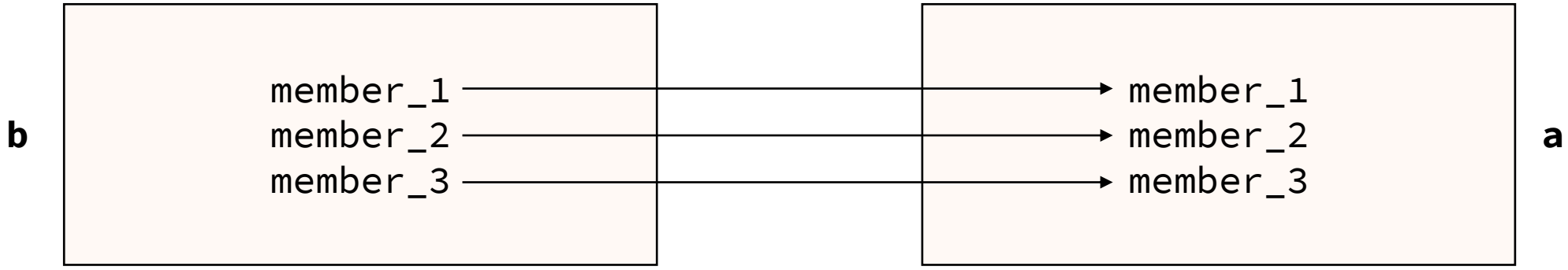
Object-oriented programming (Programmazione ad oggetti) Parte 2

Liceo G.B. Brocchi – Bassano del Grappa (VI)
Liceo Scientifico – opzione scienze applicate
Giovanni Mazzocchin

Copiare gli oggetti

- In C++, di default, copiare un oggetto b in un oggetto a significa copiare i membri di b nei membri di a
- Possiamo costruire un oggetto che è copia di un altro oggetto già esistente
- Il metodo che si occupa di costruire un oggetto a di tipo T come copia di un altro oggetto b di tipo T si chiama costruttore di copia (**copy constructor**)



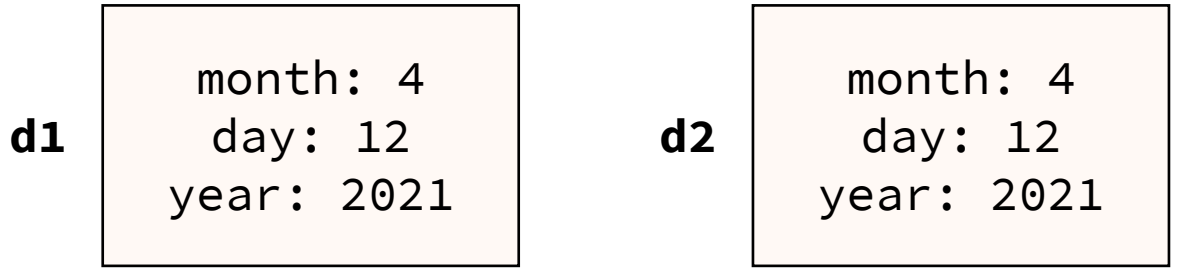
Copiare gli oggetti

```
class Date {
private:
  int month;
  int day;
  int year;
public:
  Date(int m, int d, int y): month(m), day(d), year(y) {}

  int get_month() const {
    return month;
  }
  int get_day() const {
    return day;
  }
  int get_year() const {
    return year;
  }
};
```

```
int main(int argc, char* argv[]) {
  Date d1(4, 12, 2021);
  Date d2(d1);
}
```

dopo le 2 istruzioni del `main`, la situazione in memoria è la seguente:



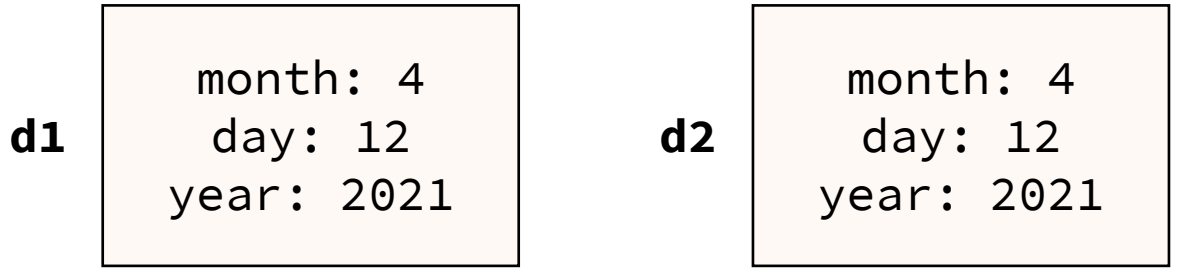
Copiare gli oggetti

```
class Date {
private:
  int month;
  int day;
  int year;
public:
  Date(int m, int d, int y): month(m), day(d), year(y) {}

  int get_month() const {
    return month;
  }
  int get_day() const {
    return day;
  }
  int get_year() const {
    return year;
  }
};
```

```
int main(int argc, char* argv[]) {
  Date d1(4, 12, 2021);
  Date d2(d1);
}
```

chi ha costruito `d2` *di copia* da `d1`?



Copy constructor

- Conosciamo bene il concetto di **costruttore**
- Esiste un costruttore, detto **copy constructor**, che si occupa di costruire un oggetto *di copia* da un oggetto esistente
- Il copy constructor con il comportamento standard (copia membro a membro) è già presente
- Il suo prototipo è `T(const T&);`
- Come tutti i costruttori, ha un nome uguale al nome della classe di cui fa parte
- Proviamo a ridefinirlo per capire quando viene invocato

Copy constructor

```
Date(const Date& d) {
  this->month = d.month;
  this->day = d.day;
  this->year = d.year;
  cout << "called Date copy constructor" << endl;
}
```

in questo modo, quando viene creato un oggetto di tipo `Date` di copia da un altro oggetto di tipo `Date`, verrà stampato `called Date copy constructor`

Operator overloading

- In C++ è possibile ridefinire quasi tutti gli operatori per gli user-defined types
- Esempio: sarebbe utile se una classe che rappresenta un punto sul piano cartesiano avesse un metodo `sum` che effettua la somma componente per componente del punto oggetto di invocazione e un punto passato come parametro
- Invece di utilizzare il nome `sum`, si potrebbe utilizzare l'usuale operatore +, che tuttavia di default è definito solo per i tipi numerici
- Dovremmo quindi *sovraccaricare* il simbolo + del significato: *somma di 2 punti componente per componente*

Operator overloading

```
class point {
private:
  double x;
  double y;
public:
  point(double x, double y): x(x), y(y) {
  }
  point operator+(const point& p) {
    return point(this -> x + p.x, this -> y + p.y);
  }
};
```

```
int main(int argc, char* argv[]) {
  point p1(7, 6);
  point p2(1, 3);
  point p3 = p1 + p2;
}
```

Membri `static`

- All'interno di una classe, un membro dichiarato `static` non dipende dalla singola istanza della classe
- Esisterà quindi una sola copia di un membro `static`, indipendentemente dal numero di istanze della classe
- I membri non `static` (quelli che conosciamo già) sono detti **membri d'istanza**
- Vediamo la differenza tra membri `static` e membri d'istanza analizzandone gli indirizzi

Membri static

```
class C {
public:
  static int x;  //declaration
  int y;
};

int C::x = 0;  //initialization

int main(int argc, char* argv[]) {
  C c;
  c.x = 12;
  cout << &c.x << endl;
  cout << &c.y << endl;
  C c1;
  cout << &c1.x << endl;
  cout << &c.y << endl;
  C c2;
  cout << &c2.x << endl;
  cout << &c.y << endl;
}
```