type int; initialized
array<int, 2> myarray{2, 8};
//10 objects of type int
array<int, 10 > myarray;
```

Arrays: An example

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

int main(){
      array<int,4> myarray = {1, 2, 3, 4};
      cout << "Element of myarray at position 1 is: "
           << myarray[1] << endl;
      return 0;
}
```

Built-in vs. Library Arrays

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

int main(){
	int myarray[3] = {10,20,30};
	for(int i=0;i<3;i++)
	    ++myarray[i];
	for(int elem:myarray)
		cout<<elem<<endl;
	return 0;
}
```

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

int main(){
	array<int,3> myarray{10, 20, 30};
	for(int i=0;i<myarray.size();i++)
	    ++myarray[i];
	for(int elem:myarray)
		cout<<elem<<endl;
	return 0;
}
```

Using arrays

- begin () - returns an iterator to the first element in the array
- end() - returns an iterator to the past-the-end element
- size () - Returns the number of elements in the array
- empty () - returns whether the array is empty
- at ()-returns a reference to the element at a specific position
- front()-returns a reference to the first element in the array
- back()-returns a reference to the last element in the array
    - fill()-sets a value for all elements in the array
    - Operator []-returns a reference to the element at a specified position in the array.

Vectors

- A collection of objects which have the same type.
- Every object has an associated index which allows access to that object.
- Efficient and flexible memory management.
- Appropriate header:

```
#include <vector>
```

Vectors

```
//vector vec holds objects of type T; vec is empty
vector<T> vec;

vector<int> vec(4); //vec holds 4 objects of type int

//4 objects of type int, each initialized to 10
vector<int> vec(4, 10);

vector<string> vec(4, "hi!");//4 strings,all initialized to "hi!"

vector<vector<int>> vec //vector whose objects are vectors
```

Vectors: An example

```
//include vector header
#include <vector>
using namespace std;

int main(){
      vector<int> vec(4);
      vec[1]=5;
      return 0;
}
```

Using vectors

- begin () - returns an iterator to the first element in the vector
- end() - returns an iterator to the past-the-end element
- size () - Returns the number of elements in the vector
- empty () - returns whether the vector is empty
- at ()-returns a reference to the element at a specific position
- insert()-inserts a new element at a specified position
- erase()-removes either a single element or a range of elements
    - push_back()-adds a new element at the end of the vector
    - pop_back()-removes the last element in the vector
    - clear()-removes all elements from the vector
    - Operator []-Oreturns a reference to the element at a specified position

Using vectors

```
#include <vector>
using namespace std;
int main(){
      vector<float> ivec(10);

      for(int i=0; i<ivec.size(); ++i)
            ivec.at(i) = 5.0f*float(i);

      return 0;
}
```

Built-in function to get the size of a vector

access element of vector ivec at position i

(Iterators in C++ STL)

Iterators

- Objects, like pointers, that point to the memory address of STL containers
- Allow iteration over a collection of elements
- Reduced complexity and execution time
- Types:
  - Input
  - Output
  - Forward
  - Bidirectional
  - Random-access

Not all iterators are supported by all the containers in STL

Why use iterators?

- Convenience in programming: Use iterators to iterate through the contents of containers.
- Reusability: Access elements of any container
- Dynamic processing of container: Dynamically add or remove elements

Iterators -- Operations

- begin (): returns the beginning position of the container
- end (): returns the after-end position of the container
- advance (): increments the iterator position till the specified number
- next (): returns the new iterator that the iterator would point after advancing the positions mentioned in the arguments
- prev ():returns the new iterator that the iterator would point after decrementing the positions mentioned in the arguments.
- inserter (): inserts the elements at any position in the container; accepts 2 arguments: 1) the container; 2) the iterator to position where the elements should be inserted.

Iterators – An example

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
        vector<int> ivec(5,20);

        for(int i=0; i<ivec.size(); ++i)
                cout << ivec.at(i) << "\n";

        return 0;
}
```

Accessing the elements of a vector

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
        vector<int> ivec(5,20);
        vector<int>::iterator it;

        for(it=ivec.begin();it<ivec.end();it++)
                cout << *it << "\n";

        return 0;
}
```

Accessing the elements of a vector using iterators

Using vectors (Con.)

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main(){
    vector<int> myvector;
    vector<int>::iterator it;

    it = myvector.begin();
    myvector.insert (it,10);
    it = myvector.end();
    myvector.insert (it,2,30);
    cout << "myvector contains:";
    for (it=myvector.begin(); it<myvector.end(); it++)
        cout << ' ' << *it;
    cout << endl;
    cout << "element at position 0 is: " << myvector.at(0) << endl;
    cout << "size of vector is:  " << myvector.size() << endl;
    myvector.erase (myvector.begin()+2);
    cout << "size of vector is:  " << myvector.size() << endl;
    myvector.push_back(50);
    cout << "myvector contains:";
    for (int i=0; i<myvector.size(); i++)
        cout << ' ' << myvector[i];
    cout << endl;
    myvector.clear();
    cout << "size of vector is:  " << myvector.size() << endl;
    return 0;
}
```

Lists

- Are implemented as doubly-linked lists; Each element is stored in different and unrelated storage locations.
- Allow constant time insertion and delete operations from anywhere in the container; iteration in both directions.
- No fast random access; Lack of direct access to the elements by their position.
- Appropriate header:

```
#include <list>
```

Lists – An example

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

int main(){
      list<int> mylist = {1, 2, 3, 4};
      for (int n : mylist)
            cout << "Elements of mylist: " << n << "\n";

      return 0;
}
```

Using Lists

- front () – acess the first element
- back () – acess the last element
- begin () - returns an iterator to the begining
- end() - returns an iterator to the end
- size () - Returns the number of elements
- empty () - checks whether the list is empty
- insert()-inserts a new element at a specified position
- erase()-removes either the element at pos. or a range of elements
    - push_back()-adds an element to the end
    - pop_back()-removes the last element
    - clear()-clears the contents

Using Lists

```
#include <iostream>
#include <list>
using namespace std;
int main(){
      list<int> mylist = {1, 2, 3, 4};
      mylist.push_front(10);
      mylist.push_back(20);
      list<int>::iterator it;
      it = find(mylist.begin(),mylist.end(), 3);
      if (it!=mylist.end())
            mylist.insert(it,30);
      for (int n : mylist)
            cout << "Elements of mylist: " << n << "\n";

      return 0;
}
```

Create a list containing integers

Add an integer at the front of the list

Add an integer at the back of the list

Insert an integer before 3 my seraching

Deque

- Deque: **d**ouble-**e**nded **que**ue
- Dynamic size; can be expanded or contracted on both ends
- deque vs. vectors: efficient insertion and deletion also at the begining -- not only at the end(vectors).
- Perform worst than lists and forward lists when frequent insertions or removals (in the middle) are required.
- Appropriate header:

```
#include <deque>
```

Using deque

- front () – acess the first element
- back () – acess the last element
- begin () - returns an iterator to the begining
- end() - returns an iterator to the end
- size () - Returns the size
- empty () - checks whether the list is empty
- insert()-inserts a new element at a specified position
- erase()-removes either the element at pos. or a range of elements
    - push_back()-adds an element to the end
    - push_front()-adds an element at beginning
    - pop_back()-removes the last element
    - pop_front()-removes first element
    - clear()-clears the contents

Deque-An example

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

int main(){
	deque<int> mydeque;
	mydeque.push_back(10);
	deque<int>::iterator it;
	for (it=mydeque.begin();it< mydeque.end();it++)
		cout << *it << "\n";

	return 0;
}
```

Exercise 1

Write a program to implement the various functions of a vector.

- A vector initially has two integer elements initialized to 50.
- Print the contents and the size of the vector.
- Insert at the beginning of the vector elements from 1 to 10 (10 9 8 7 6 5 4 3 2 1).
- Print the contents and the size of the vector.
- Remove the last element of the vector and print again the size.

Exercise 1-Solution

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main(){
    vector<int> vec(2,50);
    vector<int>::iterator it;
    cout << "vec has size " << vec.size() << " and contains: ";
    for(int i=0; i<vec.size(); i++)
        cout << ' ' << vec[i];
    cout << endl;

    it = vec.begin();
    int j=0;
    for(j=1;j<11;j++){
        it = vec.begin();
        vec.insert(it,j);
    }
    cout << "vec has size " << vec.size() << " and contains: ";
    for(int i=0; i<vec.size(); i++)
        cout << ' ' << vec[i];
    cout << endl;
    vec.pop_back();
    cout << "vec has size " << vec.size() << endl;
    return 0;
}
```

Associative Containers in the C++ standard library

Associative Containers

- Elements are stored and retrieved by a key.
- Two primary associative container types: `map` and `set`.
- The C++ library provides eight associative containers.

Associative Container Types

Container Type	
`map`	Holds key-value pairs
`set`	The key is the value
`multimap`	A key can appear multiple times
`multiset`	A key can appear multiple times
`unordered_map (c++11)`	Organized by a hash function
`unordered_set (c++11)`	Organized by a hash function
`unordered_multimap(c++11)`	Hashed map; keys can appear multiple times
`unordered_multiset(c++11)`	Hashed set; keys can appear multiple times

Ordered vs. unordered containers

- If you want guaranteed performance prefer an **ordered**.
- If you don’t have memory for a hash table prefer an **ordered** container.
- If you are using string data as a key prefer an **unordered** container.
- **map/set** containers are generally slower than **unordered_map/ unordered_set** containers to access individual elements by their *key.*
- **map/set** containers allow direct iteration on subsets based on their orders.

The map associative container

- A collection of (key, value) pairs; often referred to as an associative array.
- Values are found by a key rather than by their position (as in arrays).
- E.g.: Mapping names to phone numbers; Each pair contains a person's name as a key and a phone number as its value.

```
#include <map>
```

```
map<key, value> name;
```

The map associative container

- E.g.: `map<string,int> words;`
  - Key = Word (string)
  - Value = Word's frequency count (int)

- Several basic functions:
  - begin () - returns an iterator to the first element in the map
  - end() - returns an iterator to the theoretical element that follows the last element in the map
  - size () - Returns the number of elements in the map
    - empty () - returns whether the map is empty
    - at ()-returns the reference to the element associated with the key.

The map associative container

- Several basic functions (cont.):
  - insert(key,value)-adds a new element to the map
  - erase(iterator position)-removes the element at that position
  - erase(const g)-removes the key value ‘g’ from the map
  - clear()-removes all elements
  - find() –returns the iterator to the entry having key equal to given key.
- Operator []-used to reference the element present at position given inside the operator.
    - Operator = assigns contents of a container to a different container.

map : An example

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main (){
  map<string,string> car{{"Gabriele","Fiat"}, {"Georgia", "Audi"}};
  map<string,string> car_new;
  map<string,string>::iterator i, iter;
  for (i=car.begin();i!=car.end();i++)
        cout << "Name: " << i->first << ", car: " << i->second << endl;
    car.insert(pair<string,string>("Daniele","Renault"));
    cout  << "Name: Daniele" << ", car: " << car["Daniele"] << endl;
    iter = car.find("Georgia");
    if (iter!=car.end())
            car.erase(iter);
    cout  << "Elements in car:" << endl;
     for (i=car.begin();i!=car.end();i++)
        cout << "Name: " << i->first << ", car: " << i->second << endl;
    cout << "Size of car: " << car.size() << endl;
    car_new = car;
    cout << "Size of car new: " << car_new.size() << endl;
    car_new.at("Gabriele") = "Ford";
    for (i=car_new.begin();i!=car_new.end();i++)
        cout << "Name: " << i->first << ", car: " << i->second << endl;
     while (!car.empty()){
        cout << car.begin()->first << " => " << car.begin()->second << endl;
        car.erase(car.begin());}
  cout << "Size of car: "  << car.size() << endl;}
```

The set associative container

- It store unique elements following a specific order.
- The value of an element is its *key;* it must be unique.
- The value of the elements cannot be modified once in the container.
- The value of the elements can be either inserted or removed from the container.

```
#include <set>
```

```
set<key> name;
```

The set associative container

- begin () - returns an iterator at the beginning
- end() - returns an iterator at the end
- size () - Returns the size of the container
- empty () - returns whether the set is empty
- insert ()- inserts an element
- erase ()- erase an element
- clear()- removes all elements
- find() –returns the iterator to element
- Operator = copy the content of the container.

set : An example

```
#include <iostream>
#include <set>
using namespace std;
int main(){
	int myints[4] = {1, 2, 3, 4};
	set<int> myset(myints, myints+4);
	set<int>::iterator it;
	cout << "myset containts: ";
	for (it= myset.begin();it!= myset.end();it++)
		cout << *it << " ";
	cout << "\n";
	return 0;
}
```

Exercise 1

Write a program to print the grades of the students based on the following data (use the map container). Then you should only print the grades of students with name = Gabriele and name = Christian.

Name	Grade
Georgia	29
Gabriele	26
Chiara	30
Christian	23

Exercise 1-Solution

```
#include <iostream>
#include <map>
using namespace std;
int main(){
      map<string,int>grades{{"Georgia",29},{"Gabriele",26},
                            {"Chiara",30},{"Christian",23}};
      map<string,int>::iterator i;
      cout << "The grades of all students are: ";
      for (i=grades.begin();i!=grades.end();i++)
            cout << i->second << " ";
      cout << endl;
      cout << "The grade of student Gabriele is: "
           << grades["Gabriele"] << endl;
      cout << "The grade of student Christian is: "
           << grades["Christian"] << endl;

      return 0;
}
```

Exercise 2

Change the program of Exercise 1 as follows:

- Insert two new elements in the map
- Print the element of the map with key = "Christian"
- Print the size of the map
- Erase the element with key= "Chiara"
- Print all the elements of map
- Reprint the size of the map.

Exercise 2-Solution

```cpp
#include <iostream>
#include <map>
using namespace std;
int main(){
    map<string,int>grades{{"Georgia",29},{"Gabriele",26},
                          {"Chiara",30},{"Christian",23}};
    map<string,int>::iterator i, iter, iter1;
    cout << "The grades of all students are: ";
    for (i=grades.begin();i!=grades.end();i++)
        cout << i->second << " ";
    cout << endl;
    cout << "The grade of student Gabriele is: "
         << grades["Gabriele"] << endl;
    cout << "The grade of student Christian is: "
         << grades["Christian"] << endl;

    grades.insert(pair<string,int>("Edoardo",25));
    grades.insert(pair<string,int>("Daniele", 19));
    iter = grades.find("Christian");
    if (iter!=grades.end())
            cout << "Grade of student Christian is: " << iter->second << endl;
    cout << "Size of grades is: " << grades.size() << endl;
    iter1 = grades.find("Chiara");
    if(iter1!=grades.end())
        grades.erase(iter1);
    cout << "The grades of all students are: ";
    for (i=grades.begin();i!=grades.end();i++)
        cout << i->second << " ";
    cout << endl;
    cout << "Size of grades is: " << grades.size() << endl;
    return 0;
}
```

Additional Resources

- http://www.cplusplus.com/doc/tutorial/
- https://en.cppreference.com/w/
- Programming: Principles and Practice Using C++, Bjarne Stroustrup (Updated for C++11/C++14)
- C++ Primer, Stanley Lippman, Josée Lajoie, and Barbara E. Moo (Updated for C++11)