Calendario Lezioni (Provvisorio e Soggetto a Modifiche)

Data	Ore di attività frontale: 48	Numero Lezione	Lezione	Esercitazione	Modulo	Descrizione	Topic
17/09/24	2	1	1				A - Logistica 00 - Introduzione ai Linguaggi di Programmazione
18/09/24	2	2	1		M0	Conoscenza C	01 - Fundamentals/Computability
24/09/24	2	3	1				02 - Memory management per i blocchi inline e per le chiamate di funzione
25/09/24	2	4	1				03-04 Passaggio parametri valore, riferimento, puntatori … Funzioni ricorsive + esercizio di RA, litmus test, tail recursion.
01/10/24	2	5		1	M 1	scope e memory management	E01 - Esercitazione C
02/10/24	2	6	1				05 - 06 - Sicurezza dei tipi C is not Safe
08/10/24	2	7	1				06 - Programmi sicuri in c [cyclon rimosso] Lint \ Valgrind
09/10/24	2	8		1	M 2	Type Safety	E02 - Esercitazione C
15/10/24	2	9	1				08 - Object Oriented [Information hiding / tipi opachi in c]
16/10/24	2	10	1				09 - Design Pattern
30/10/24	2	11		1	M 3	object oriented	E03 - esercizi + ereditarietà e sottitpazione
01/11/24	2	12	1				11 - java Classes and Inheritance
05/11/24	2	13	1				12 - Java Generics
12/11/24	2	14		1	M4	Java	E4 - Java Classes and Inheritance Generici

Calendario Lezioni (Provvisorio e Soggetto a Modifiche)

13/11/24	2	15	1				16 - cpp classes
19/11/24	2	16	1				17 - cpp encapsulation and inheritance
20/11/24	2	17		1			E05 - esercizi cpp
26/11/24	2	18	1		M5	C++	18 - virtual functions/ sottotipazione/STL
27/11/24	2	19	1				19 - Scala Introduzione
03/12/24	2	20	1				22 - Scala Introduzione
04/12/24	2	21	1				23 - Scala Introduzione
10/12/24	2	22		1	M6	Programmazione Funzionale Scala	E06 - Esercitazione Scala
11/12/24	2	23		1			Simulazione tema d'esame
17/12/24	2	24		1			Simulazione tema d'esame

TOTALE ORE		Tot Lezione [32]	Tot Esercitazione [16]	Totale
48		32	16	48

Introduction to C++

Informatica III – parte A
A. Gargantini

History

- C++ is an object-oriented extension of C [1985 from 1997 OO]
- C was designed by Dennis Ritchie at Bell Labs
  - used to write Unix
  - based on BCPL
- C++ designed by Bjarne Stroustrup at Bell Labs
  - His original interest at Bell was research on simulation
  - Early extensions to C are based primarily on Simula
  - Called “C with classes” in early 1980’s
  - Popularity increased in late 1980’s and early 1990’s
  - Features were added incrementally
- Classes, templates, exceptions, multiple inheritance, type tests...

Design Goals

- Provide object-oriented features in C-based language, without compromising efficiency
  - Backwards compatibility with C
  - Better static type checking
  - Data abstraction
  - Objects and classes
  - Prefer efficiency of compiled code where possible
- Important principle
  - If you do not use a feature, your compiled code should be as efficient as if the language did not include the feature.

What is Data Abstraction?

- Abstract Data Types (ADTs)
    - type implementation & operations
    - hidden implementation
- types are central to problem solving
    - Not procedures like in C
- a weapon against complexity
- built-in and user-defined types are ADTs

How Well are ADTs Supported in C?

- Does C enforce the use of the ADTs interface and the hiding of its implementation?

- No

C++

- C++ is a superset of C, which has added features to support object-oriented programming

- C++ supports classes
    - things very like ADTs

How successful?

- Given the design goals and constraints,
  - this is a very well-designed language
- Many users -- tremendous popular success
- However, very complicated design
  - Many specific properties with complex behavior
  - Difficult to predict from basic principles
  - Most serious users chose subset of language
    - Full language is complex and unpredictable
  - Many implementation-dependent properties
  - Language for adventure game fans

Further evidence

- Many style guides for using C++ “safely”
- Every group has established some conventions and prohibitions among themselves.
  - don’t inherit implementation
  - SGI compiler group -- no virtual functions
  - Others

Significant constraints

- C has specific machine model
  - Access to underlying architecture
- No garbage collection
  - Consistent with goal of efficiency
  - Need to manage object memory explicitly
- Local variables stored in activation records
  - Objects treated as generalization of structs, so some objects may be allocated on stack
  - Stack/heap difference is visible to programmer

Overview of C++

- Additions and changes not related to objects
  - type bool
  - pass-by-reference & the Copy-Constructor
  - user-defined overloading
  - function template
  - exception handling
  - …

OO Programming Languages

- Four main concepts:

1. **Abstraction**: implementation details hidden inside a program unit with a specific *interface*. The interface is a set of public functions (or methods) over hidden data.
2. **Inheritance**: reusing the definition of one kind of object to define another kind of object.
3. **Dynamic lookup**: a method is selected at run time, according to the *implementation* of the object, not some static property of the pointer/var used to name the object.
4. **Subtyping** is a relation on types that allows values (or objects) of one type to be used in place of values (or objects) of another.

Inheritance Is Not Subtyping!

"Subtyping is a relation on interfaces,
inheritance is a relation on implementations."

C++ Object System

- Object-oriented features
  1. Classes and Data Abstraction
  2. Encapsulation
  3. Inheritance
     - Single and multiple inheritance
     - Public and private base classes
  4. Objects, with dynamic lookup of virtual functions
  5. Subtyping
     - Tied to inheritance mechanism
     - A will be recognized by the compiler as a subtype of B only if B is a public base class of A

Objects in C++
Classes and Data Abstraction

Programmazione avanzata
Angelo Gargantini
AA 22/23

Classi vs strutture

Classi vs oggetti

C++ Object System

Object-oriented features

- Classes and Data Abstraction
- Encapsulation
- Inheritance
    - Single and multiple inheritance
    - Public and private base classes
- Objects, with dynamic lookup of virtual functions
- Subtyping
    - Tied to inheritance mechanism

C++ Object System

Object-oriented features

- **Classes and Data Abstraction**
- Encapsulation
- Inheritance
  - Single and multiple inheritance
  - Public and private base classes
- Objects, with dynamic lookup of virtual functions
- Subtyping
  - Tied to inheritance mechanism

Abstraction

- **Abstraction** means that implementation details are hidden inside a program unit with a *specific interface*.
- For objects, **the interface** consists of a set of public functions (or methods) that manipulate hidden data.
- Abstraction involves restricting access to a program component according to its specified interface.

C++: Classes and Data Abstraction

- C++ supports Object-Oriented Programming (OOP)
- OOP models real-world objects with software counterparts
- OOP encapsulates data (attributes) and functions (behavior) into packages called objects
- Objects have the property of information hiding

C++: Classes and Data Abstraction

Objects communicate with one another across interfaces

The interdependencies between the classes are identified

- makes use of
- a part of
- a specialisation of
- a generalisation of
- etc.

C and C++

- C programmers concentrate on writing functions
- C++ programmers concentrate on creating their own user-defined types called classes
- Classes in C++ are a natural evolution of the C notion of **`struct`**

Namespaces

A namespace is a declarative region that provides a scope to the identifiers inside it.

Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

Namespaces

All identifiers at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access the members by using the fully qualified name for each identifier, for example std::vector<std::string> vec;, or else by a using declaration for a single identifier (using std::string), or a using directive for all the identifiers in the namespace (using namespace std;).

Namespaces

The following example shows a namespace declarantion

```
namespace ContosoData{
    class ObjectManager   { ....
    };
    void func(ObjectManager) {}
}
```

A User-Defined Type Time with a struct

```
// Create a structure, set its members, and print it
// structure definition
struct Time {
   int hour; // 0-23
   int minute; // 0-59
   int second; // 0-59
};

void printMilitary(const Time &); // prototype
void printStandard(const Time &); // prototype
```

```
main()
{
    Time dinnerTime;  // variable of new type Time

    // set members to valid values
    dinnerTime.hour = 18;
    dinnerTime.minute = 30;
    dinnerTime.second = 0;

    cout << "Dinner will be held at";
    printMilitary(dinnerTime);  // 18:30:00
    cout << " military time,\nwhich is ";
    printStandard(dinnerTime);  //  6:30:00 PM
    cout << " standard time." << endl;
```

adds a newline ("\n") and flushes the buffer

```
   // set members to invalid values
   dinnerTime.hour = 29;
   dinnerTime.minute = 73;
   dinnerTime.second = 103;

   cout << "\nTime with invalid values: ";
   printMilitary(dinnerTime);  // 29:73:103 bad values!
   cout << endl;

   return 0;
}// end main
```

```
// Print the time in military format
void printMilitary(const Time &t)
{
    cout << (t.hour < 10 ? "0" : "") << t.hour << ":"
   << (t.minute < 10 ? "0" : "") << t.minute << ":"
   << (t.second < 10 ? "0" : "") << t.second;
}

// Print the time in standard format
void printStandard(const Time &t)
{
  cout << ((t.hour == 0 || t.hour == 12) ? 12 : t.hour % 12)
   << ":" << (t.minute < 10 ? "0" : "") << t.minute
   << ":" << (t.second < 10 ? "0" : "") << t.second
   << (t.hour < 12 ? " AM" : " PM");
}
```

Comments

- Initialization is not required --> can cause problems
- A program can assign bad values to members of Time
- If the implementation of the `struct` is changed, all the programs that use the `struct` must be changed [No “interface”]

A Time Abstract Data Type with a Class

```
#include <iostream.h>
// Time abstract data type (ADT) definition
class Time {
public:
    Time();                  // default constructor
  void setTime(int, int, int);
  void printMilitary();
  void printStandard();
private:
  int hour;    // 0 - 23
  int minute;  // 0 - 59
  int second;  // 0 - 59
};
```

NOTA: può essere separato dalla dichiarazione di classe. In altro file .cpp

```
// Time constructor initializes each data member to zero.
// No return value
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time.
// Perform validity checks on the data values.
// Set invalid values to zero (consistent state)
void Time::setTime(int h, int m, int s)
{
  hour = (h >= 0 && h < 24) ? h : 0;
  minute = (m >= 0 && m < 60) ? m : 0;
  second = (s >= 0 && s < 60) ? s : 0;
}
```

```
// Print Time in military format
void Time::printMilitary()
{
  cout << (hour < 10 ? "0" : "") << hour << ":"
   << (minute < 10 ? "0" : "") << minute << ":"
   << (second < 10 ? "0" : "") << second;
}

// Print time in standard format
void Time::printStandard()
{
  cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
   << ":" << (minute < 10 ? "0" : "") << minute
   << ":" << (second < 10 ? "0" : "") << second
   << (hour < 12 ? " AM" : " PM");
}
```

```
// Driver to test simple class Time
main()
{
    Time  t; // instantiate object t of class Time

   cout << "The initial military time is ";
   t.printMilitary(); // 00:00:00
   cout << "\nThe initial standard time is ";
   t.printStandard(); // 12:00:00 AM

   t.setTime(13, 27, 6);
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary(); // 13:27:06
   cout << "\nStandard time after setTime is ";
   t.printStandard(); // 1:27:06 PM
   t.setTime(99, 99, 99);
   // attempt invalid settings
   cout << "\n\nAfter attempting invalid settings:\n"
        << "Military time: ";
   t.printMilitary(); // 00:00:00
   cout << "\nStandard time: ";
   t.printStandard(); // 12:00:00 AM
   cout << endl;

   return 0;
} // end main
```

Output

- The initial military time is 00:00:00
- The initial standard time is 12:00:00 AM

- Military time after setTime is 13:27:06
- Standard time after setTime is 1:27:06 PM

- After attempting invalid settings:
- Military time: 00:00:00
- Standard time: 12:00:00 AM

Comments

- `hour`, `minute`, and `second` are private data members. They are normally not accessible outside the class. [Information Hiding]
- Use a constructor to initiailize the data members. This ensures that the object is in a consistent state when created.
- Outside functions set the values of data members by calling the setTime method, which provides error checking.

Classes as User-Defined Types

- Once the class has been defined, it can be used as a type in declarations as follows:

L'intero oggetto Time è allocato sullo stack

L'intero array di 5 Time è allocato sullo stack

```
Time sunset;                      //object of type Time
Time arrayOfTimes[5];    //array of Time objects
Time *pointerToTime;      //pointer to a Time object
```

Solo il puntatore è sullo stack – devo creare l'oggetto

Using Constructors

- Constructors can be overloaded, providing several methods to initialize a class.

Interface

```
Time();          // default constructor
Time(int hr);
Time(int hr, int min, int sec);
```

Implementation

```
Time::Time(){ hour = minute = second = 0; }
Time::Time(int hr) { setTime(hr, 0, 0); }
Time::Time(int hr, int min, int sec)
    { setTime(hr, min, sec); }
```

Using Constructors for «static» object (sullo stack)

```
Time t1; // Time() is invoked
Time t1(); //ERROR, intended as a funct prototype

Time t2(08);  // class_name object_name(values)
Time t2 = Time(08);
Time t2 = 08;
Time t2 = (Time) 08; // cast

Time t3(08,15,04);
Time t3 = Time(08,15,04);
```

Using Constructors and dynamic objects

```
Type_name * pointer_name;
pointer_name = new Type_name;
```

where `Type` is a Class or a primitive type

```
int *ptr;
ptr = new int;

Time *t;
t = new Time;             // Time() is invoked
t = new Time(08);         // Time(int) is invoked
t = new Time(08,15,04);   // Time(int, int, int)
                          // is invoked
```

Using Constructors and array of objects

```
Time arrayOfTimes[5]; //Time() is invoked
```

Explicit array initialization:

```
//Only the first four elements are inizialized
//Time() (if any) is invoked for the other elements
Time arrayOfTimes[8] = { 3, Time(05), Time(),
  Time(01,12,03)}
```

Using Constructors and dynamic arrays

```
Time *t = new Time[8];
// Time() is invoked for each element
```

positive, can be variable   positive, constant

```
int i = 3;
Time (*t) [20] = new Time[3*i] [20];
// Multi-dimension array
// Time() is invoked for each element
```

In both cases, explicit initialization is not allowed!

The constructor initializer list

- **A list of "constructor calls"** that appears only **in the definition of the constructor** – after the argument list
- The initialization in the list is executed before any of the main constructor code.
- This is the place to put all **const** initializations, primitive type variables and object variables, **except arrays**.

```
class Info
private:
 const int i;
 double m;
 Time t;
Public:
 Info(); // default constructor
};

Info::Info(int j, double n) : i(j), m(n), t(i) {}
```

Copy constructors

An object can be built starting from an existing one:

S u (s); // Calls S’s copy - constructor

S v = s; // Calls S’s copy – constructor

The copy constructor can be defined:

class S { public: S(const S&); };

In case it is not defined, the compiler will add a default that copies all the fields (even the pointers)

Destructors (1)

- To guarantee cleanup when using dynamic memory
- Destroy objects by
  - Calling the destructors of object member variables
  - Calling superclass destructors (if virtual)
- The destructor is called
  - At the end of object lifetime
  - Or during a call to delete
- Normaly the is no need to call the destructor explicitly

Destructors (2)

- A public function member **~class_name** with no parameters and no return values

```
Class_name::~class_name() {
//delete operations
…
}
```

- Operator **delete**
- can be called only for an object created by **new**

```
delete ptr;
delete [] ptr; //se è un array
```

Attenzione a non fare delete di puntatori a zone sullo stack

new() and delete() (1)

- For each new statement, you must provide exactly one corresponding delete statement
- Failing to do so causes memory and resource leaks and can cause undefined behavior ...

new() and delete() (2)

- Allocating memory

```
int* myInt = new int;
int* myIntArray = new int[10];
```

- Deallocating memory

```
delete myInt;
delete[] myIntArray;
```

Deleting zero pointers

If the pointer you're deleting is zero, nothing will happen.

For this reason, people often recommend setting a pointer to zero immediately after you delete it, to prevent deleting it twice.

```
delete p;
p = nullptr;
```

Deleting an object more than once is definitely a bad thing to do, and will cause problems.

Function Declaration

- A function is declared by

```
returnType funcName(
            typename arg1, ...,
            typename argN)
```

- Member function can include a const modifier in their signature

```
void helloWorld::sayHello(void) const
```

- A const method cannot modify class members
- private/protected/public modifier are not part of the function declaration

Function declaration: *Const* modifier

```
#include <iostream.h>
Class Car{
 private:
  int lenght;
  double weight;
 public:
  int fun_weight(double) const;
};

int Car::fun_weight(double new_weight) const
{
 // weight++; ERROR
 new_weigth += weight;
 return (int) new_weight;
}
```

Function Declaration Examples

```
void output(const std::string& s);

double multiply(const double fac1,
                const double fac2);

void addHeader(void* buf, const Date& date);

int main(int argc, char* argv[]);
int main(int argc, char** argv);

void doSomething(SomeBigObject bo);
void doSomething(SomeBigObject* bo);
void doSomething(SomeBigObject& bo);
```

pass-by-value

pass-by-reference *

pass-by-reference &

Call by value

- Called function has its own local copy of the data
- Changes to the data are local and
- Will be discarded as soon as the namespace is left
- (highly) inefficient with large objects

Call by reference

- Passes memory address of variables to the function (word-size variable)
  - Very efficient
    - Allows variable modification avoiding double copy
- Two possible realizations in C++
  - void doSomething(Data* data);
    - Pointer based
  - Advantages and drawbacks of pointer approach
  - void doSomething(Data& data);
    - Reference based
    - No null-check necessary

pass-by-reference **&**

Object Variable Classification (like in C)

- Extern variables **`double x`**
  - global variables, the prefix *extern* when declared by other files
- Static global variables **`static double x`**
  - global variables, but can't be used by other files
  - are zero-initialized by default
- Automatic internal variables
  - defined within a function/block
- Static internal variables
  - like static external variables,
  - but defined within a function/block
  - retains its state between calls to that function

```
int count_calls()
{ static int
calls=0;
//local static
return ++calls; }
```

Extern variables

- file1.c: declares an external global var

```
int GlobalVariable;
// implicit definition
void SomeFunction();
// function prototype (decl.)


int main() {
   GlobalVariable = 1;
   SomeFunction();
   return 0;
}
```

- file2.c uses the variable

```
extern int GlobalVariable;
// explicit declaration


 void SomeFunction() {
// function header (definition)
   ++GlobalVariable;
 }
```

Static member variables

- A *static variable*, member of a class, is a variable **shared by all objects** created from the class

```
Class Car{
 private:
 static int num_cars;
 public:
   …
};
//Outside initialized, like an external variable,
//even if private!
int Car::num_cars = 22;
```

Static member functions (1)

- Executed in the same manner for all objects of the given class, e.g., to open a file or to set *static variables*.
- They can’t:
  - access to non static variables,
  - invoke non static functions,
  - use the pointer *this*
  - be declared *virtual*
  - Constructors and destructors can’t be *static*

Static member functions (2)

```
#include <iostream.h>
class Car{
 private:
 static int num_cars;
 public:
 Car(); // default constructor
 static void n_car();
};
```

Static member functions (3)

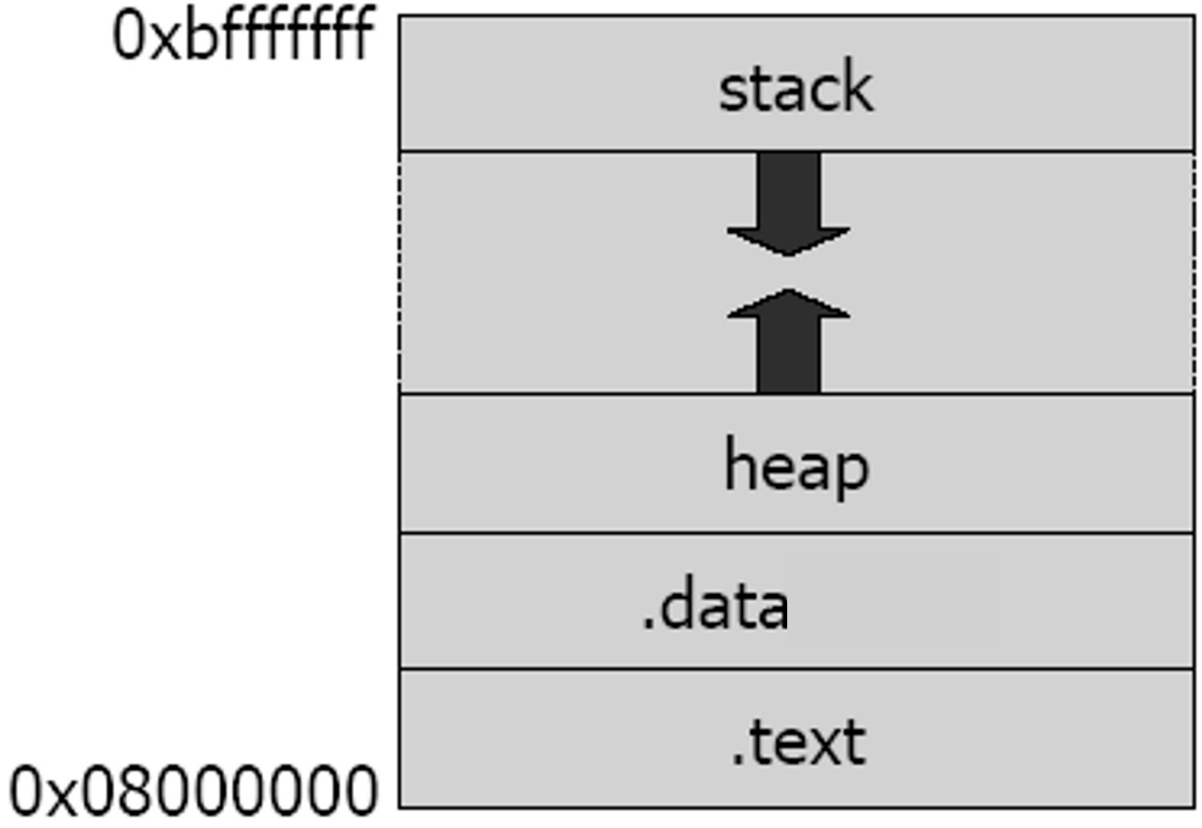
```
Car::Car() { num_cars++; }

void Car::n_car(){cout << num_cars << '\n';}

// Access to the static private variable is allowed!
int Car::num_cars = 0;

int main(int argc, char *argv[])
{
//cout << Car::num_cars;  ERROR Access to a
//private variable!
Car a;
Car::n_car(); // or a.n_car() bad style!
return 0;
}
```

Memory layout (1)



automatic variables (including pointers)

dynamic variables

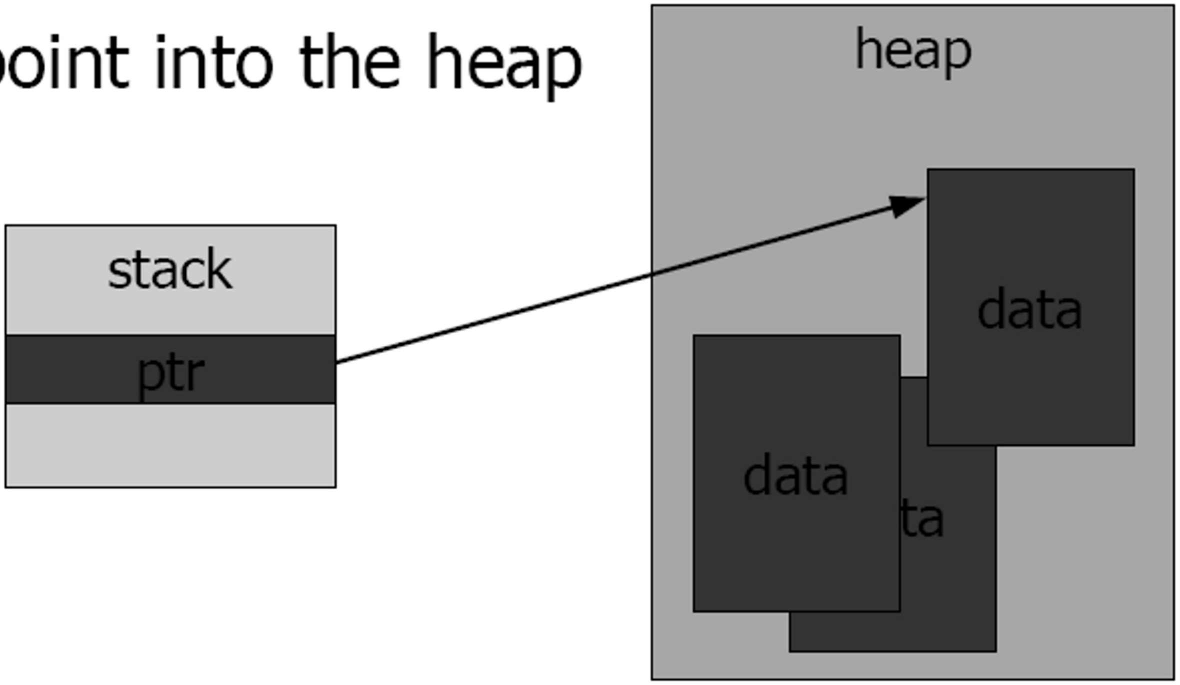
extern variables, static variables

member functions

Memory layout (2)

Pointer have a constant size of 1 word (16, 32, 64 bit)

- reside on the stack
- point into the heap



Inline functions

- Any function defined within a class body is automatically inline, but you can also make a non-class function inline by preceding it with the **inline** keyword.

```
inline int plusOne(int x) { return ++x; }

inline int plusOne(int x); //has no effect
```

- Any behavior you expect from an ordinary function, you get from an inline function.

- The only difference is that an inline function is expanded in place, like a preprocessor macro in C, so the overhead of the function call is eliminated.

Inline + ricorsione

- Anche le funzioni ricorsive possono essere dichiarate inline, in quel caso verranno sviluppate solo fino ad un certo grado impostabile come opzione in gnu cpp ad esempio
- Le funzioni definite nel .h sono inline (e spesso viceversa)

Default arguments

- When functions have long argument lists, it is tedious to write (and confusing to read) the function calls
- when most of the arguments are the same for all the calls.

- A commonly used feature in C++ is called *default arguments*.
- A *default argument* is one the compiler inserts if it isn't specified in the function call.

```
void f(int size, int initQuantity = 0);
void g(int x, int = 0, float = 1.1);
void h(int = 0, int x, float = 1.1); //ERROR
```

Function overloading

```
void f(int size, int initQuantity);
void f(int size, double initQuantity);
int f(int size, int initQuantity);//ERROR
```

- The compiler resolves the correct version of an overloaded function based on the number/type of arguments in each call

- Functions differing only in their return type cannot be overloaded.
- Since the returned value may be implicitly converted, the compiler cannot resolve which version is intended to use

- An immediately useful place for overloading is in constructors.
- In C invece non si può!!!