POLITECNICO DI MILANO

HEAP LAB





Moving to C++

Advanced Operating Systems

Giuseppe Massari
giuseppe.massari@polimi.it

Outline

C++ Programs

- Memory allocation
- Passing parameters to functions

Classes

- Constructors and assignments
- Object pointers
- Qualifiers
- Inheritance and polymorphism

Standard Template Library (STL)

- Templates
- Example: `vector`, `list`, `map`

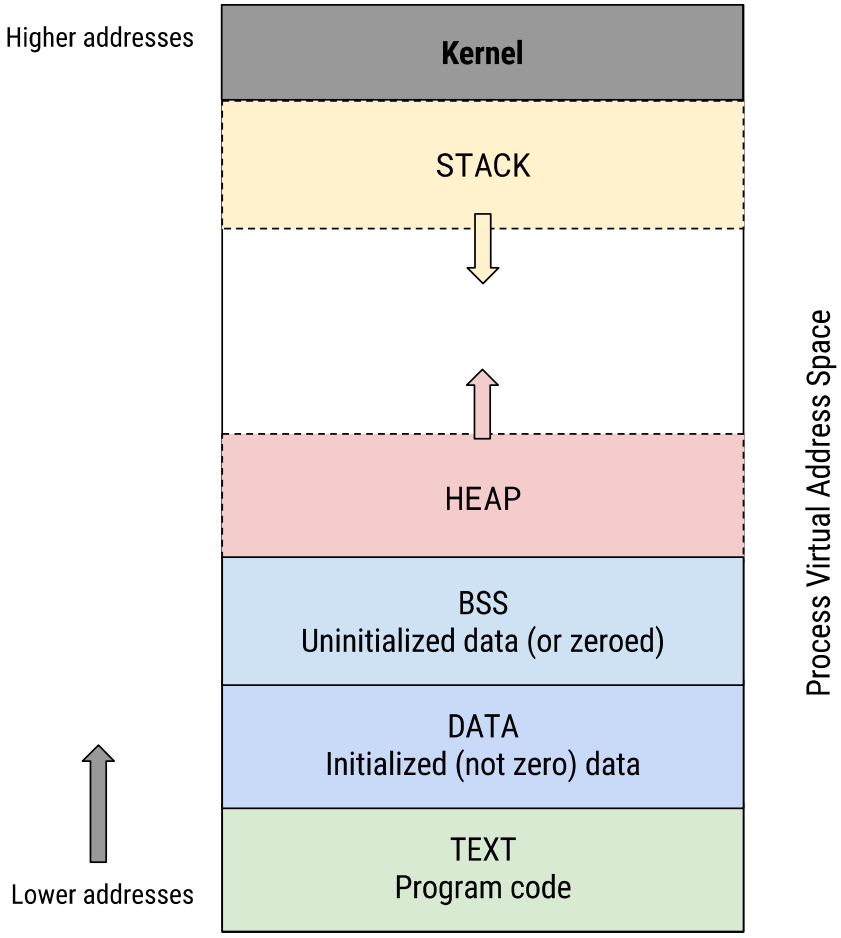
Memory management in C++11

- From raw to smart pointers

Program memory

Memory layout

- Once loaded in the main memory a program is typically divided into segments
  - **TEXT** : Program instructions
  - **DATA** : Initialized variables
  - **BSS** : Uninitialized or zeroed variables
  - **STACK** : Local variables and return address of functions
  - **HEAP** : *Dynamically* allocated data, i.e., allocated/de-allocated at run-time



Memory allocation

- In C programs dynamic memory allocation and de-allocation is performed through **`malloc()`** and **`free()`** functions

```
char * buff = malloc(buff_size);
```

```
free(buff);
```

- In C++ programs operators **`new`** and **`delete`** are used
  - For dynamic array allocation **`new ….[]`** / **`delete[]`**

```
char * buff = new char[5]

Object * obj = new Object();
```

```
delete[] buff;

delete obj;
```

- **`new`**/**`delete`** call class constructor and destructor respectively
  - Do not mix **`malloc`**/**`free`** with **`new`**/**`delete`** usage!

    Do not use **`malloc`**/**`free`** with C++ objects since they do not call the class constructor/destructor!

Memory allocation

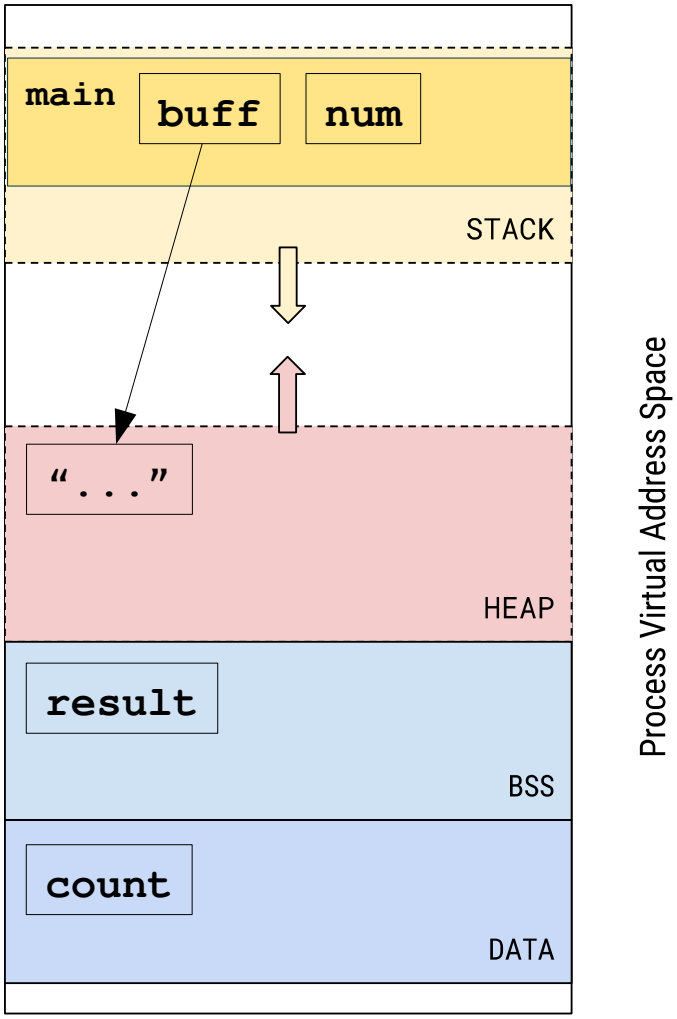
- Example

```
static result;
unsigned int count = 20;

int main() {

    int num;
    char * buff = new char[4];

    …
    delete[] value;
    return 0;
}
```



Memory allocation common errors

Memory leak

- Memory allocated but NOT released
- Application continuously increase memory consumption
  - Reducing the availability for other applications
- Performance slow down
  - Operating system may re-map the virtual memory page to the hard drive
- Application or even the system may stop working
  - Unpredictable behaviour when system memory resource limits are reached

Sources

- We forgotten the related `delete` call for some `new`
- We changed the value of a pointer variable
  - No other variables point to the previously dynamically allocated data
  - Such data becomes unreachable!

Memory allocation common errors

Memory corruption

- Memory location alteration without an explicit assignment
  - Attempt of freeing an already freed memory location (dangling pointer)

```
int main() {
    char * buff = new char[4];
    delete[] buff;
    …
    delete[] buff;
}
```

```
$ ./program

*** Error in `./program':
 double free or corruption
: 0x00000000011a5040 ***

Aborted (core dumped)
```

  - Attempt of accessing a freed (or not allocated yet) memory location

```
int main() {
    char * buff;
    cout << buff << endl;
}
```

```
$ ./program

Segmentation fault (core dumped)
```

Passing parameters

- By *value*
  - Passed variables/objects are COPIED into the called function stack

```
void func(Object var);
```

```
func(var);   // call
```

- By *address* (C approach)
  - Caller passes the address of the object/var to the callee (NO COPY)
  - Callee can modify the content of the object/variable

```
void func(Object * var);
```

```
func(&var);   // call
```

- By *reference* (C++ approach)
  - Same effect of passing by address (NO COPY)
  - Callee can modify the content of the variable/object

```
void func(Object & var);
```

```
func(var);   // call
```

Outline

Syntax

- A C++ class includes different sections to set the visibility of member *data* (attributes) and member *functions* (methods)
  - `public:` Publicly accessible
  - `private:` Only other class members can access
  - `protected:` Only other class members can access - Derived class will inherit base class protected members

```
class People {
public:
...
protected:
...
private:
...
};
```

Syntax

- Common practice:
  - Definitions in the header files (`.h`, `.hh`) / Implementation in source files (`.cc`, `.cpp`)
  - This unless they are “template” or “inline” functions

```
// people.h

class People {
public:
...
    void Talk();
};
```

```
// people.cpp

void People::Talk {

}
```

Classes

Syntax

```
#ifndef PEOPLE_H_
#define PEOPLE_H_

#include <string>

class People {
public:
    People();
    int Walk();
    void Talk() const;

private:
    std::string name;
    int step_count;
};

#endif // PEOPLE_H_

// people.h
```

```
#include <iostream>
#include "people.h"

People::People():
    name("guy"),
    step_count(0) {
}

int People::Walk() {
    return ++step_count;
}

void People::Talk() const {
    std::cout << name
        << " made " << step_count
        << " steps" <<  std::endl;
}

// people.cpp
```

Constructors

- A function having the same class name but not returning any type
- Called every time an instance of a class is created
- If not provided, the compiler will generate it
- A class can have multiple constructors, with different argument lists

```
class People {
public:
    People();
    People(std::string _name);
    …
private:
    std::string name;
    int step_count;
…
};

// people.h
```

```
People::People():
    name("Guy"),
    step_count(0) {
}

People::People(std::string _name):
    name(_name),
    step_count(0) {
}

// people.cpp
```

Constructors

- Called when we instantiate an object, by declaring a variable of the class type, or we dynamically allocate an object of the class type (see later)

```
int main() {
...
    People who1;
    People who2();
    People simon("Simon");


    who1.Talk();
    who2.Talk();
    simon.Talk();
...
}
```

People() version is called

People(std::string) version is called

Output:

```
Guy made 0 steps
Guy made 0 steps
Simon made 0 steps
```

Destructors

- Called when the object is going out of scope or it is explicitly deleted
- Syntax: `~ClassName()` (no arguments)
  - Very often qualified as “virtual” to prevent derived class from invoking the base class destructor
- Commonly includes the implementation of end of life actions, e.g., clean-up code, memory deallocation,...

```
class People {
public:
    People();
    People(std::string _name);
    virtual ~People();
…
};
// people.h
```

```
…
People::~People() {
    std::cout << name
        << ": i'm dying..."
        << std::endl;
}
…
// people.cpp
```

Copy constructors

- Special constructors having an object of the same class as argument
- If not implemented the compiler will automatically generate one
- The default implementation is a *member-wise copy* of the object

```
People::People( const People & other )
{
    name       = other.name;
    step_count = other.step_count;
}
```

- Member-wise copy can be expensive
- Depending on the class definition, it may also lead to undesired behaviours
  - What if member data are pointers? Pointer object “complex” class types?

Copy constructors

- Example
  - Fred is created starting from Simon
    - ➔ Do we want Fred to have the same Simon's information?
  - Default copy constructor is called (member-wise copy)

```
int main() {
...
    People simon("Simon");

    for(int i=0; i<3; ++i)
        simon.Walk()
    simon.Talk();

    People fred(simon);
    fred.Talk();
...
}
```

Output:

```
Simon made 3 steps
Simon made 3 steps
```

fred's member data have been initialized to the same value of object **simon**

Copy constructors

- Creating a “People” object starting from another one...
  - Do we want the “new person” Fred to start with the number of steps of the Simon?
- Overload of the copy constructor implementing our custom version
  - Set `step_count` to zero, since we are constructing a new object that has never walked before!
  - Desired behaviour: *Just copy the name of the other person*

```
People::People( const People & other )
{
    name       = other.name;
    step_count = 0;
}
```

Copy constructors

- Example
  - We create a second “Simon” object
  - Custom copy constructor initializes the object using the same of the first Simon

```
int main() {
...
    People simon1(“Simon”);

    for(int i=0; i<3; ++i)
        simon.Walk()
    simon.Talk();

    People simon2(simon1);
    simon2.Talk();
...
}
```

Output:

```
Simon made 3 steps
Simon made 0 steps
```

simon2 has the same name of **simon1**, but since it has just born, it has never walked before!

Object assignment

- Assignment operation is provided by the compiler
- The default implementation is the member-wise copy of the object
  - Conversely from Java the object variables content is a *value* and not a reference

```
int main() {
...
    People simon("Simon");
    People fred;

    fred = simon;
    simon.Walk();
    simon.Talk();

    fred.Talk()
...
}
```

Output:

```
Simon made 1 steps
Simon made 0 steps
```

Member data **name** value has been copied

Walk() has updated only **simon** status

Object assignment

- We can overload the assignment operator (“=”) too in order to modify the default behaviour
- Assignment operator (“=”) overloading
  - Syntax similar to member function implementation, but replacing the function name with **operator**<operator_symbol>

```
class People {
public:
…
    People & operator=(const People & other) {
        name       = other.name;
        step_count = other.step_count;
    }
};
```

Object pointers

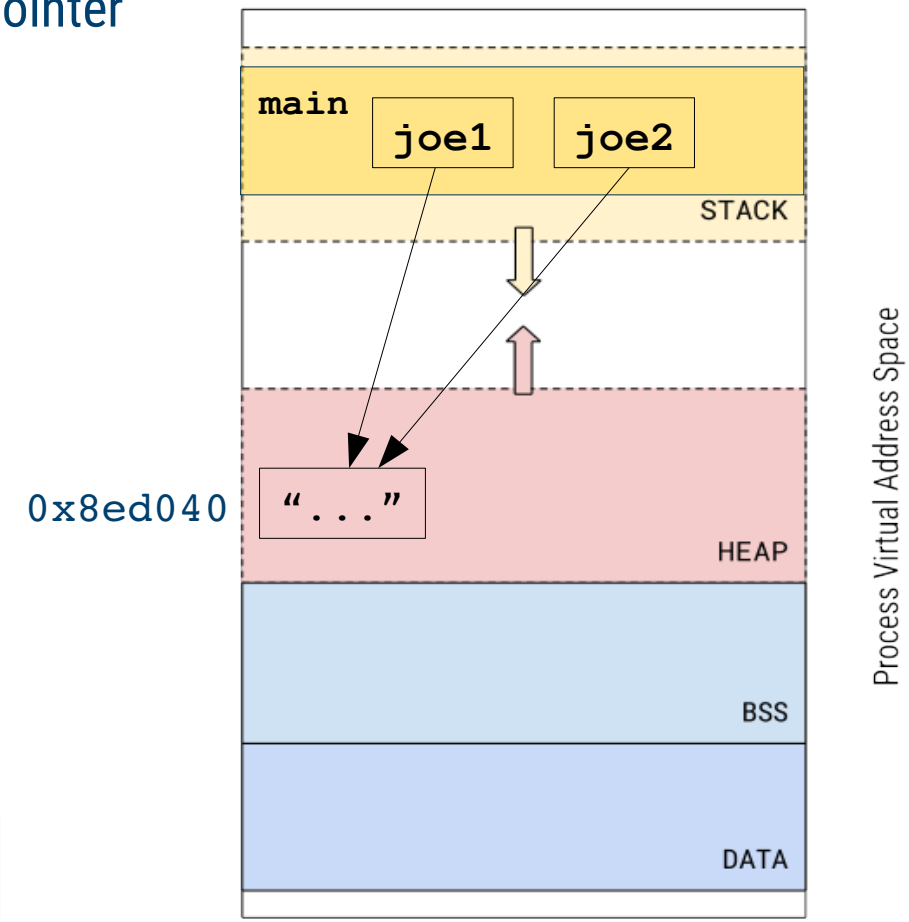
- *Pointers* allows multiple variables to jointly access the same object
  - Assignment operation copies the pointer value, i.e., the memory address

```
int main() {
    People * joe1 =
        new People("Joe");
    People * joe2;

    joe2 = joe1;
    joe1->Walk();
    joe1->Talk();
    joe2->Talk()

    delete joe2;
    return 0;
}
```

```
Joe made 1 steps
Joe made 1 steps
```



Constant members

- Data or function members qualified with `const` keyword
- Const-qualified variable are used to store data that must not change
- Qualifier `const` applies to whatever is on its immediate left
  - If there is nothing there applies to whatever is its immediate right

```
const int * k; // k is a variable pointer to a constant integer

int const * k; // ...alternative syntax which does the same

int * const k; // k is constant pointer to an integer variable

int const * const k; // k is a constant pointer to a constant integer
```

Constant members

- The `const` keyword can be used also in functions
- To specify a “read-only” function, i.e., that does not change the object status (member data values)

```
void Talk() const;
```

- To pass (by reference) arguments (typically objects) that we do not want to be modified

```
void TalkWith(const & People other);
```

  - Function `TalkWith()` cannot access not `const` member functions of `other`
- To prevent alteration of returned variable of pointer types
  - Alteration attempts are detected at compile-time

```
const People * GetParent() const;
```

Static members

- The `static` keyword can be used in a class definition for different purposes, depending on if it is applied to data or functions
- Static member ***data*** are variables shared among all the instances
  - Member data initialization cannot be performed in the class

```
class MyClass { // myclass.h
    static int count;
};
```

```
// myclass.cpp
int MyClass::count = 0;
```

- Static member ***functions*** do not require a class instance
  - Functions meaningful at *class-level* instead of *instance-level*
  - They cannot access *non static* members since don't "belong" to any instance

```
class MyClass { // myclass.h
public:
    static int GetCount();
};
```

```
std::cout
    << MyClass::GetCount()
    << std::endl;
```

Static members

- Example

```
class People {
public:
    People();
…
    static int GetPopulation();
private:
    …
    static int population;
};
// people.h
```

```
int main() {
    People joe;
    std::cout <<
        People::GetPopulation();
    …
}
```

```
//Init
int People::population = 0;

People::People():
    name("guy"),
    step_count(0) {
    population++;
}

People::~People() {
    population--;
}

int People::GetPopulation() {
    return population;
}

// people.cpp
```

Inheritance

- One of the most important properties of object-oriented programming
- A derived class inherits *public* and *protected* members of the base class
  - ctors/dtor, friend and private members are excluded
- C++ supports multiple inheritance
  - A derived class can have multiple base classes
- For the base class(es) we can specify an *access-specifier*
  - `public` → *Same access policy of the base class*
  - `protected` → *Public members of base class become protected in the derived one*
  - `private` → *Public members of base class become private in the derived one*

Inheritance

- Syntax

```
class Man: public People, public Animal {

public:
...

protected:
...

private:
...

};
```

Polymorphism

- The property of an object to take on multiple “forms“
- In C++ a pointer to a derived class is type-compatible with a pointer to its base class
  - `base_class * ptr = new derived_class(…); //valid`
- We can dynamically allocate objects of a derived class into arrays declared of type “base class”
- Let’s consider an example...
  - Base class: Polygon
  - Derived classes: Triangle and Rectangle

Polymorphism

- Example

```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

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

class Triangle: public Polygon {
  public:
    int area() { return width*height/2; }
};
```

Polymorphism

- Example: base class Polygon, derived classes Triangle and Rectangle

```
...
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  std::cout << rect.area() << '\n';
  std::cout << trgl.area() << '\n';
  return 0;
}
```

```
20
10
```

- *ppoly1* and *ppoly2* can access only members inherited from Polygon
  - What if we want to have access to member function `area()`?

Polymorphism

- *virtual* functions → functions later redefined in derived classes

```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area() { return 0; }
};

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

class Triangle: public Polygon {
  public:
    int area() { return width*height/2; }
};
```

Polymorphism

- Example: base class Polygon, derived classes Triangle and Rectangle

```
...
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  std::cout << ppoly1->area() << '\n';
  std::cout << ppoly2->area() << '\n';
  return 0;
}
```

```
20
10
```

- **`virtual`** qualifier made Rectangle and Triangle's member **`area()`** accessible through pointers to Polygon

Polymorphism

- An *abstract* class is a class including *pure virtual* functions, i.e., a function for which the implementation is not provided
  - Member function must be implemented in derived classes

```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area() = 0;
};
```

- An abstract class cannot be used for instantiate objects
  - We can still use to creating pointers (e.g., `Polygon * ppoly`) and exploit polymorphism propery
- Useful for defining “interfaces”

Outline

Outline

Standard Template Library (STL)

Template

- A way of making functions or classes more abstract
- Focus on the behaviour of the function (or the class) without knowing variables data type
  - A single function / class implementation to cover several data type cases
  - Avoid code replications (simply due to variables data type)
- Data type resolution performed at compile-time
  - Compilers generate additional code
    - → Large use of template may strongly increase the executable size
    - → Longer compilation times
  - Debugging is complicated by hard to read compiler messages
  - Template code is placed in header files
    - → Modify a template class may lead to entire project re-build

Template functions

- Example of global function performing addition between two variables of a generic type

```
…
template<typename T>
T add(T a1, T a2)
{
    return a1 + a2;
}
…
int main() {
    int i1, i2;
    float f1, f2;
    …
    cout << add<int>(i1, i2)
        << endl;
    cout << add<float>(f1, f2)
        << endl;
    return 0;
}
```

The compiler will generate two versions of function **add()**

```
int add(int a1, int a2)
{
    return a1 + a2;
}
```

```
float add(float a1, float a2)
{
    return a1 + a2;
}
```

Class template

- A class including generic type members (Example: stack.h)

```
#ifndef STACK_H_
#define STACK_H_

namespace aos {

template <class T>
class Stack {
Public:
    template <class T> Stack(int _size) { … }

    template <class T> void push(T item) {   … }

    template <class T> T pop() { … }

};
} // namespace aos

#endif // STACK_H_
```

Class template

- We can instantiate objects of the same class, working on data of different types

```
#include “my_stack.h”
namespace aos {

int main() {
    Stack<int> s1;
    Stack<float> s2;
    Stack<People> s3;

    s1.push(1);
    s2.push(6.0);
    People simon(“Simon”);
    s3.push(simon);
    ...
    return 0;
}

} // namespace aos
```

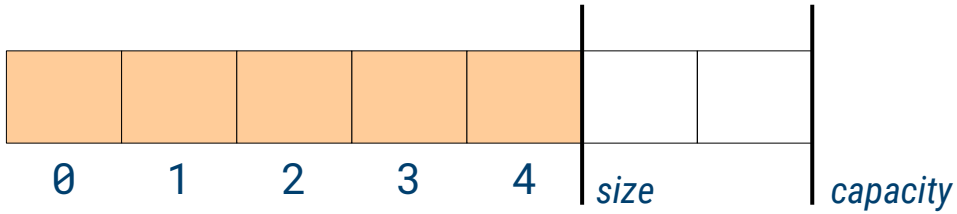
Standard Template Library (STL)

STL

- A powerful set of C++ template classes
- Provides general-purpose template-based classes and functions
- Implement commonly used algorithms and data structure
- *Containers* are the most noticeable example of data structures
  - List, queues, stack, map, set, …
  - Used to manage a collection of object of a certain class
  - The differ for functionality, typology (*sequential*, *associative*, *unordered*), and complexity of accessing and manipulation operations
- *Iterators* are used to step through the elements of the containers
- *Algorithms* includes function manipulating containers
  - Search, copy, sort, transform,...

For a complete list: http://www.cplusplus.com/reference/stl/

vector



- Sequence container representing dynamic arrays (mutable size)
- Contiguous storage of elements with random access
- High efficiency in accessing items - O(1)
- Relatively efficient in insert/delete items at the end
- Not much efficient in insert/delete operations in/from other positions
- If *size* > *capacity* the entire vector is reallocated in a bigger memory space

vector

- Example (C++03)

```
int main() {
    People claire("Claire");
    People john("John");
    vector<People> citizens;      // Container of People
    citizens.push_back(john);     // Add to vector
    citizens.push_back(claire);   // Add to vector
    vector<People>::iterator it;
    for(it = citizens.begin(); it != citizens.end(); ++it) {
        cout << "Name: " << it->GetName() << endl;
    }
    cout << "Citizen 1 is" << citizens[1].GetName() << endl;
    return 0;
}
```

```
Name: John
Name: Claire
Citizen 1 is Claire
```

vector

- Example (C++11)

```
int main() {
    People claire("Claire");
    People john("John");
    vector<People> citizens;      // Container of People
    citizens.push_back(john);     // Add to vector
    citizens.push_back(claire);   // Add to vector

    for(auto & c: citizens) {
        cout << "Name: " << c.GetName() << endl;
    }
    cout << “Citizen 1 is” << citizens[1].GetName() << endl;
    return 0;
}
```

```
Name: John
Name: Claire
Citizen 1 is Claire
```

list

- Sequence containers implementing double-linked lists
- Very efficient insert/remove operations - O(1)
- Linear complexity in accessing the single item - O(n)
- Extra memory utilization due to storing pointers to ‘prev’ and ‘next’ item
- Good for sorting-like operations

list

- Example (C++11)

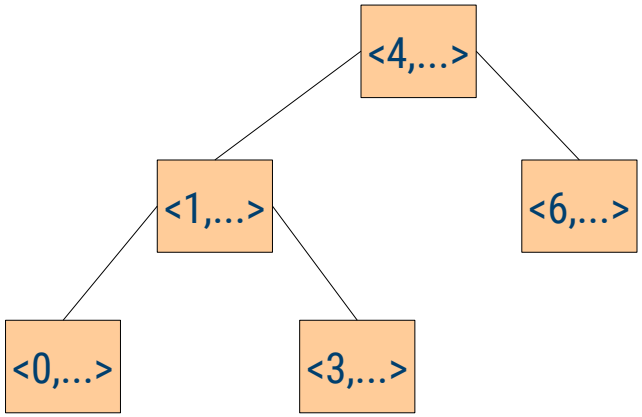
```
int main() {
    People claire("Claire");
    People john("John");
    list<People> citizens;          // Container of People
    citizens.push_back(john);       // Add to list
    citizens.push_back(claire);     // Add to list

    for(auto & c: citizens) {
        cout << "Name: " << c.GetName() << endl;
    }
    return 0;
}
```

```
Name: John
Name: Claire
```

Standard Template Library (STL)

map



- Associative containers storing pairs *<key, value>* where the value represents the data, and the key is used for sorting and access purposes
- O(log n) complexity to access elements
- O(log n) complexity to insert/remove elements
- Typically implemented as binary search trees
- Slower than *unordered* containers

map

- Example (C++11)

```
int main() {
    People claire("Claire");
    People john("John");
    map<std::string, People> citizens;    // Container of People
    citizens.emplace("John", john);       // Add to map
    citizens.emplace("Claire", claire);   // Add to map

    for(auto & c_pair: citizens) {
        cout << "Key: " << c_pair.first
            << " / Name: " << c_pair.second.GetName() << endl;
    }
    return 0;
}
```

```
Key: Claire / Name: Claire
Key: John / Name: John
```

Outline

C++ Programs

- Memory allocation
- Passing parameters to functions

Classes

- Constructors and assignments
- Object pointers
- Qualifiers
- Inheritance and polymorphism

Standard Template Library (STL)

- Templates
- Example: `vector`, `list`, `map`

Memory management in C++11

- From raw to smart pointers

Raw pointers

- Using (*raw*) pointers is a powerful C/C++ feature, but in general difficult to handle and error-prone
- Consider the following example:

```
int main() {
    People * joe = new People("Joe");
    People * ann = new People("Ann");
    People * alex = new People("Alex");
    joe->SetChild(alex);
    ann->SetChild(alex);

    delete joe;
    delete ann;
    delete alex;
    return 0;
}
```

```
class People {
public:
    virtual ~People();
    void SetChild(
        People * _child);
private:
    People * child;
}
```

```
People::~People() {
    delete child;
}
```

- → "Segmentation fault" since "**`alex`**" is deallocated multiple times!

Smart pointers

- To solve the problem we should start from the concept of *ownership*
- To own a dynamically allocated object means to be the entity in charge of deallocating it

- A feature introduced by the C++11 standard, to simplify dynamic memory management in C++ programs
  - Concept of ownership introduced and exploited
  - Limited *garbage collection* facilities

- *Base idea*: *To wrap raw pointers in stack allocated objects that could control the life-cycle of the dynamically allocated object*
- We are going to focus on just two class of smart pointer
  - Unique pointers (class **`unique_ptr<>`**)
  - Shared pointers (class **`shared_ptr<>`**)

Shared pointers

- Pointers of type *shared* allows us to keep track of the number of shared ownership, i.e. references to the objects
- Reference counting based mechanism
  - If the count is decreased to zero the object is automatically deallocated
  - Assignments and scope exit alter the reference counters
  - A little additional memory overhead

```
#include <memory>
using namespace std;
int main() {
    shared_ptr<People> jack = make_shared<People>("Jack");
…
}
```

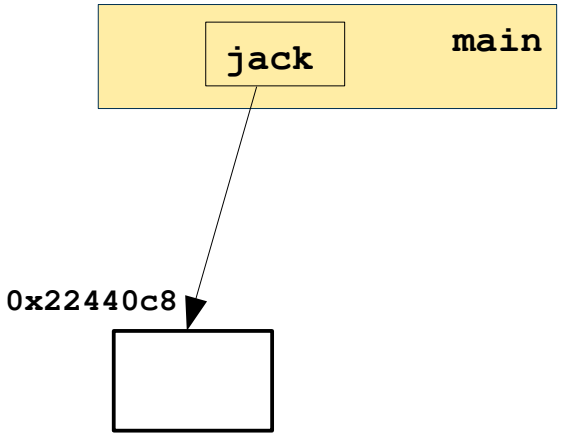
- No more explicit **`new`**/**`delete`** calls
- Append **`-std=c++11`** to gcc command line

Shared pointers

- Similarly to pointer variables...
  - Object “jack” (of class **`shared_ptr<>`**) is allocated onto the stack
  - Members of the pointed object are accessed by using the “**`->`**” operator
- **`shared_ptr<>`** member function **`get()`** returns the raw pointer value
  - Using “ **`.`** ” operator to access **`shared_ptr<>`** member functions

```
…
  shared_ptr<People> jack
        = make_shared<People>("Jack");
  cout << "Name = "<< jack->GetName() << endl;
  cout << "Raw ptr = " << jack.get() << endl;
…
```

```
Name = Jack
Raw pointer = 0x22440c8
```



Shared pointers

- Example: an assignment operation and a scope exit

```
#include <memory>
using namespace std;
int main() {
    …
    shared_ptr<People> jack = make_shared<People>("Jack");
    cout << "Jack ref.count = " << jack.use_count() << endl;
    {
        shared_ptr<People> jack2;
        jack2 = jack;
        cout << "Jack ref.count = " << jack.use_count() << endl;
    }
    cout << "Jack ref.count = " << jack.use_count() << endl;
    return 0;
}
```

```
Jack ref.count = 1
Jack ref.count = 2  // jack2 = jack
Jack ref.count = 1  // jack2 out of scope => jack ref.count decrease
Jack: i'm dying...  // People object “Jack” destruction
```

Shared pointers

- The previous “parents” example can be rewritten as follows, using shared pointers (changing also `SetChild()` signature)

```
#include <memory>
using namespace std;
int main() {
    shared_ptr<People> jack = make_shared<People>("Jack");
    shared_ptr<People> ann  = make_shared<People>("Ann");
    shared_ptr<People> alex = make_shared<People>("Alex");
    jack->SetChild(alex);
    ann->SetChild(alex);
    return 0;
}
```

- No need of explicit delete (objects deallocated at main function exit)
- No multiple deallocation thanks to reference counting

Unique pointers

- Do not share the ownership of a pointed object
  - Unique pointer means unique ownership
  - Only one object responsible of memory deallocation
- Copy assignments not allowed, only move assignments are

```
#include <memory>
using namespace std;
int main() {
    unique_ptr<People> jason(new People("Jason"));
    unique_ptr<People> laura;
    cout << "Jason = " << jason.get() << endl;
    laura = std::move(jason);
    cout << "Jason = " << jason.get() << endl;
    cout << "Laura = " << laura.get() << endl;
}
```

```
Jason = 0x15fd040
Jason = 0
Laura = 0x15fd040
```

Unique pointers

- Standard C++14 has introduced **`make_unique<>`** function
  - Append **`-std=c++14`** to gcc command line
- **`make_*`** functions are preferrable with respect to to using "**`new ...`**"
  - Safer and more efficient option

```
#include <memory>
using namespace std;
int main() {
    unique_ptr<People> jason = make_unique<People>("Jason"));
    unique_ptr<People> laura;
    cout << "Jason = " << jason.get() << endl;
    laura = std::move(jason);
    cout << "Jason = " << jason.get() << endl;
    cout << "Laura = " << laura.get() << endl;
}
```

Objects and containers

- Containers can be used to store objects (stack allocated)
  - An object copy is performed when a new object is added to the collection
- Containers can be used to store pointers to objects (heap allocated)

```
int main() {
    vector<People *> citizens;  // Using raw pointers
    People * p1 = new People("p1");
    People * p2 = new People("p2");
    citizens.push_back(p1);
    citizens.push_back(p2);
    citizens.clear();           // Clear the container
    return 0;
}
```

- For dynamically allocated objects keep in mind the potential pitfalls about memory management

  `clear()` calls objects destructors... but what about raw pointers?

Objects and containers

- We may used smart (e.g., shared) pointers also in containers

```
int main() {
    vector<shared_ptr<People>> citizens; // Using shared pointers
    shared_ptr<People> p1 = make_shared<People>("p1");
    shared_ptr<People> p2 = make_shared<People>("p2");
    citizens.push_back(p1);
    citizens.push_back(p2);
    cout << "p1 use count = " << p1.use_count() << endl;
    return 0;
}
```

```
p1 use count = 2
p1: i'm dying...
p2: i'm dying...
p3: i'm dying...
```

- Insert/remove operations affect shared pointers reference counting
- Destroying the vector the reference counters are decreased

References

Web links

- http://www.cplusplus.com
- http://en.cppreference.com/w/
- http://www.horstmann.com/ccj2/ccjapp3.html
- http://www.cprogramming.com/tutorial/constructor_destructor_ordering.html
- http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.html
- http://duramecho.com/ComputerInformation/WhyHowCppConst.html
- http://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm