Programmazione 2
e Laboratorio di Programmazione

Corso di Laurea in

Informatica

Università degli Studi di Napoli "Parthenope"

Anno Accademico 2023-2024

Prof. Luigi Catuogno

1

Informazioni sul corso

Docente	Luigi Catuogno **luigi.catuogno@uniparthenope.it**
Orario	Lun: 9:00-11:00 Mer: 11:00-13:00
Sede	Centro Direzionale Napoli **Aula Magna**
Ricevimento	Mer: 14:00-16:00 (previo appuntamento) Ufficio docente oppure Team: **cxxa3bo**

2

Libri di testo

Introduzione al linguaggio – costrutti e tecniche di base

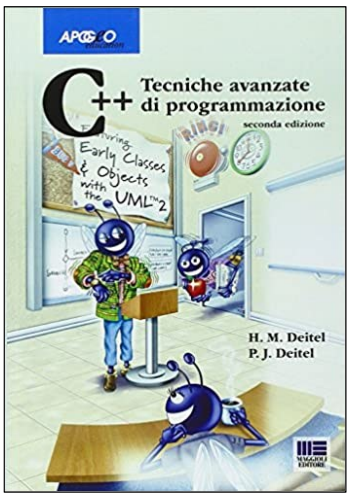
[FdP] H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

3

Libri di testo

Tecniche avanzate e strutture dati elementari

[TAP] H. M. Deitel, P. J. Deitel
C++ Tecniche avanzate di programmazione
II ed. (2011) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8572-6



4

Risorse on-line

Team del corso

Programmazione 2 AA 2023-24 - Prof. Catuogno
Comunicazioni, incontri e avvisi per il corso
Codice: **ftomzjx**

Piattaforma e-learning

Programmazione II e Laboratorio di Programmazione II - A.A. 2023-24
Materiale didattico, manualistica, esercitazioni.
URL: https://elearning.uniparthenope.it/course/view.php?id=2386

5

Strutture dati elementari

6

Liste a doppi puntatori

7

Liste *a doppi puntatori*

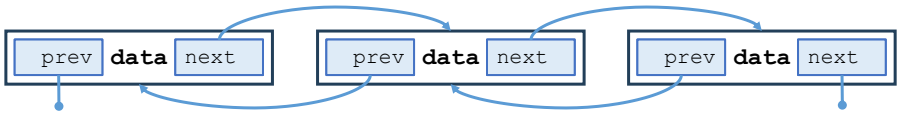
- Nelle liste a doppi puntatori, ciascun nodo possiede un puntatore al nodo che lo segue nella lista e un altro al nodo che la precede.
  - Più efficienti per implementare scansioni e ricerche all'interno di sequenze di dati.

8

Esempio: lista a *doppi puntatori*

File: **doubleList.hpp**

```
…
12  class doubleNode {
13  public:
14          int data;
15          doubleNode *next;
16          doubleNode *prev;
17
18          doubleNode(int i): next(nullptr), prev(nullptr), data(i) {};
19          ~doubleNode(){};
20  };
…
```

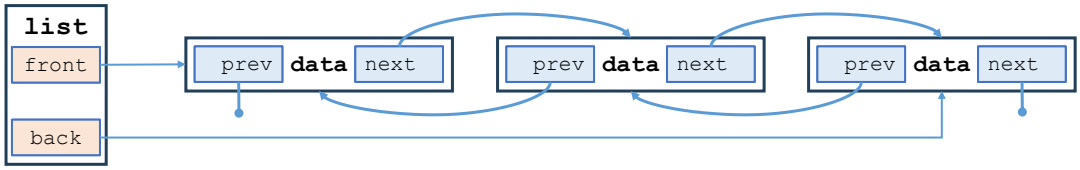


9

Esempio: lista a *doppi puntatori*

File: **doubleList.hpp**

```
…
23  class doubleList {
24  protected:
25          doubleNode *front;
26          doubleNode *back;
27  public:
28          doubleList():front(nullptr), back(nullptr) {};
29          virtual ~doubleList();
30          virtual doubleList *insert(int); //at back
32          virtual doubleList *remove(int&); //from front
32          bool isEmpty();
33  };
```



10

Esempio: lista a *doppi puntatori*

File: **doubleList.cpp**

```
…
4  bool doubleList::isEmpty(){
5          return front==nullptr;
6  }
7
8  doubleList::~doubleList() {
9          doubleNode *tmp;
10         while(front!=nullptr) {
11                 tmp=front;
12                 front=front->next;
13                 delete tmp;
14         }
15 }
```

…questi operatori non riserva «sorprese»…

11

Esempio: lista a *doppi puntatori*

File: **doubleList.cpp**

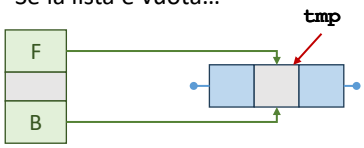
```
…
16 doubleList *doubleList::insert(int item) { //at back
17         doubleNode *tmp;
18         tmp=new doubleNode(item);
19
20         if (isEmpty())
21                 front=back=tmp;
22         else {
23                 tmp->prev=back;
24                 back->next=tmp;
25                 back=tmp;
26         }
27         return this;
28 }
…
```

12

Esempio: lista a *doppi puntatori*

```
 …  …
16  doubleList *doubleList::insert(int item) {
17          doubleNode *tmp;
18          tmp=new doubleNode(item);
19
20          if (isEmpty())
21                  front=back=tmp;
22          else {
23                  tmp->prev=back;
24                  back->next=tmp;
25                  back=tmp;
26          }
27          return this;
28  }
 …  …
```

Se la lista è vuota…

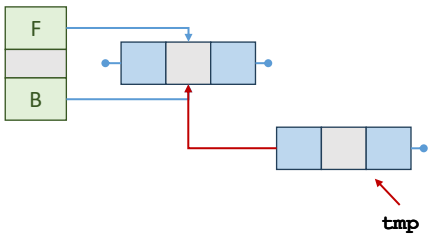


13

Esempio: lista a *doppi puntatori*

```
 …  …
16  doubleList *doubleList::insert(int item) {
17          doubleNode *tmp;
18          tmp=new doubleNode(item);
19
20          if (isEmpty())
21                  front=back=tmp;
22          else {
23                  tmp->prev=back;
24                  back->next=tmp;
25                  back=tmp;
26          }
27          return this;
28  }
 …  …
```

Altrimenti…

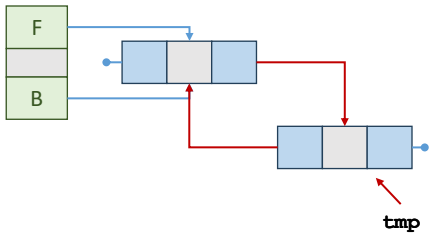


14

Esempio: lista a *doppi puntatori*

```
 …  …
16  doubleList *doubleList::insert(int item) {
17          doubleNode *tmp;
18          tmp=new doubleNode(item);
19
20          if (isEmpty())
21                  front=back=tmp;
22          else {
23                  tmp->prev=back;
24                  back->next=tmp;
25                  back=tmp;
26          }
27          return this;
28  }
 …  …
```

Altrimenti…

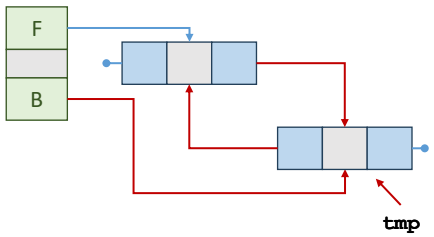


15

Esempio: lista a *doppi puntatori*

```
 …  …
16  doubleList *doubleList::insert(int item) {
17          doubleNode *tmp;
18          tmp=new doubleNode(item);
19
20          if (isEmpty())
21                  front=back=tmp;
22          else {
23                  tmp->prev=back;
24                  back->next=tmp;
25                  back=tmp;
26          }
27          return this;
28  }
 …  …
```

Altrimenti…



16

Esempio: lista a *doppi puntatori*

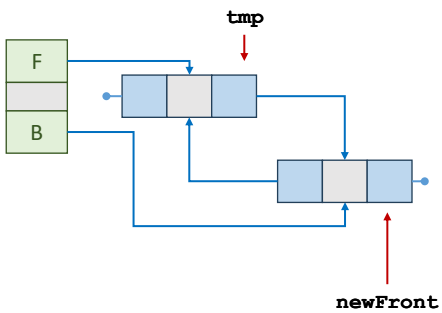
File: **doubleList.cpp**

```
 …  …
29  doubleList *doubleList::remove(int &item) { // from front
30          doubleNode *tmp,*newFront;
31          if(isEmpty()) {
32                  // Gestione dell'errore!!
33          }
34          tmp=front;
35          if(front==back) {
36                  front=back=nullptr;
37          } else {
38                  newFront=front->next;
39                  newFront->prev=nullptr;
40                  front=newFront;
41          }
42          item=tmp->data;
43          delete tmp;
44          return this;
45  }
```


Esempio: lista a *doppi puntatori*

```
 …  …
29  doubleList *doubleList::remove(int &item)
30          doubleNode *tmp,*newFront;
31          if(isEmpty()) {
32                  // Gestione dell'errore!!
33          }
34          tmp=front;
35          if(front==back) {
36                  front=back=nullptr;
37          } else {
38                  newFront=front->next;
39                  newFront->prev=nullptr;
40                  front=newFront;
41          }
42          item=tmp->data;
43          delete tmp;
44          return this;
45  }
```

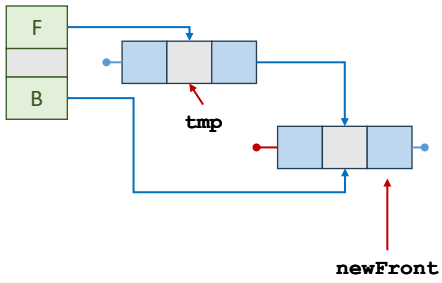
Lista con più di un nodo…



Esempio: lista a *doppi puntatori*

```
 …  …
29  doubleList *doubleList::remove(int &item)
30          doubleNode *tmp,*newFront;
31          if(isEmpty()) {
32                  // Gestione dell'errore!!
33          }
34          tmp=front;
35          if(front==back) {
36                  front=back=nullptr;
37          } else {
38                  newFront=front->next;
39                  newFront->prev=nullptr;
40                  front=newFront;
41          }
42          item=tmp->data;
43          delete tmp;
44          return this;
45  }
```

Lista con più di un nodo…

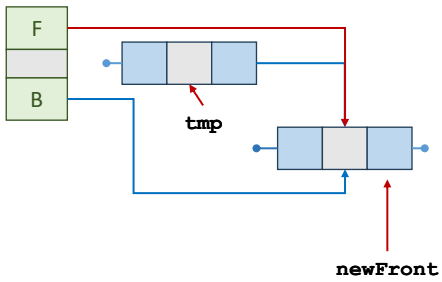


19

Esempio: lista a *doppi puntatori*

```
 …  …
29  doubleList *doubleList::remove(int &item)
30          doubleNode *tmp,*newFront;
31          if(isEmpty()) {
32                  // Gestione dell'errore!!
33          }
34          tmp=front;
35          if(front==back) {
36                  front=back=nullptr;
37          } else {
38                  newFront=front->next;
39                  newFront->prev=nullptr;
40                  front=newFront;
41          }
42          item=tmp->data;
43          delete tmp;
44          return this;
45  }
```

Lista con più di un nodo…

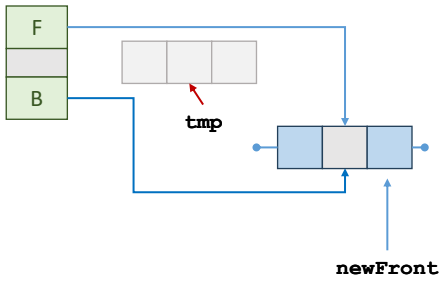


20

Esempio: lista a *doppi puntatori*

```
 …  …
29  doubleList *doubleList::remove(int &item)
30          doubleNode *tmp,*newFront;
31          if(isEmpty()) {
32                  // Gestione dell'errore!!
33          }
34          tmp=front;
35          if(front==back) {
36                  front=back=nullptr;
37          } else {
38                  newFront=front->next;
39                  newFront->prev=nullptr;
40                  front=newFront;
41          }
42          item=tmp->data;
43          delete tmp;
44          return this;
45  }
```

Lista con più di un nodo…



21

Gestione delle *eccezioni*

22

Gestione delle *eccezioni*

- Il verificarsi di certi *«eventi eccezionali»* può portare il programma in una condizione di instabilità e determinarne l'arresto *«incontrollato»*

  per esempio:

  lunghezza di array inammissibile;

  fallimento di un'operazione di I/O su stream;

  errori di overflow nei calcoli aritmetici

  passaggio di argomenti non validi

  …

23

Gestione delle *eccezioni*

- Tali eventi sono detti *eccezioni* e:
  - Costituiscono delle *anomalie* nel funzionamento del programma che possono condurre alla sua interruzione;
  - Talvolta si verificano durante l'esecuzione di codice che non ne è la causa diretta (*e.g.* espressioni composte correttamente, i cui operandi a *runtime* risultano inammissibili);
  - Le operazioni da compiere per risolvere il problema, o per contenerne gli effetti deleteri, possono essere generiche, indipendenti dal codice in esecuzione o praticabili solo a posteriori.

24

Gestione delle *eccezioni*

- Il C++ fornisce un meccanismo di gestione delle eccezioni che *separa il codice che le rileva le gestisce* dal resto del programma.
- Una *eccezione* è rappresentata con un oggetto.
  - Quando l'eccezione è *lanciata (o sollevata)*, il controllo e l'oggetto passano a un blocco di codice designato: il gestore (*handler*)
  - Ciascun *handler* specifica il tipo di eccezione che può gestire;
  - L'*handler* prende in carico l'oggetto e effettua le istruzioni richieste dal caso, quindi restituisce il controllo al programma oppure…

25

Gestione delle *eccezioni*

- Gli oggetti che descrivono le diverse eccezioni sono tutti derivati da una generica classe **`exception`** definita nell'header **`exception.hpp`**
  - La classe espone un'interfaccia relativamente semplice, composta da due costruttori, il distruttore e il metodo **`what()`** che restituisce una descrizione testuale dell'eccezione
  - Dalla classe exception derivano classi che descrivono eventi via via più dettagliati (*e.g.* **`runtime_error`**, **`bad_alloc`**…)
  - E' possibile definire nuove eccezioni derivando dalla classe base.

26

Gestione delle *eccezioni*

Le eccezioni possono essere gestite se occorrono in un blocco di codice specifico, delimitato dalla keyword **`try`**

```
#include<exception>
…
try {
      … codice …
}…
```

27

Gestione delle *eccezioni*

Il blocco **`try`** è seguito da uno o più handler che sono costituiti da blocchi delimitati dalla keyword **`catch`**

```
#include<exception>
try {
      … codice …
}
catch (const std::exception &ex) {
      … codice di gestione …
}
```

28

Esempio: gestione delle *eccezioni*

```
 1  #include <iostream>
 2  #include <exception>
 3  using namespace std;
 4
 5  int main () {
 6          int len;
 7          cout<<"inserisci la lunghezza dell'array: ";
 8          cin >> len;
 9          try {
10                  int *myArray= new int[len];
11          }
12          catch (exception& e) {
13                  cout << "Questo e' l'handler"<<endl;
14                  cout << "Exception: " << e.what() << endl;
15      }
16       return 0;
17  }
```

29

Esempio: gestione delle *eccezioni*

Blocco **`try{}`**. Se una eccezione è lanciata da qui: si istanzia un oggetto di una classe **`exception`** (o derivata) e si cerca l'handler che la intercetta:

```
 1  #include <iostream>
 2  #include <exception>
 3  using namespace std;
 4
 5  int main () {
 6          int len;
 7          cout<<"inserisci la lunghezza dell'arra      ';
 8          cin >> len;
 9          try {
10                  int *myArray= new int[len];
11          }
12          catch (exception& e) {
13                  cout << "Questo e' l'handler"<<endl;
14                  cout << "Exception: " << e.what() << endl;
15      }
16       return 0;
17  }
```

30

Esempio: gestione delle *eccezioni*

```
1  #include <iostream>
2  #include <exception>
3  using namespace std;
4
5  int main () {
6          int len;
7          cout<<"inserisci la lunghezza dell'arra
8          cin >> len;
9          try {
10                 int *myArray= new int[len]
11         }
12         catch (exception& e) {
13                 cout << "Questo e' l'handler"<<endl;
14                 cout << "Exception: " << e.what() << endl;
15     }
16      return 0;
17 }
```

Se l'oggetto creato, corrisponde a quello indicato nella *clausola* **`catch`**, il controllo passa al codice in essa contenuto:

31

Esempio: gestione delle *eccezioni*

```
1  #include <iostream>
2  #include <exception>
3  using namespace std;
4
5  int main () {
6          int len;
7          cout<<"inserisci la lunghezza dell'array: ";
8          cin >> len;
9          try {
10                 int *myArray= new int[len];
11         }
12         catch (exception& e) {
13                 cout << "Questo e' l'handler"<<endl;
14                 cout << "Exception: " << e.what() << endl;
15     }
16      return 0;
17 }
```

Se dal blocco precedente non sono state lanciate eccezioni, il codice del/dei blocchi **`catch`** è ignorato.

Il metodo **`what`** visualizza la descrizione dell'eccezione occorsa.

32

Esempio: gestione delle *eccezioni*

```
1  #include <iostream>
2  #include <exception>
3  using namespace std;
4
5  int main () {
6          int len;
7          cout<<"inserisci la lunghezza dell'array: ";
8          cin >> len;
9          try {
10                 int *myArray= new int[len];
11         }
12         catch (exception& e) {
13                 cout << "Questo e' l'ha
14                 cout << "Exception: " <
15     }
16      return 0;
17 }
```

```
$ ./testexception
inserisci la lunghezza dell'array: -1
Questo e' l'handler
Exception: std::bad_array_new_length
$
```


Esempio: gestione delle *eccezioni*

```
1  #include <iostream>
2  #include <string>
3  #include <exception>
4  using namespace std;
5  int main () {
6          int len;
7          string s("0123456789");
8          try {
9                  int i=14;
10                 cout << "String 22:" <<s.at(i)<<endl;
11         }
12         catch (exception &e) {
13                 cout << "Questo e' l'handler"<<endl;
14                 cout << "Exception: " << e.what() << endl;
15         }
16         return 0;
17 }
```

Il tentativo di accedere alla 12ma posizione di una stringa di 10 caratteri lancia un'eccezione di classe **out_of_range**

Esempio: gestione delle *eccezioni*

```
1  #include <iostream>
2  #include <string>
3  #include <exception>
4  using namespace std;
5  int main () {
6          int len;
7          string s("0123456789");
8          try {
9                  int i=14;
10                 cout << "String 22:" <<s.at(i)<<endl;
11         }
12         catch (exception &e) {
13                 cout << "Questo e' l'handler"<<endl;
14                 cout << "Exception: " << e.what() << endl;
15 $ ./testexception2
16 String 22:Questo e' l'handler
17 Exception: basic_string::at: __n (which is 14) >= this->size() (which is 10)
   $
```

Il tentativo di accedere alla 12ma posizione di una stringa di 10 caratteri lancia un'eccezione di classe **out_of_range**

La descrizione può contenere informazioni sulle cause dell'eccezione

35

Gestione delle *eccezioni*

Il blocco **try** può essere seguito da uno o più handler, in caso di eccezione, il controllo passa al primo di questi che dichiara l' oggetto **exception** (o derivato) corrispondente.

```
#include<exception>
try {
        … codice …
}
catch (exception1 &ex) {
        … codice di gestione exception1…
}
catch (exception2 &ex) {
        … codice di gestione exception2…
}
```

36

Gestione delle *eccezioni*

Se il codice del blocco **try** è eseguito senza sollevare eccezioni…

Le clausole **catch** sono ignorate…

Il controllo passa alla prima istruzione che segue l'ultima clausola **catch**

```
try {
        … }
catch (exception1 &e) {
        … }
catch (exception2 &e) {
        … }
catch (exception3 &e) {
        … }
catch(…) {
        … clausola «catch all» …
}
prossima istruzione…
```

37

Gestione delle *eccezioni*

Se il codice del blocco **try** solleva l'eccezione **exception2**

Si cerca (se c'è) una clausola **catch** per quella eccezione (oppure una più «generica»)

Quelle precedenti sono ignorate…

```
try {
        … }
catch (exception1 &e) {
        … }
catch (exception2 &e) {
        … }
catch (exception3 &e) {
        … }
catch(…) {
        … clausola «catch all» …
}
prossima istruzione…
```

Nota bene: le clausole sono disposte in ordine: dall'eccezione più specializzata a quella più generale.

38

Gestione delle *eccezioni*

Se il codice del blocco **try** solleva l'eccezione **exception2**

Quando la clausola **catch** selezionata termina (senza ulteriori eccezioni)…

…quelle successive sono ignorate…

```
try {
        … }
catch (exception1 &e) {
        … }
catch (exception2 &e) {
        … }
catch (exception3 &e) {
        … }
catch(…) {
        … clausola «catch all» …
}
prossima istruzione…
```

39

Gestione delle *eccezioni*

Se il codice del blocco **try** solleva l'eccezione **exception2**

Quando la clausola **catch** selezionata termina (senza ulteriori eccezioni)

Quelle successive sono ignorate…

Il controllo passa alla prima istruzione che segue l'ultima clausola **catch**

```
try {
        … }
catch (exception1 &e) {
        … }
catch (exception2 &e) {
        … }
catch (exception3 &e) {
        … }
catch(…) {
        … clausola «catch all» …
}
prossima istruzione…
```

40

Gestione delle *eccezioni*

Se il codice del blocco **try** solleva l'eccezione **exceptionX**

Si cerca (se c'è) una clausola **catch** per quella eccezione (oppure una più «generica»)

Quelle precedenti sono ignorate…

La clausola **catch(…)** intercetta tutte le eccezioni. Se c'è, deve essere l'ultima e non fornisce un riferimento a una eccezione. Per il resto funziona come le altre.

```
try {
        … }
catch (exception1 &e) {
        … }
catch (exception2 &e) {
        … }
catch (exception3 &e) {
        … }
catch(…) {
        … clausola «catch all» …
}
prossima istruzione…
```

41

Gestione delle *eccezioni*

Se il codice del blocco **try** solleva l'eccezione **exceptionX**

Si cerca (se c'è) una clausola **catch** per quella eccezione (oppure una più «generica»)

Se non si trova nessuna clausola adatta, il controllo passa all'handler di default, che generalmente causa la terminazione del programma.

```
try {
        … }
catch (exception1 &e) {
        … }
catch (exception2 &e) {
        … }
catch (exception3 &e) {
        … }
prossima istruzione…
```

terminate()

42

Gestione delle *eccezioni*

Si noti che in caso di eccezione, il codice del blocco **`try`** è interrotto immediatamente e non sarà ripreso in nessun caso.

Questo è il modello di gestione (delle eccezioni) per *terminazione*.

```
try {
        … }
catch (exception1 &e) {
        … }
catch (exception2 &e) {
        … }
catch (exception3 &e) {
        … }
catch(…) {
        … clausola «catch all» …
}
prossima istruzione…
```

43

Gestione degli errori con le *eccezioni*

- Talvolta, la gestione di condizioni d'errore particolari, può rendere il codice particolarmente complicato e inefficiente. Questo avviene quando:
  - La verifica esaustiva di tutte le possibili condizioni di errore è particolarmente laboriosa e «appesantisce» il codice;
  - Il codice che rileva l'errore è annidato in profondità in una funzione, rendendo problematico restituire il controllo al «chiamante» e notificargli l'evento;
  - Il codice che rileva l'errore e quello che deve gestirlo potrebbero non «comunicare» tra loro (e.g. programmi costruiti con componenti «eterogenee»)
  - …

44

Gestione degli errori con le *eccezioni*

- In questi (e altri) casi, le eccezioni risultano essere un meccanismo efficiente e portabile per gestire condizioni d'errore.
  - Il codice (*e.g.* una funzione) che rileva la condizione d'errore non la gestisce direttamente ma lancia deliberatamente una eccezione definita allo scopo;
  - Il controllo passa all' handler definito per quella determinata eccezione, che la gestisce *indipendentemente* dal codice che l'ha lanciata
  - Il codice della funzione e il codice dell'handler sono separati.

45

Gestione degli errori con le *eccezioni*

Una eccezione è lanciata arbitrariamente con l'operatore **`throw`**. Se questo è contenuto in un blocco **`try`**, l'eccezione lanciata può essere servita da uno dei blocchi **`catch`** che lo seguono;

```
#include<exception>
try {
        …
        throw exception1();
        …
}
catch (exception1 &ex) {
        … codice di gestione exception1…
}
…
```

46

Gestione degli errori con le *eccezioni*

Il programma può lanciare, mediante **`throw`**, eccezioni già definite dalle librerie C++ oppure eccezioni definite dall'utente, derivandole da quelle standard.

```
#include<exception>
class MyException1: public exception1 {
…
};
try {
        …
        throw MyException1();
        …
}
catch (exception1 &ex) {
        … codice di gestione exception1…
}
```

47

Esempio: lista a *doppi puntatori*

File: **`doubleList.cpp`**

```
 …  …
 29  doubleList *doubleList::remove(int &item) { // from front
 30          doubleNode *tmp,*newFront;
 31          if(isEmpty()) {
 32                  // Gestione dell'errore!!
 33          }
 34          tmp=front;
 35          if(front==back) {
 36                  front=back=nullptr;
 37          } else {
 38                  newFront=front->next;
 39                  newFront->prev=nullptr;
 40                  front=newFront;
 41          }
 42          item=tmp->data;
 43          delete tmp;
 44          return this;
 45  }
```

Da specifica, il metodo è scritto per restituire il puntatore all'oggetto stesso. Restituire nullptr se la lista è vuota, può casusare errori a runtime in espressioni del tipo:

```
doubleList *L;
L->insert(i1)->remove(i2)->remove(i3)->insert(i2);
```

48

Esempio: lista a *doppi puntatori*

File: **doubleList.cpp**

```
…
29 …
30 doubleList *doubleList::remove(int &item) { // from front
31         doubleNode *tmp,*newFront;
32         if(isEmpty()) {
33                 // Gestione dell'errore!!
34         }
35         tmp=front;
36         if(front==back) {
37                 front=back=nullptr;
38         } else {
39                 newFront=front->next;
40                 newFront->prev=nullptr;
41                 front=newFront;
42         }
43         item=tmp->data;
44         delete tmp;
45         return this;
   }
```

In caso di ritorno anticipato, il valore di **item** non è specificato e potrebbe propagarsi nel codice chiamante creando inconsistenze etc.

Assegnare un valore di default (*e.g.* **item=0**) in caso di errore non è sempre praticabile…

49

Esempio: lista a *doppi puntatori*

File: **doubleList.hpp**

La classe **runtime_error** è derivata da **exception** e descrive una gerarchia di errori vari che si verificano a runtime

Il suo costruttore prende come parametro la stringa di descrizione che sarà restituita dal metodo **what()**

```
…
3  …
4  #include<stdexcept>
5  using std::runtime_error;
6
7  class listIsEmpty : public runtime_error {
8  public:
9          listIsEmpty(): runtime_error("Errore: lista vuota"){};
   };
…  …
```

50

Esempio: lista a *doppi puntatori*

File: **doubleList.cpp**

```
…
doubleList *doubleList::remove(int &item) { // from front
        doubleNode *tmp,*newFront;
        if(isEmpty()) {
                throw listIsEmpty();
        }
        tmp=front;
        if(front==back) {
                front=back=nullptr;
        } else {
                newFront=front->next;
                newFront->prev=nullptr;
                front=newFront;
        }
        item=tmp->data;
        delete tmp;
        return this;
}
```

Lancia una eccezione **listIsEmpty()**: la funzione è interrotta senza restituire il controllo al chiamante (e senza passargli valori inconsistenti);

Il programma cerca l'handler per l'eccezione…

51

Esempio: lista a *doppi puntatori*

File: **doubleListMain.cpp**

La chiamata **x.remove(item)** è inserita in un blocco **try** in caso di eccezione, il controllo passa all'handler successivo che, in questo caso, avvisa l'utente con il messaggio d'errore.

```
…
        case 'r':
                try {
                        x.remove(item);
                        cout << "item: "<<item<<endl;
                } catch (listIsEmpty &ex) {
                        cout<<endl<<"Errore: "<<ex.what()<<endl<<endl;
                }
                break;
        case 'e':
…
```

52

…tornando alle liste…

53

Esempio: scansione di una lista…

File: **doubleList.hpp**

```
…
class doubleList2: public doubleList {
private:
        doubleNode *current;
public:
        doubleList2(): current(nullptr) {};
        ~doubleList2() {};
        int showCurrent();
        doubleList2 *forward();
        doubleList2 *backward();
        doubleList2 *begin();
        doubleList2 *end();
        bool atFront();
        bool atBack();
        doubleList *remove(int&) override;
        doubleList2 *insertNext(int);
        doubleList2 *removeCurrent(int&);
};
```

La chiamata **current** punta all'elemento della lista attualmente «all'attenzione» del programma. d'errore.

Restituisce l'intero contenuto nel nodo corrente (*e.g.* puntato da **current**)

54

Esempio: scansione di una lista…

File: **doubleList.hpp**

```
…  …
34 class doubleList2: public doubleList {
35 private:
36         doubleNode *current;
37 public:
38         doubleList2(): current(nullptr) {};
39         ~doubleList2() {};
40         int showCurrent();
41         doubleList2 *forward();
42         doubleList2 *backward();
43         doubleList2 *begin();
44         doubleList2 *end();
45         bool atFront();
46         bool atBack();
47         doubleList *remove(int&) override;
48         doubleList2 *insertNext(int);
49         doubleList2 *removeCurrent(int&);
50 };
```

Sposta **current** in avanti (dalla testa verso la coda) di un nodo per volta;

Uguale al precedente, ma sposta **current** in senso opposto;

55

Esempio: scansione di una lista…

File: **doubleList.hpp**

```
…  …
34 class doubleList2: public doubleList {
35 private:
36         doubleNode *current;
37 public:
38         doubleList2(): current(nullptr) {};
39         ~doubleList2() {};
40         int showCurrent();
41         doubleList2 *forward();
42         doubleList2 *backward();
43         doubleList2 *begin();
44         doubleList2 *end();
45         bool atFront();
46         bool atBack();
47         doubleList *remove(int&) override;
48         doubleList2 *insertNext(int);
49         doubleList2 *removeCurrent(int&);
50 };
```

Spostano **current** rispettivamente in posizione **front** e **back**;

Restituiscono **true** se **current** coincide rispettivamente con **front** e **back**;

56

Esempio: scansione di una lista…

File: **doubleList.cpp**

```
…  …
51  int doubleList2::showCurrent(){
52          if(isEmpty())
53                  throw listIsEmpty();
54
55          if(current==nullptr)
56                  current=front;
57          return current->data;
58  }
59
60  bool doubleList2::atFront() {
61          return current==front;
62  }
63
64  bool doubleList2::atBack() {
65          return current==back;
66  }
```

57

Esempio: scansione di una lista…

File: **doubleList.cpp**

```
68  doubleList2 *doubleList2::forward() {
69          if(isEmpty())
70                  throw listIsEmpty();
71          if(current==nullptr)
72                  current=front;
73          if(!atBack())
74                  current=current->next;
75          return this;
76  }
77  doubleList2 *doubleList2::backward() {
78          if(isEmpty())
79                  throw listIsEmpty();
80          if(current==nullptr)
81                  current=front;
82          if(!atFront())
83                  current=current->prev;
84          return this;
85  }
```

58

Esempio: scansione di una lista…

File: **doubleList.cpp**

```
 86 doubleList2 *doubleList2::begin() {
 87         if(isEmpty())
 88                 throw listIsEmpty();
 89         current=front;
 90         return this;
 91 }
 92 doubleList2 *doubleList2::end() {
 93         if(isEmpty())
 94                 throw listIsEmpty();
 95         current=back;
 96         return this;
 97 }
 98 doubleList *doubleList2::remove(int &item){
 99         current=nullptr;
100         doubleList::remove(item);
101         return this;
102 }
```