# Definizione della classe orario

```
int orario::Ore() {
  return sec / 3600;
}

int orario::Minuti() {
  return (sec / 60) % 60;
}

int orario::Secondi() {
  return sec % 60;
}
```

Dichiarazione della classe orario

```
class orario {
private:              // unico campo dati della classe
 int sec;             // scegliamo di rappresentare l'ora
                      // del giorno con il numero di
                      // secondi trascorsi dalla mezzanotte

public:               // metodi pubblici della classe
 int Ore();           // selettore delle ore
 int Minuti();        // selettore dei minuti
 int Secondi();       // selettore dei secondi
};
```

Avremmo anche potuto scrivere:

```
class orario {
public:
 int Ore() { return sec / 3600; }
 int Minuti() { return (sec / 60) % 60; }
 int Secondi() { return sec % 60; }
private:
 int sec;
};
```

Avremmo anche potuto scrivere:

```
class orario {
public:
 int Ore() { return sec / 3600; }
 int Minuti() { return (sec / 60) % 60; }
 int Secondi() { return sec % 60; }
private:
 int sec;
};
```

In questo caso diciamo che i metodi sono definiti *inline*

## Inline function

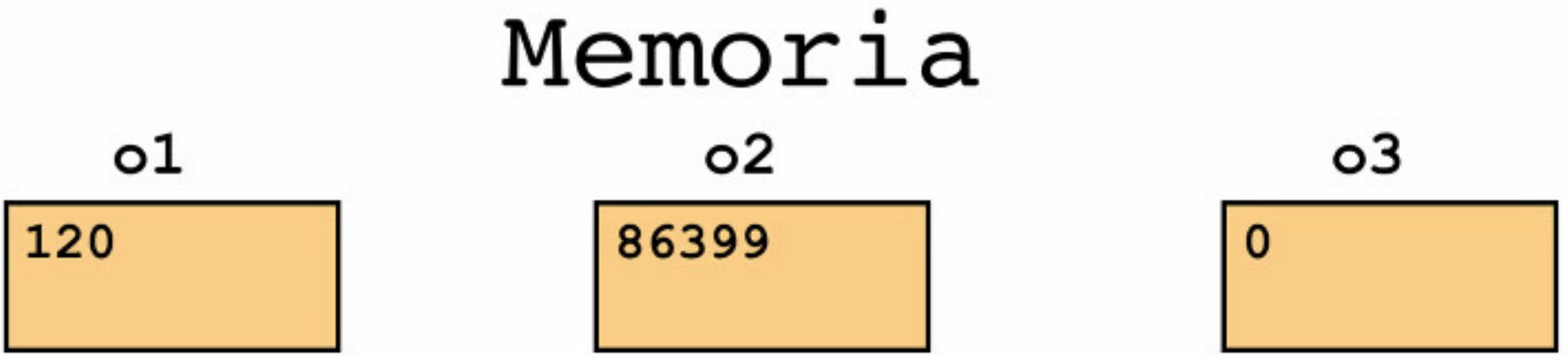
From Wikipedia, the free encyclopedia

An **inline function** is a function in C and C++ programming languages qualified with the keyword *inline* which tells the compiler to substitute the body of the function inline by performing inline expansion i.e. by inserting the function code at the address of each function call thereby saving the overhead of function invocation and return (register saving and restore) by avoiding a jump to a sub-routine.

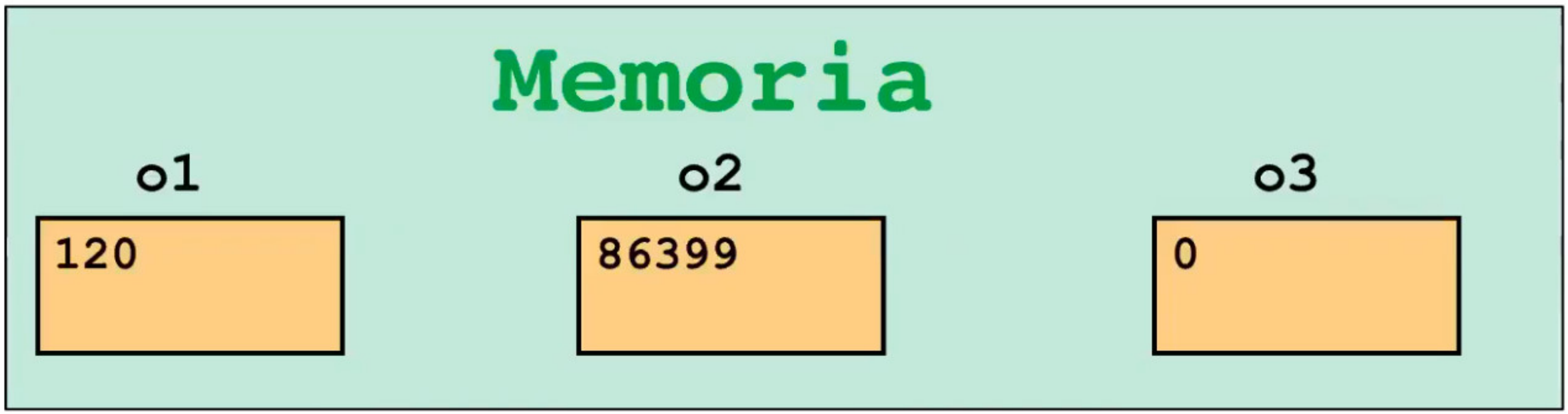
Le variabili di tipo **`orario`** vengono dette *oggetti* della classe **`orario`**.

```
int main() {
   orario mezzanotte;
   cout << mezzanotte.Secondi() << endl;
}
```

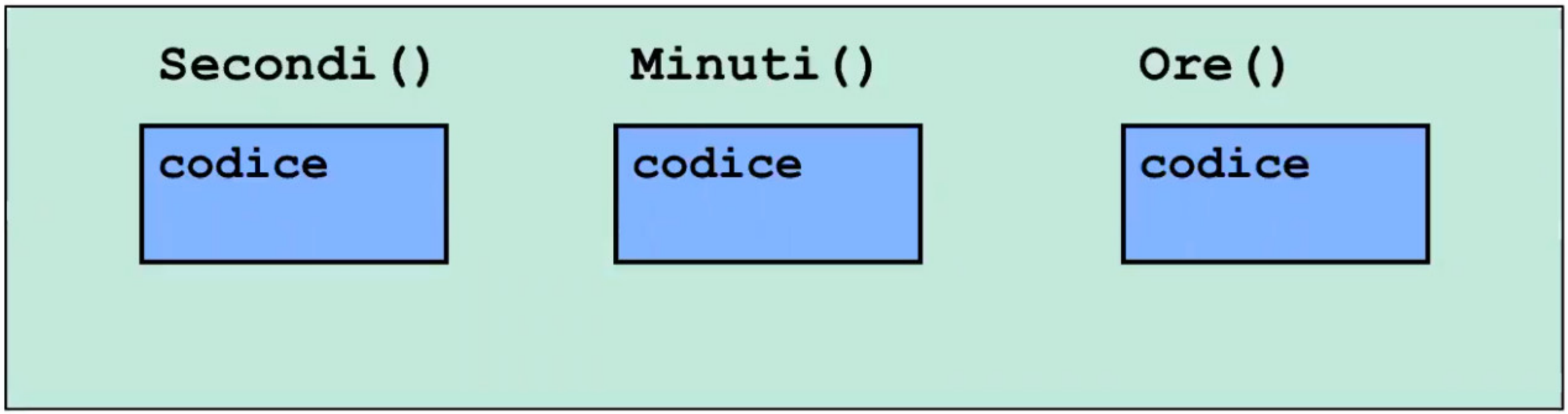
Ogni oggetto memorizza i propri valori dei campi dati



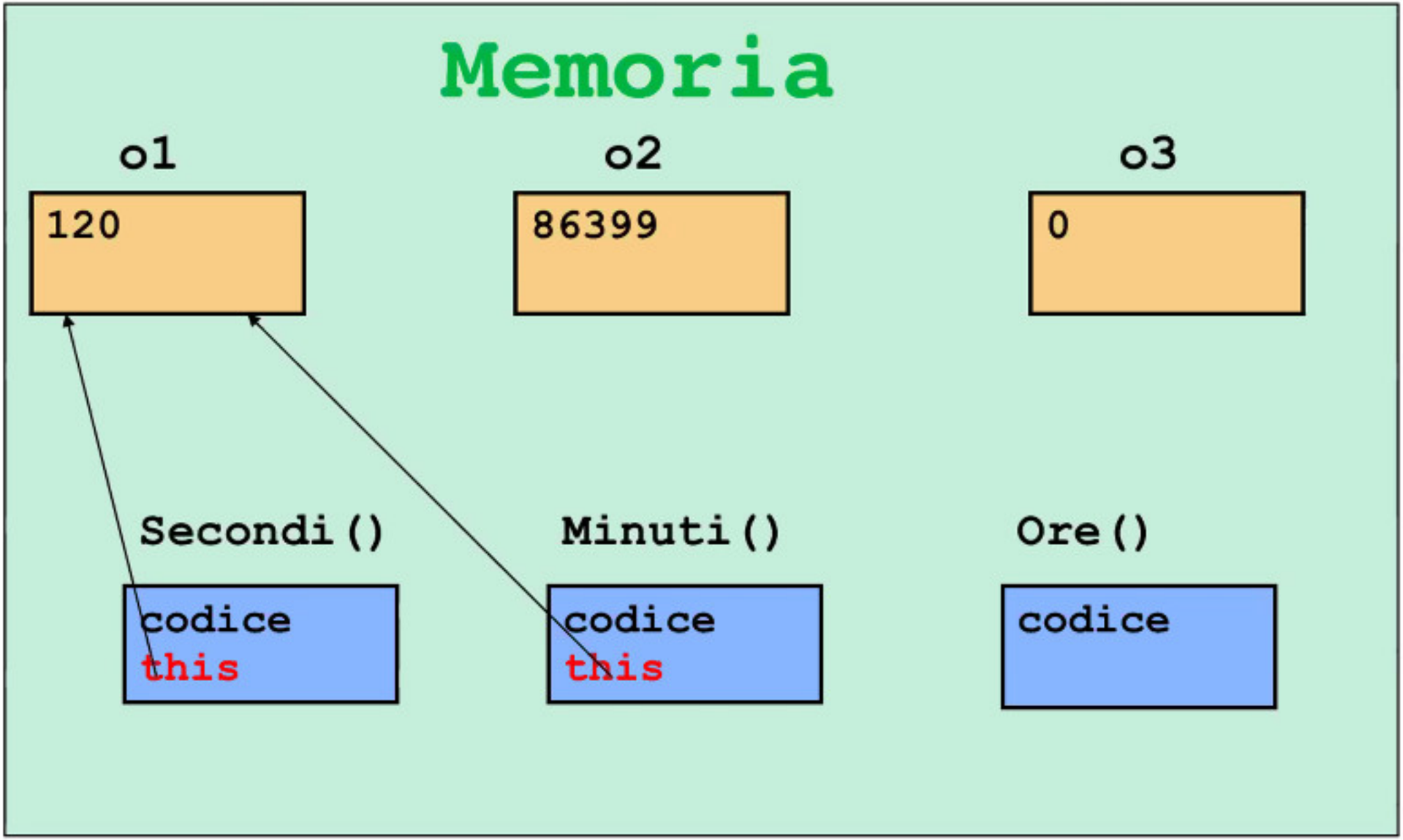
Ogni oggetto memorizza i propri valori dei campi dati




Unica copia in memoria dei metodi per tutti gli oggetti



# Legame tra oggetto di invocazione di tipo C e metodo invocato: parametro implicito **this** di tipo **C***



Possiamo "esplicitare" il parametro implicito **`this`** nella dichiarazione del metodo **`Secondi()`** e nella sua chiamata:

```
// la definizione
int orario::Secondi() { return sec % 60; }
// esplicitando il parametro this diventerebbe
int orario::Secondi(orario* this) {
                                  return (*this).sec % 60;
}

// mentre la chiamata
int s = mezzanotte.Secondi();
// esplicitando il parametro implicito diventerebbe
int s = Secondi(&mezzanotte);
```

`this` è una keyword

Dereferenziazione `*this` nel corpo di un metodo

Ad esempio:

```
int orario::Secondi() {
  return (*this).sec % 60;
}
```

```
int orario::Secondi() {
  return this->sec % 60;
}
```

A volte l'utilizzo esplicito del puntatore **`this`** diviene necessario nella definizione di qualche metodo.

```
class A {
 private:
    int a;
 public:
    A f();
}

A A::f() {
   a = 5;
   return *this;
}
```

# Information hiding

From Wikipedia, the free encyclopedia

In computer science, **information hiding** is the principle of segregation of the *design decisions* in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

Written another way, information hiding is the ability to prevent certain aspects of a class or software component from being accessible to its clients, using either programming language features (like private variables) or an explicit exporting policy.

# Information hiding

From Wikipedia, the free encyclopedia

In computer science, **information hiding** is the principle of segregation of the *design decisions* in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

Written another way, information hiding is the ability to prevent certain aspects of a class or software component from being accessible to its clients, using either programming language features (like private variables) or an explicit exporting policy.

- Specificatori di accesso public e private

- Parte pubblica di una classe (interfaccia pubblica)

- Parte privata di una classe (è il default)

- Interfaccia pubblica come documentazione

Dall'esterno della classe si può accedere **solamente** alla parte pubblica.

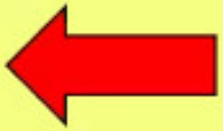
```
orario o;
cout << o.Ore() << endl; // OK: Ore() è pubblica
cout << o.sec << endl;   // Errore: sec è privato
```

Invece, **i metodi della classe ovviamente possono accedere alla parte privata**: non solo a quella dell'oggetto di invocazione ma anche alla parte privata **di qualsiasi altro oggetto della classe** (ad esempio i parametri del metodo).

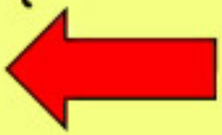
# BACK TO

complesso

```
//complesso.cpp

class complesso {
 private:
  double re, im;
 public:
  void inizializza(double, double);
  double reale();
  double immag();
};

void complesso::inizializza(double r, double i)
{ re = r; im = i; }

double complesso::reale()
{ return re; }

double complesso::immag()
{ return im; }
```



```
//complesso2.cpp

#include<math.h>

// rappresentazione polare: modulo mod e argomento arg
class complesso_pol {
private:
  double mod, arg;
 public:
  void inizializza(double, double);
  double reale();
  double immag();
};

void complesso_pol::inizializza(double r, double i)
{ mod = sqrt(r*r + i*i); arg = atan(i/r); }

double complesso_pol::reale()
{ return mod*cos(arg); }

double complesso_pol::immag()
{ return mod*sin(arg); }
```



```
orario mezzanotte;
mezzanotte.sec = 0; // Errore: sec è privato
```

# Sarà la mezzanotte o no?

```
orario mezzanotte;
mezzanotte.sec = 0; // Errore: sec è privato
```

```
orario mezzanotte;
mezzanotte.sec = 0; // Errore: sec è privato
```

# Constructor (object-oriented programming)

From Wikipedia, the free encyclopedia

In class-based object-oriented programming, a **constructor** in a class is a special type of subroutine called to create an object. It prepares the new object for use, often accepting arguments that the constructor uses to set required member variables.

I costruttori sono (tipicamente) dichiarati nella parte pubblica di una classe

```
class orario {
public:
  orario(); // costruttore senza parametri
  ...       // detto anche costruttore di default
};
```

```
orario::orario() {  // definizione del costruttore
  sec = 0;          // di default
}
```

```
orario mezzanotte;  // viene invocato il
                    // costruttore di default
cout << mezzanotte.Ore() << endl; // stampa 0
```

Si possono definire più costruttori purché differiscano nella lista dei parametri, ovvero nel numero e/o tipo dei parametri. Si fa quindi **overloading** dell'identificatore del metodo che ha lo stesso nome della classe

```
class orario {
public:
  orario();          // costruttore di default
  orario(int,int); // costruttore ore-minuti
  orario(int,int,int); // ore-minuti-secondi
  ...
};
```

```
orario::orario() {  // costruttore di default
  sec = 0;
};
```

```
orario::orario(int o, int m) {
  if (o < 0 || o > 23 || m < 0 || m > 59)
    sec = 0;
  else sec = o * 3600 + m * 60;
};
```

```
orario::orario(int o, int m, int s) {
  if (o < 0 || o > 23 || m < 0 || m > 59 ||
      s < o || s > 59) sec = 0;
  else sec = o * 3600 + m * 60 + s;
};
```

Esempi d'uso dei costruttori:

```
orario adesso_preciso(14,25,47);
     // usa il costruttore ore-minuti-secondi
orario adesso(14,25);
     // usa il costruttore ore-minuti
orario mezzanotte;
     // usa il costruttore senza parametri
orario mezzanotte2;
    // usa il costruttore senza parametri
orario troppo(27,25);
     // usa il costruttore ore-minuti
```

```
cout << adesso_preciso.Secondi() << endl;
                                        // stampa 47
cout << adesso.Minuti() << endl;
                                        // stampa 25
cout << mezzanotte.Ore() << endl;
                                        // stampa 0
cout << troppo.Ore() << endl;
                                        // stampa 0
```

I costruttori si possono anche invocare nel seguente modo:

```
orario adesso_preciso = orario(14,25,47);
orario adesso = orario(14,25);
orario mezzanotte = orario();
```

I costruttori si possono anche invocare nel seguente modo:

```
orario adesso_preciso = orario(14,25,47);
orario adesso = orario(14,25);
orario mezzanotte = orario();
```

Si tratta in questo caso di invocazioni del **costruttore di copia**

### Copy constructors [edit]

Copy constructors define the actions performed by the compiler when copying class objects. A copy constructor has one formal parameter that is the type of the class (the parameter may be a reference to an object). It is used to create a copy of an existing object of the same class. Even though both classes are the same, it counts as a conversion constructor

Nella seguente assegnazione:

```
orario t;
t = orario(12,33,25);
```

il costruttore a tre parametri crea un cosiddetto **oggetto temporaneo anonimo** (cioè senza l-value, quindi non indirizzabile) della classe `orario`.

Un'espressione come `orario(12,33,25)` può essere considerata come un "valore" del tipo `orario`. È un r-valore non indirizzabile.

## Assignment: l-values and r-values [edit]

Some languages use the idea of **l-values** and **r-values**. L-values have storage addresses that are programmatically accessible to the running program (e.g., via some address-of operator like "&" in C/C++), meaning that they are variables or dereferenced references to a certain memory location. R-values can be l-values (see below) or non-l-values—a term only used to distinguish from l-values. Consider the C expression (4 + 9). When executed, the computer generates an integer value of 13, but because the program has not explicitly designated where in the computer this 13 is stored, the expression is an r-value. On the other hand, if a C program declares a variable x and assigns the value of 13 to x, then the expression (x) has a value of 13 and is an l-value.

Viene invocato un costruttore anche quando si crea dinamicamente un oggetto sullo heap con la **new**.

```
orario* ptr = new orario;
orario* ptr1 = new orario(14,25);

cout << ptr->Ore() << endl;   // stampa 0
cout << ptr1->Ore() << endl; // stampa 14
```

# Riassunto della classe orario.

```
class orario {
public:
  orario();                 // costruttore di default
  orario(int,int);          // costruttore ore-minuti
  orario(int,int,int); // costruttore ore-minuti-secondi
  int Ore();                // selettore delle ore
  int Minuti();             // selettore dei minuti
  int Secondi();            // selettore dei secondi
private:
  int sec;                  // campo dati
};
```