Introduction to C++

Marco Bernardi - Georgia Koutsandria

Internet of Things A.Y. 20-21

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

IOT Lab Classes

NS-3
NETWORK SIMULATOR

ROS

Compilers

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 commad

```
g++ -std=c++0x example.cpp -o example_program
```

Basic Input/Output

The Standard Library

- C++ uses convenient abstraction to perform input and output operations in sequential media, e.g., screen, keyboard or a file.

- **Stream:** Insert or extract characters to/from.

```
#include <iostream>
```

Standard input (cin)

- Default standarad input: keyboard
- It is used together with the extraction operator (>>) and it usually appears with the scope operator :: to indicate that cin is in the namespace std.

```
int age;
cin >> age;
```

Extracts from cin a value to be stored in the variable age

Declares a variable of type int called age

- The characters introduced using the keyboard are only transmitted to the program when the ENTER (or RETURN) key is pressed.

Standard output (cout)

- Default standarad output: screen
- It is used together with the insertion operator (<<) and it usually appears with the scope operator :: to indicate that cin is in the namespace std.

```
// prints Output sentence on screen
cout << " Output sentence";
// prints number 2 on screen
cout << 2;
// prints the value of x on screen
cout << x;
```

I/O example

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

int main(){
        int i = 0;
        cout << "Please enter an integer value:  ";
        cin >> i;
        cout << "The value you entered is " <<  i ;
        cout << " and its double is "  <<  i*2 << ".\n " ;
        return 0;
}
```

Statements and Flow Control

Conditional Statements

1. *if* statement: Tests a condition or a set of conditions. sequentially.

```
if (condition)
    statement
```

2. *switch* statement: Evaluates an integral expression and chooses one of several execution paths based on the expression's value.

```
switch (condition)
    statement
```

Condition(s)

- The *Condition* must be enclosed in parenthesis
- It can be an expression or an initialized variable declaration. It must have a type that is convertible to bool.

```
if (condition)
    statement
```

True → The statement is executed.

False → The statement is skipped.

The *if* Conditional Statement

- Conditionally executes another statement based on whether a specified condition is true.

simple if

```
int number=0;
cout << "Enter an integer: ";
cin >> number;
// checks if the number is positive
if ( number > 0) {
        cout << "You entered a positive integer: " << number << endl;
}
```

Condition

The *switch* Conditional Statement

- A convenient way of selecting among a (possible large) number of fixed alternatives.

```
switch(x){
    case 1:
            cout << "x is 1";
            break;
    case 2:
            cout << "x is 2";
            break;
    default:
            cout << "value of x is unknown";

}
```

Iterative Statements (loops)

- Repeated execution until a condition is true
- Statements that test the condition before executing the block: while, for
- Statement that executes the body and then tests the condition: do while

```
while (condition)
      statement
```

```
for (initializer; condition; expression)
      statement
```

```
do
            statement
while       (condition);
```

The for loop

- It repeats *statement* while *condition* is true.

```
#include <iostream>
using namespace std;
int main(){
        for (int n=0; n<5; n++)
           cout << n << " ";
        cout << endl;
}
```

The while loop

- It simply repeats *statement* while *condition* is true.
- The loop ends if, after any execution of *statement*, *expression* is no longer true.

```
#include <iostream>
using namespace std;
int main(){
        int n = 10;
        while (n>0){
          cout << n << ", ";
          --n;
        }
        cout << "liftoff!\n";
}
```

The do-while loop

- It behaves like the *while* loop, except that *condition* is evaluated after the execution of the *statement.*

```
#include <iostream>
using namespace std;
int main(){
        string str;
        do {
          cout << "Enter text: ";
          getline(cin,str);
          cout << "You entered: " << str << "\n";
        } while(str!="ciao");
}
```

Functions

Functions Basics

- A *function* is a block of code with a name.
- It is executed by calling the given name, and it can be called from some point of the program.
- Common syntax:

```
type name(parameter1, parameter2, …){statements}
```

Functions: An example

```
//function example
#include <iostream>
using namespace std;
int addition(int a, int b){
        int r;
        r=a+b;
        return r;
}
int main(){
        int z;
        z = addition(5,3);
        cout << "The result is " << z << ".\n ";

}
```

Calling a Function

- A *function* call
  - Initializes the parameters from the arguments
  - Transfers control to that *function.*
- Execution of the called *function* begins.

```
int main(){
        int z;
        z = addition(5,3);
        cout << "The result is " << z << ".\n ";

}
```

Functions with no type

- When a function does not need to return a value, the type to be used is void.

- This is a special type to represent the absence of value.

- The void can also be used in the function’s parameter list to specify that the function takes no actual parameters when called.

```
printmessage();
```

Note the use of the empty pair of parentheses!

Declaring functions

- Functions cannot be called before they are declared.
- Functions should be declared before calling main.
- If the main is defined before an undeclared function is called, then the compilation of the program will fail.

```
int fact(int a, int b);
void even(int x);
```

Function declaration

Passing arguments by value

- Arguments can be passed by value

```
int x=5, y=3, z;
z = addition (x,y);
```

- Only copies of the variables values at that moment are passed to the function.

- Modifications on the values of the variables have not effect on the values of the variables outside the function.

Passing arguments by reference

- Access an external variable from within a function.
- The variable itself is passed to the function.
- Any modification on the local variables within the function are reflected in the variables passed as arguments in the call.
- References are indicated with an ampersand (&) following the parameter type.

Passing arguments by reference

```
//passing parameters by reference
#include <iostream>
using namespace std;
void duplicate(int& a, int& b){
        a*=2;
        b*=3;
}
int main(){
        int x=1, y=3;
        duplicate(x, y);
        cout << "x=" << x << ", y=" << y << "\n";
        return 0;
}
```

Exercise 1

Write a program that prompts the user to give two integer numbers. Declare a function that compares the two integers and returns the maximum one.

Exercise 1-Solution

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

int returnMax(int a, int b);
int main(){

    int num1, num2, maximum;

    cout << "Enter two numbers:";
    cin >> num1 >> num2;
    maximum = returnMax(num1,num2);
    cout << "Max of " << num1 << " and " << num2 << " is: " << maximum <<"\n";

    return 0;
}

int returnMax(int a, int b){
    int maximumNumber;
    if ( a > b )
        maximumNumber = a;
    else
        maximumNumber = b;
    return maximumNumber;
}
```

Arrays

Arrays

- A structure which stores many variables of the same type, e.g., int, double, bool, etc..

- Uses an 'index' to access each variable or 'element' of the array.

- C++ includes static arrays, allocated arrays, and standard library containers.

Array declaration

- Arrays are declared like normal variables.
- Square brackets [] indicate the number of variables which the array can store.

```
#include <iostream>

int main()
{
        int array[5];
        double array_d[2]={1.0,2.0};

        return 0;
}
```

array can store five int variables

array_d can store 2 double variables, initialized

Accessing array elements

- Array ‘index’ starts from zero
- Can be used for arithmetic, copied to other variables etc..

```
#include <iostream>

int main()
{
        int array[5];
        array[0] = 3;
        array[1] = array[0]+5;
        return 0;
}
```

array can store five int variables

set the first element to 3

set the second element to 8

Accessing Arrays using loops

```
#include <iostream>

int main()
{
        int array[5];

        for(int i=0; i<5; ++i)
          array[i] = 0;

        for(int i=0; i<5; ++i)
          array[i] += i;

        return 0;
}
```

loop over all elements and set to zero

Add i to each element of the array

Pointers

Pointers

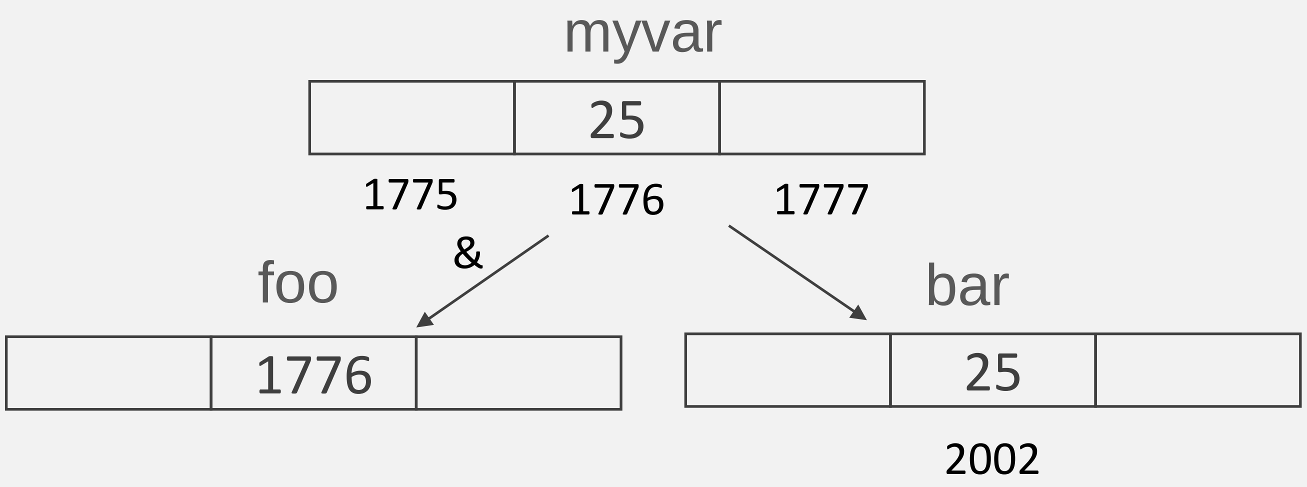
- Variables: Locations in the computer's memory which can be accessed by their identifier (their name).

- 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.

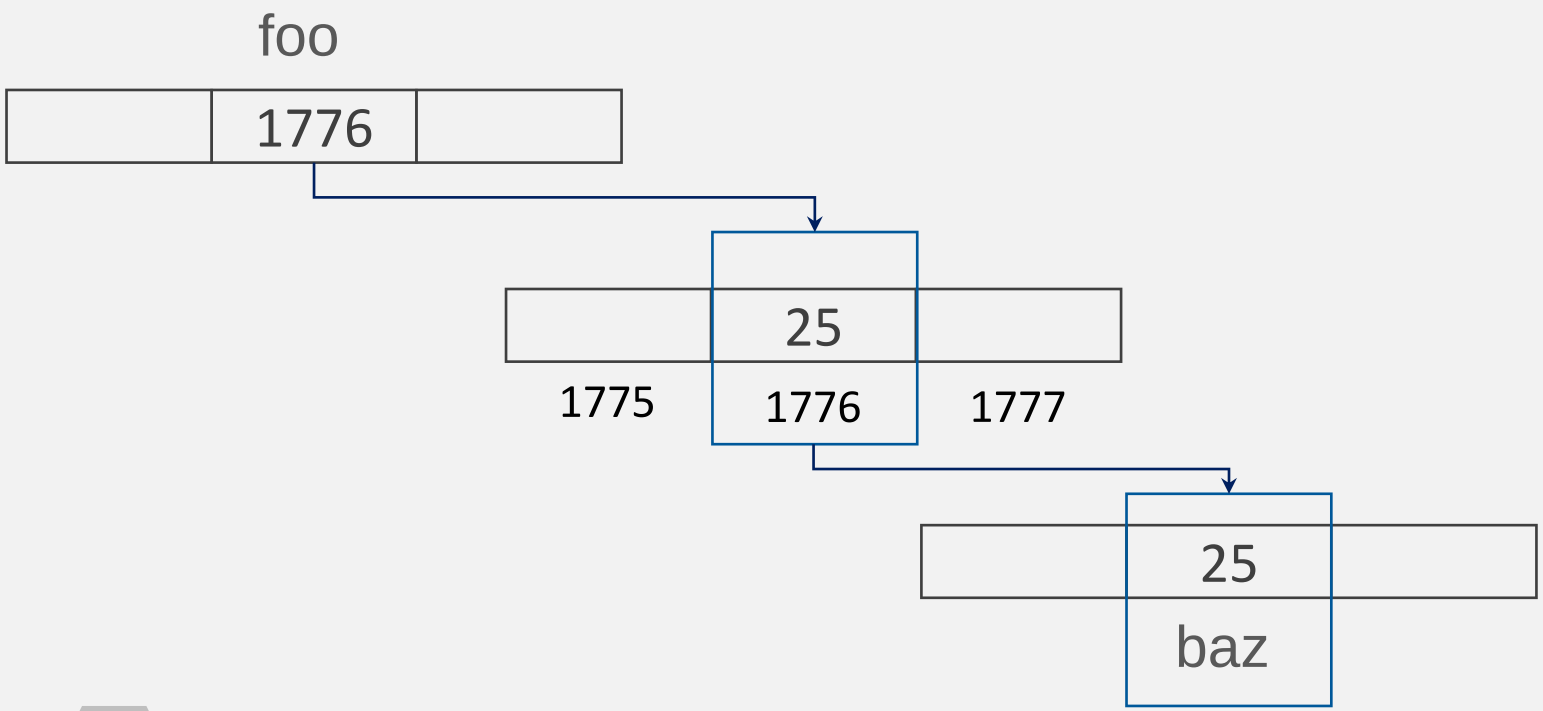
Pointers

- **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 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.

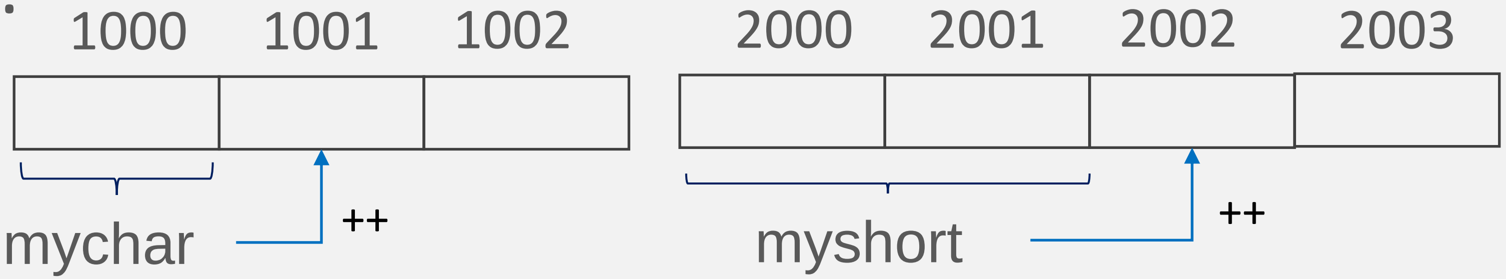
Pointers- An example

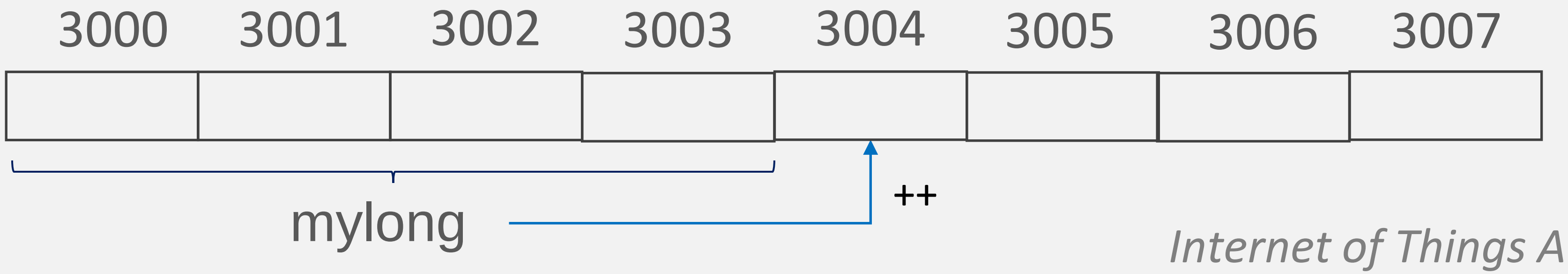
```
#include <iostream>
using namespace std;
int main(){
        int firstvalue = 0;

        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

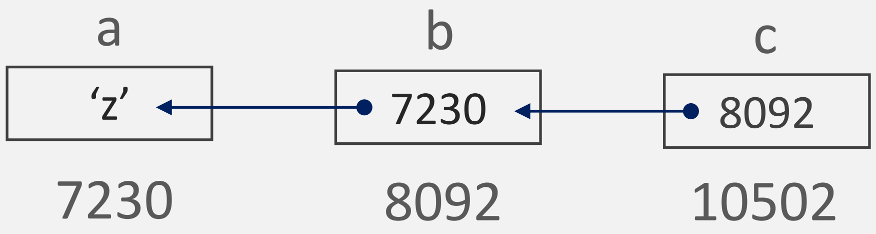
- The increment/decrement operators can be used as either prefix or suffix of an expression.
- The increment/decrement operator has a higher precedence than the *.

```
//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 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 numbers[5];
		int * p;
		p = numbers; *p = 10;
		p++; *p = 20;
		p = &numbers[2]; *p = 30;
		p = numbers + 3; *p = 40;
		p--; *p = 100;
		p = numbers; *(p+4) = 50;
		cout << "Output: " << endl;
		for (int n=0;n<5;n++)
				cout << numbers[n] << "\n";
		return 0;
}
```

Exercise 1--Solution

```
#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--; *p = 100;
        p = numbers; *(p+4) = 50;
        cout << "Output: " << endl;
        for (int n=0;n<5;n++)
                cout << numbers[n] << "\n";
        return 0;
}
```

Output:
10
20
100
40
50

Exercise 2

What is the exact output of the following program?

```
#include <iostream>
using namespace std;
int main(){
	int array[3]={3, 6, 9};
	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=0;k<3;k++)
		cout << *p+k+2 << endl;

	return 0;
}
```

Exercise 2--Solution

```
#include <iostream>
using namespace std;
int main(){
        int array[3]={3, 6, 9};
        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=0;k<3;k++)
                cout << *p+k+2 << endl;

        return 0;
}
```

```
Print a:
5
8
11
Print b:
5
6
7
```

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.

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;
}
```

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>
```

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

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;
}
```

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;
```

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;}
```

Associative containers

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;
```

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;
}
```

Range-based Loop

Range-based loop

- A more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container (array, vector, map, set, etc.).
- For observing elements in a container. i.e., read-only:
  1. If the objects are cheap to copy (capture by value)

     ```
     for (auto elem : container_name)
     ```

  2. Capture by const reference

     ```
     for (const auto& elem : container_name)
     ```

  - When modifying the elements in the container:
    - Capture by non-const reference

      ```
      for (auto& elem : container_name)
      ```

Functions (Cont.)

Passing arguments to a function

- Arguments can be passed *by value*; Only copies of the variables values at that moment are passed to thefunction; Modifications on the values of the variables have not effect on the values of the variables outside the function.

- Arguments can be passed *by reference*; The variable itself is passed to the function; Any modifications on the local variables within the function are reflected in the variables passed as arguments in the call

Passing arguments to a function (Cont.)

- Passing arguments *by const reference*. **Why**?
- Passing *by value* requires that all arguments are copied into the function parameters.->time consuming when handling large structs, classes, etc.
    - *Solution*: arguments are passed *by reference.*
    - *Problem:* Undesirable when we want read-only arguments.
        - *(More appropriate) solution:* pass *by const reference*
            - *Minimum performance penalty (not copying arguments)*
            - *Function cannot change the value of the arguments.*

Passing arguments by const reference: An (wrong)example

```
//passing parameters by const reference
#include <iostream>
using namespace std;
void foo(const int &a){
        a = 2;
}
```

!

Compiler will complain! A const reference cannot have its value changed!

Passing arguments to a function

Q: Can we pass an entire array as an argument to a function?

Passing arguments to a function

Q: Can we pass an entire array as an argument to a function?

A: Not directly but «indirectly!»

Passing arguments to a function

- While an entire array cannot be passed as an argument to a function, pointers to an array can.
- There are different ways to do so:

1. Formal parameter as a pointer:

```
void function_name(type *param){}
```

1. Formal parameter as a sized array:

```
void function_name(type param[n]){}
```

1. Formal parameter as an unsized array:

```
void function_name(type param[]){}
```

Passing arguments to a function
An example (Case 1)

```
#include <iostream>
using namespace std;
double calcAverage(int *arr, int size);
int main(){
        int numbers[5] = {2, 4, 6, 8};
        double avg;
        // int * p = numbers;
        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;

            }
```

Functions

Data Structures

Data Structures

- A group of data elements of different kinds grouped together under a single name.
- Data elements (*members)* can be of different types and lengths.

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

Defining data structures

- Keyword "struct" is used to create the structure.
- type_name: The name of the structure type.
- member_name: The name of the data member.
- object_names: A set of valid identifiers for objects that have the type of this structure.

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

Defining data structures: An example (Alternative option)

structure type name

```
struct product{
    int weight;
    double price;
};

product apple;
product melon,orange;
```

Objects of type product

structure type name

```
struct product{
    int weight;
    double price;
} apple, melon, orange;
```

Name objects can be used to directly declare objects of the structure type.

Accessing the members

- Once a member is declared, it can be accessed directly.
- Syntax: Insert a dot (.) between the object name and the member name.
- E.g.: Each of the objects has a data type corresponds to the member it refers to.
    - apple.weight
    - apple.price
    - melon.weight
    - melon.price
    - orange.weight
    - orange.price

**.weight* are of type int

**.price* are of type double

Initializing structure members

- Structure members can be initialized using curly braces, i.e., {}.

```
#include <iostream>
using namespace std;
struct point{
    int x, y;
};
int main (){
            point p1 = {0,1};
            cout << "Printing x coordinate of p1: " << p1.x << "\n";
            cout << "Printing y coordinate of p1: " << p1.y << "\n";
            return 0;


}
```

Initializing structure members

- Structure members can be initialized using curly braces, i.e., {}, or with declaration.

```
#include <iostream>
using namespace std;
struct point{
    int x = 0;
    int y = 1;
};
int main (){
        point p1;
        cout << "Printing x coordinate of p1: " << p1.x << "\n";
        cout << "Printing y coordinate of p1: " << p1.y << "\n";
        return 0;

}
```

Array of structures

- We can create an array of structures. Each array will have the same structure members.

```
#include <iostream>
using namespace std;
struct student{
    int studentId;
    string firstName, lastName;
};
int main (){
        student stud[2];
        for(int i=0;i<2;i++){
                cout << "Enter the id of the student:";
                cin >> stud[i].studentId;
                cout << "Enter the first name of the student :";
        cin >> stud[i].firstName;
                cout << "Enter the last name of the student :" << endl;
        cin >> stud[i]. lastName;}
        return 0;
}
```

Data structures and functions

- Structure elements can be passed to a function as normal agruments.
  1. by value
     - The values of the elements are passed to the function.
     - The entire structure can be passed to a function.
  2. by reference
     - The address of the structure element is passed to the function.
- Structure elements can be returned from a function as normal arguments.

Data structures and functions

```
struct product{
    int weight;
    double price;
} apple;
```

Individual elements are passed in a function

```
void func1(apple.weight, apple.price){}
```

Data structures and functions

- The entire structure can be passed to a function by value.
- Any changes to the contents of the structure inside the function, do not affect the structure itself.

```
struct product{
    int weight;
    double price;
} apple;
```

Entire structue is passed to a function

```
void func1(product fruit){}
```

Data structures and functions: An example

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
struct movies_t{
  int year;
  string title;
}mine, yours;

void printmovie (movies_t movie){
            cout << movie.title;
            cout << " (" << movie.year << ")"
          << endl;
}
```

```
int main(){
  string mystr;
  mine.title = "Goodbye Bafana";
  mine.year = 2007;
  cout << "Enter a title: ";
  getline(cin, yours.title);
  cout << "Enter year: ";
  getline(cin, mystr);
  stringstream(mystr) >> yours.year;
  cout << "My favorite movie is: ";
  printmovie(mine);
  cout << "Your favorite movie is: ";
  printmovie(yours);

  return 0;
}
```

Pointers to Structures

Pointers to Structures

- A structure can be pointed to by its own type of pointers.

```
struct movies_t{
  int year;
  string title;
};

movies_t amovie;
movies_t  * pmovie;
pmovie = &amovie;
```

An object of structure type movies_t

A pointer that points to objects of structure type movies_t

The value of the pointer pmovie is assigned the address of object amovie.

Pointers to Structures

- The arrow operator (->) is a dereference operator that is used exclusiveley with pointers to objects that have members; It allows access to the member of an object directly from its address.

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed to by a	(*a).b
*a.b	Value pointed to by member b of object a	*(a.b)

Pointers to Structures: An example

```
#include <iostream>
using namespace std;
struct movies_t{
  int year;
  string title;
}mine;
void printmovie (movies_t *movie){
              cout << movie->title;
              cout << " (" << movie->year << ")" << endl;
}
int main(){
  mine.title = "Goodbye Bafana";
  mine.year = 2007;
  cout << "My favorite movie is: ";
  printmovie(&mine);
  return 0;
}
```

Nesting Structures

Nesting Structures

- Structures can be nested in such a way that an element of a structure is itself another structure.

```
struct movies_t{
  int year;
  string title;
};

struct friends_t{
  int year;
  string name;
  string email;
  movies_t favorite_movie;

}gina, gabriele;

friends_t * pfriends = &gina;
```

```
gina.name
gabriele.favorite_movie.title
gina.favorite_movie.year
pfriends->favorite_movie.year
```

Classes

Classes

- class_name: A valid identifier for the class.
- object_names: An optional list of names for objects;
 An object is an instantiation of a class.
- members: Contained in the body of the declaration; can be data or function declarations.
- access_specifiers: Modify the access rights for the members of the class (optional).

```
class class_name{
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  . . .
}object_names;
```

Access specifier

- **Private**: Accessible only from within other members of the same class (default).
- **Protected**: Accessible from other members of the same class and also from members of their derived classes.
- **Public**: Accessible from anywherhe where the object is visible.

Classes: An example

Class declaration

Name of class

```
class Rectangle{
   int width, height;
  public:
   void set_values(int,int);
   int area(void);
}rect;
```

Class contains four members

Two data members of type int; private access

Two member functions; public access. Only the declaration is included.

An object, i.e., a variable, of the class

Class vs. Object name

- Rectangle: The class name
- rect: An object of type Rectangle
- Analogy: int a;

The type name (the class) — The variable name (the object)

```
class Rectangle{
int width, height;
  public:
    void set_values(int,int);
    int area(void);
}rect;
```

Accessing public members of a class

- Public objects can be accessed as if they were normal functions or variables.

- Use of dot (.) between object name and member name.

- E.g.:
```
rect.set_values(3,4);
myarea = rect.area();
```

```
class Rectangle{
int width, height;
  public:
    void set_values(int,int);
    int area(void);
}rect;
```

Accessing members of a class

```
class Rectangle{
    int width, height;
  public:
    void set_values(int,int);
    int area(void);
}rect;
```

!

Members with private access cannot be accessed from outside of the class.
They can only be referred to from within
other members of the same class.

Defining a member function

1. Within the class definition: Function is automatically considered an inline member function by the compiler.

1. Include declaration and define it later outside the class: A normal (not-inline) class member function.

An example

The scope operator (::) is used in the definition of a class member to define a member of class outside the class itself.

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    void set_values(int,int);
    int area(){return width*height;}
};
void Rectangle::set_values(int x, int y){
              width = x;
              height = y;
}
int main(){
  Rectangle rect;
  rect.set_values(3,4);
  cout << "area: " << rect.area() << endl;
  return 0;
}
```

Multiple object declaration

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    void set_values(int,int);
    int area(){return width*height;}
};
void Rectangle::set_values(int x, int y){
              width = x;
              height = y;}
int main(){
  Rectangle rect, rectb;
  rect.set_values(3,4);
  rectb.set_values(5,6);
  cout << "area: " << rect.area() << endl;
  cout << "areab: " << rectb.area() << endl;
  return 0;
}
```

Two instances (objects)

Q: What would happen in the previous example if we called the member function `area` before having called `set_values`?

Q: What would happen in the previous example if we called the member function `area` before having called `set_values`?

A: An undetermined result, since the members `width` and `height` had never been assigned a value.

Q: What would happen in the previous example if we called the member function area before having called set_values?

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    void set_values(int,int);
    int area(){return width*height;}
};
void Rectangle::set_values(int x, int y){
              width = x;
              height = y;}
int main(){
  Rectangle rect, rectb;
  cout << "area: " << rect.area() << endl;
  rect.set_values(3,4);
  rectb.set_values(5,6);
  cout << "areab: " << rectb.area() << endl;
  return 0;
}
```

Q: What would happen in the previous example if we called the member function area before having called set_values?

A: An undetermined result, since the members width and height had never been assigned a value.

```
Georgias-MacBook-Pro:C++ examples gina$ g++ -std=c++11  rectangleError.cpp -o rectangleError
Georgias-MacBook-Pro:C++ examples gina$ ./rectangleError
area: 1718552992
areab: 30
```

Constructor

- A special member function of a class which is automatically called whenever a new object of a class is created.
- It allows the class to initialize member variables or allocate storage.
- They are only executed once, when a new object is created.
- Declaration: like a regular member function; the name matches the class name; no return type (they initialize an object)

Constructor - An example

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    Rectangle(int, int);
    int area(){return width*height; }
};
Rectangle::Rectangle(int a, int b){
          width = a;
          height = b;
}
int main(){
  Rectangle rect(3,4);
  Rectangle rect_b(5,6);
  cout << " rect area: " << rect.area() << endl;
  cout << " rect_b area: " << rect_b.area() << endl;
  return 0;
}
```

→ Constructor prototype declaration

Constructor definition

Overloading constructors

Overloading constructors

- A constructor can be overloaded with different versions taking different parameters.
- The compiler will automatically call the one whose parameters match the arguments.
- The *default constructor*: A special kind constructor that takes no parameters. It is called when an object is declared but is not initialized with any arguments.

```
Rectangle rectb; // ok, default constructor called
Rectangle rectc(); // Oops!
```

Constructors: An example

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    Rectangle();
    Rectangle(int, int);
    int area(){return width*height; }
};
Rectangle::Rectangle(){
                width = 5;
                height = 5;
}
Rectangle::Rectangle(int a, int b){
                width = a;
                height = b;
}
```

```
int main(){
  Rectangle rect(3,4);
  Rectangle rect_b;
  cout << "rect area:" << rect.area()
              << endl;
  cout << " ect_b area: " << rect_b.area()
              << endl;
  return 0;
}
```

Calling constructors

Calling constructors

- *functional form*: Enclose the arguments of the constructor in parentheses.

  class_name object_name (value1, value2, value3, …)

- *Single parameter:*

  class_name object_name = initialization_value;

- *Uniform initialization:* Same as the functional form but using braces instead of parentheses. (Optional: an equal sign before the braces.)

  class_name object_name { value1, value2, value3, … }

  class_name object_name = { value1, value2, value3, … }

An example

```
#include <iostream>
using namespace std;
class Circle{
    double radius;
  public:
    Circle(double r){radius = r;};
    double circum(){return 2*radius*3.14159265; }
};

int main(){
  Circle foo(10.0); //functional form
  Circle bar = 20.00; // assignment init.
  Circle baz {30.00}; // uniform init.
  Circle qux = {40.00}; //uniform init.

  return 0;
}
```

Constructors: Initialization

- It is mainly a matter of programming style!

- Uniform vs. functional: Braces cannot be confused with function delcarations.

```
Rectangle rectb; // default constructor called
Rectangle rectc(); // function declaration
Rectangle rectd{}; // default constructor called
```

Member initialization in constructors

Member initialization

- When a constructor is used to initialize other members, these members can be initialized directly.
- Initialization is done by inserting, before the contructor’s body, a colon (:) and a list of initializations for class members.

```
class Rectangle{
int width, height;
  public:
    Rectangle(int, int);
    int area(){return width*height; }
};
```

---

1. `Rectangle::Rectangle(int a, int b){ width = a; height = b; }`
2. `Rectangle::Rectangle(int a, int b) : width(a) { height = b; }`
3. `Rectangle::Rectangle(int a, int b) : width(a), height(b) { }`

Member initialization

```
class Rectangle{
int width, height;
  public:
    Rectangle(int, int);
    int area(){return width*height; }
};
```

---

1. `Rectangle::Rectangle(int a, int b){ width = a; height = b; }`

Classic constructor definition

2. `Rectangle::Rectangle(int a, int b) : width(a) { height = b; }`
3. `Rectangle::Rectangle(int a, int b) : width(a), height(b) { }`

Constructor definition with member initialization

Destructor

- A member function of a class that deletes an object
- It helps deallocate the memory of an object
- It does not take any arguments and does not return anything
- There cannot be more than one destructor in a class
- Syntax: ~className
- The compiler creates a default desctructor
  - Problem: Dynamically allocated memory or pointer in a class.
    - Solution: Write a destructor to release memory and avoid memory leak (using delete object).

Pointers to classes

Pointers to classes

- Objects can be pointed to by pointers.
- The members of an object can be accessed directly from a pointer by using the arrow operator(->).
- Syntax:

```
class_name * pointer_name;
```

Operators

Expression	
*x	Pointed to by x
&x	Address of x
x.y	Member y of object x
x->y	Member y of object pointed to by x
(*x).y	Member y of object pointed to by x
x[0]	First object pointed to by x
x[0]	Second object pointed to by x
x[n]	(n+1)th object pointed to by x

Pointers to classes: An example

```
#include <iostream>
using namespace std;
class Rectangle{
int width, height;
  public:
    Rectangle(int x, int y): width(x), height(y){};
    int area(void){return width*height; }
};
int main(){
  Rectangle rect(3,4);
  Rectangle * foo, * bar, * baz;
  foo = &rect;
  bar = new Rectangle (5,6);
  baz = new Rectangle[2]{{2,5},{3,6}};
  cout << " rect’s area: " << rect.area() << endl;
  cout << " *foo’s area: " << foo->area() << endl;
  cout << " *bar’s area: " << bar->area() << endl;
  cout << " baz[0] area: " << baz[0].area() << endl;
  cout << " baz[1] area: " << baz[1].area() << endl;
  delete bar;
  delete[] baz;
  return 0;
}
```

Classes- Alternative definitions

- Classes can be defined also with keywords *struct* and *union*.
- Keyword *struct:* Plain data structures; public access by default.
- Keyword *union:* Store only one data member at a time; public access by default.

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)