

# Ingegneria del Software

Corso di Laurea in Ingegneria Informatica



## Lezione 7 Specifiche, contratti e documentazione

P. Foggia – N. Capuano


DIEM - Università di Salerno



# Schema

- Specifiche, contratti e incapsulamento
- Documentazione della specifica
- Documentazione automatizzata





# Specifiche, contratti e incapsulamento

# Assertzioni in Java

Per impostazione predefinita, le **asserzioni sono disabilitate in Java** (ovvero, la loro condizione non viene controllata durante l'esecuzione)

- Per abilitarle, è necessario **eseguire il programma** con il flag **-ea o -enableassertions**
- **Esempio:** `java -ea MyProgram`

**Sintassi di base:**

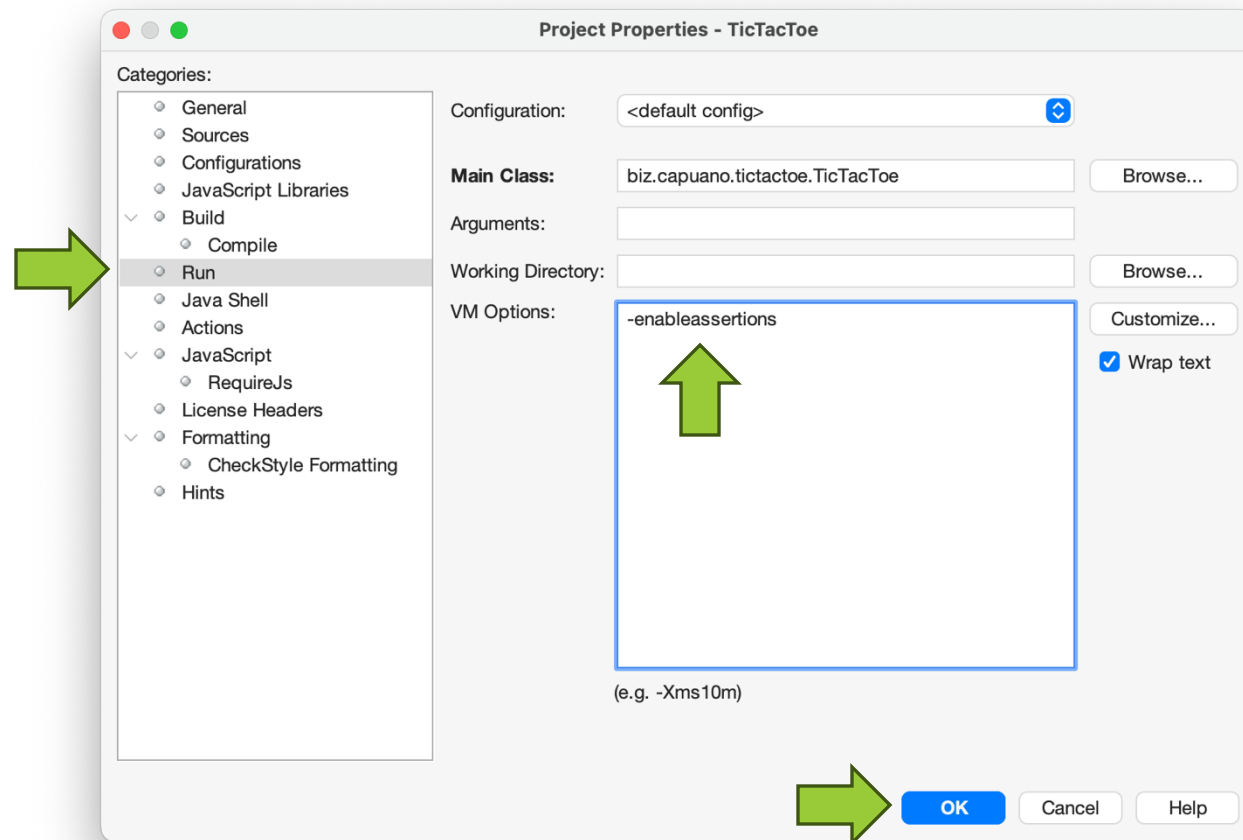
```
assert condition;
```

**Con un messaggio di errore:**

```
assert condition : "Error message";
```

# Abilitare le asserzioni in NetBeans

Selezionare le **proprietà del progetto**, quindi...



# Specifica, contratto e incapsulamento

Nella programmazione Object-Oriented (ad es. in Java) è possibile definire come *privati* alcuni metodi e alcune strutture dati di una classe

- **Domanda:** i metodi e le strutture dati privati fanno parte del "contratto" della classe?
- **Risposta:** il contratto regola i rapporti tra la classe e i suoi client; poiché i client non possono accedere ai membri privati della classe, **questi non fanno parte del contratto / della specifica della classe**

# Specifica, contratto e incapsulamento

- Tuttavia, la metafora del contratto può essere usata, oltre che descrivere la specifica di una classe, anche per aiutare a garantire la correttezza della sua realizzazione
- In un certo senso, la classe deve rispettare due contratti
  - Un contratto "pubblico", verso i clienti della classe, che deve essere definito al momento della specifica della classe
  - Un contratto "privato", che serve a garantire che le varie parti della classe lavorino correttamente tra loro, che deve essere definito al momento della progettazione di dettaglio della classe

# Specifica, contratto e incapsulamento

- Se le **invarianti** riguardano solo la **parte privata** della classe, il meccanismo dell'incapsulamento aiuta a **garantire** che **un client non possa erroneamente violare le invarianti**
  - Se i costruttori garantiscono la verifica delle invarianti al momento della creazione dell'oggetto, e inoltre...
  - ... ogni metodo pubblico garantisce che se le invarianti erano soddisfatte all'inizio del metodo, saranno soddisfatte anche alla fine, allora...
  - ... per conseguenza il client non può portare l'oggetto in uno stato che non soddisfa le invarianti!
  - In questo caso, il rispetto delle invarianti diventa una responsabilità esclusivamente del provider (è uno dei vantaggi dell'incapsulamento)



# Specifica, contratto e incapsulamento

- **Domanda:** Il "contratto privato" di una classe deve essere documentato?
- **Risposta:** Gli elementi del contratto privato possono essere descritti in una documentazione "interna", separata dalla documentazione esterna resa disponibile agli sviluppatori che devono usare la classe. Questa documentazione interna serve solo agli sviluppatori che dovranno modificare la classe.

# Modifica di un componente software

- La **metafora del contratto** può essere utile a capire l'impatto di una modifica di un componente software
- Possiamo domandarci come la modifica cambia i benefici e gli obblighi per i **client**:
  - **Gli obblighi dei client restano uguali o diminuiscono:** la modifica non ha impatto sui client e quindi ha un costo contenuto.
  - **Gli obblighi dei client aumentano:** oltre al codice del componente, bisogna verificare e forse modificare anche il codice dei client. La modifica è potenzialmente costosa.

# Modifica di un componente software

- La metafora del contratto può essere utile a capire l'impatto di una modifica di un componente software
- Possiamo domandarci come la modifica cambia i benefici e gli obblighi per i client:
  - I benefici dei client restano uguali o aumentano: la modifica non ha impatto sui client e quindi ha un costo contenuto.
  - I benefici dei client diminuiscono: oltre al codice del componente, bisogna verificare e forse modificare anche il codice dei client. La modifica è potenzialmente costosa.

# Modifica di un componente software


- **Domanda:** Quando si verifica che gli obblighi dei client aumentano?
- **Risposta:** Quando le precondizioni di un'operazione diventano più "restrittive"
  - Viene aggiunta una precondizione che prima non c'era, oppure
  - una precondizione viene modificata in modo che la nuova versione possa essere falsa in alcune situazioni in cui la vecchia versione era vera

# Modifica di un componente software

- **Domanda:** Quando si verifica che i benefici del client diminuiscono?
- **Risposta:** Quando viene eliminata una delle operazioni previste dal contratto, oppure...
- **Risposta:** ... quando le postcondizioni di un'operazione diventano meno "restrittive"
  - Viene rimossa una postcondizione esistente, oppure
  - una postcondizione viene modificata in modo che la nuova versione possa essere vera in alcune situazioni in cui la vecchia versione era falsa

# Modifica di un componente software

- Quanto più sono gli elementi di una classe che rientrano nel suo "contratto pubblico", tanto maggiori sono le probabilità che una modifica abbia un impatto elevato
- Per questo è una buona idea mantenere nel contratto pubblico solo gli elementi strettamente necessari (un altro vantaggio dell'incapsulamento)



# Documentazione della specifica

# Documentazione della specifica

- La specifica di un componente software deve essere documentata, e la documentazione deve essere accessibile a:
  - Gli sviluppatori che **usano** il componente software (gli sviluppatori dei client)
    - Loro potrebbero non avere accesso al *codice sorgente* del componente software
  - Gli sviluppatori che **realizzano** il componente software
    - Loro dovranno garantire che la progettazione e l'implementazione del componente rispetti quello che dice la specifica
  - Gli sviluppatori che devono **manutenere** il componente software
    - Per loro è importante che il codice sorgente e la documentazione siano "allineati"



# Documentazione della specifica

- Un modo efficace per favorire l'allineamento tra la documentazione e il codice sorgente è di inserire la descrizione della specifica nel codice sorgente (sotto forma di commenti detti *commenti di documentazione*) e usare degli strumenti automatici per *estrarre* dal codice sorgente un documento che descriva la specifica
  - Per gli sviluppatori che usano il componente è importante poter accedere alla documentazione senza bisogno di dover leggere il codice sorgente!
- Vedremo in seguito uno di questi strumenti (Doxygen)

# Cosa inserire nei commenti di documentazione?

- È inutile inserire nei commenti di documentazione le informazioni che possono essere ricavate dai prototipi delle operazioni (che sono nel codice sorgente)
  - I tool che estraggono la documentazione ricavano automaticamente queste informazioni!
- È importante inserire invece le informazioni sul contratto che *non* possono essere ricavate dal prototipo.

# Esempio

```
/* Nome della funzione: calcola_somma  
* Valore di ritorno: nessuno  
* Parametri  
*   array   Tipo: int *  
*   n       Tipo: int  
*   somma   Tipo: int *  
*/
```

```
void calcola_somma(int *array, int n, int *somma);
```

- è un buon commento di documentazione?

# Esempio

```
/* Nome della funzione: calcola_somma
 * Valore di ritorno: nessuno
 * Parametri
 *   array   Tipo: int *
 *   n       Tipo: int
 *   somma   Tipo: int *
 */
void calcola_somma(int *array, int n, int *somma);
```

- No! Il commento riporta solo informazioni che sono già visibili nel prototipo (e quindi **inutili**)!
- No! Il commento non riporta informazioni **necessarie** che non sono visibili nel prototipo!

**Quali informazioni mancano?**

# Esempio

```
/* Calcola la somma di un array di interi  
*/
```

```
void calcola_somma(int *array, int n, int *somma);
```

- Cosa fa la funzione

# Esempio

```
/* Calcola la somma di un array di interi
 * Parametri
 *   array puntatore al primo elemento di un
 *   array di interi
 *   n    numero di elementi dell'array (>= 0)
 *   somma puntatore a una variabile intera che
 *   riceverà il risultato; la variabile
 *   non ha bisogno di essere inizializzata
 */
void calcola_somma(int *array, int n, int *somma);
```

- Cosa deve fornire il chiamante nei parametri (precondizioni)

# Esempio

```
/* Calcola la somma di un array di interi
 * Parametri
 *   array puntatore al primo elemento di un
 *   array di interi
 *   n    numero di elementi dell'array (>= 0)
 *   somma puntatore a una variabile intera che
 *   riceverà il risultato; la variabile
 *   non ha bisogno di essere inizializzata
 * Risultato
 *   Al termine della chiamata, la variabile
 *   *somma conterrà la somma degli elementi
 *   dell'array. La funzione non modifica l'array.
 */
void calcola_somma(int *array, int n, int *somma);
```

- Cosa garantisce la funzione alla fine della chiamata (postcondizioni)

# *Modi dei parametri*

```
void calcola_somma(int *array, int n, int *somma);
```

- Nell'esempio precedente, i due parametri array e somma sono dello **stesso tipo** (int \*), ma **sono usati per realizzare due modalità diverse di comunicazione tra il chiamante e la funzione**
  - array rappresenta un'informazione che viene **definita** dal chiamante prima della chiamata, e viene **usata** dalla funzione
  - somma rappresenta un'informazione che viene **definita** dalla funzione, e viene **usata** dal chiamante dopo la chiamata



# *Modi* dei parametri

- Questa differenza è importante, e fa parte del contratto della funzione, ma (almeno in linguaggi come C e Java) non fa parte della sintassi del linguaggio
  - Perciò deve essere evidenziata nei commenti di documentazione
- Un modo conciso per indicare questa differenza è di indicare per ciascun parametro un *modo*, che rappresenta la "direzione" in cui viaggiano le informazioni descritte da quel parametro

# Modi dei parametri

- Ci sono tre modi possibili, descritti nella seguente tabella

Modo	Direzione delle informazioni	Il chiamante...	La funzione...
Ingresso (In, Input)	chiamante -> funzione	Deve definire l'informazione prima della chiamata. Può assumere che l'informazione non sia modificata dalla chiamata.	Può usare il valore dell'informazione. Non deve modificare il valore dell'informazione (a meno che non lavori su una copia).
Uscita (Out, Output)	funzione -> chiamante	Può usare il valore dell'informazione dopo la chiamata.	Deve definire il valore dell'informazione. Non deve usare il valore dell'informazione prima di averlo definito.
Ingresso/Uscita (In Out, Input Output)	chiamante -> funzione +  funzione -> chiamante	Deve definire l'informazione prima della chiamata. Può usare il valore dell'informazione dopo la chiamata, ma deve assumere che può essere diverso da quello iniziale.	Può usare il valore dell'informazione. Può modificare il valore dell'informazione.

# Esempio

```
/* Calcola la somma di un array di interi
 * Parametri di ingresso
 *   array puntatore al primo elemento di un
 *       array di interi
 *   n    numero di elementi dell'array (>= 0)
 *
 * Parametri di uscita
 *   somma la somma calcolata
 */
void calcola_somma(int *array, int n, int *somma);
```

- L'uso dei modi semplifica la descrizione dei parametri nella documentazione

# Esempio

```
/* Ordina un array di interi interi
 * Parametri di ingresso/uscita
 *   array in ingresso l'array da ordinare; in
 *   uscita l'array ordinato
 *
 * Parametri di ingresso
 *   n    il numero di elemento dell'array (>=0)
 */
void ordina_array(int *array, int n);
```

- L'uso dei modi semplifica la descrizione dei parametri nella documentazione



# Documentazione automatizzata

# Strumenti per automatizzare la documentazione

doxygen

- **Doxygen:** supporta più linguaggi (C, C++, Java, Python)
- **Javadoc:** specifico per Java
- **Sphinx:** specifico per Python



# Come funzionano

## Annotazione del codice:

- Gli sviluppatori aggiungono **annotazioni** nei commenti di documentazione del codice sorgente usando una sintassi specifica (dipendente dallo strumento)
  - Nota per Java: queste annotazioni, che vanno inserite nei commenti, sono qualcosa di diverso dalla *annotazioni Java* che si inseriscono nel codice, e che incontrerete più avanti nel corso di Programmazione ad Oggetti
- Queste annotazioni descrivono **classi, funzioni, parametri** e altri **elementi** (ad esempio, **precondizioni** e **postcondizioni**)

# Come funzionano

## Generazione di documentazione:

- Lo strumento analizza i file sorgente del progetto, e converte queste annotazioni (e le informazioni ricavate dalla sintassi del linguaggio) in uno o più file **documentazione**, pensati per essere consultati agevolmente da un essere umano.

## Formati di uscita:

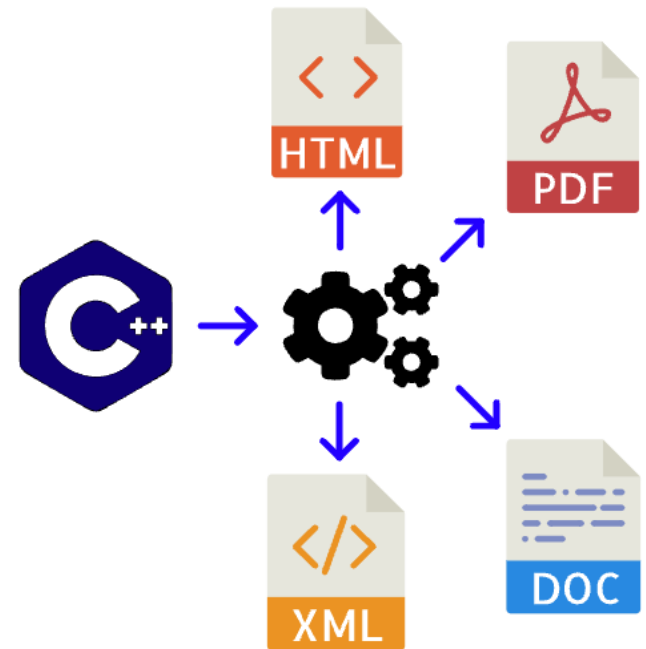
- I **formati** più comuni sono HTML, PDF, XML e Markdown.
- Alcuni strumenti possono anche generare **diagrammi** (ad esempio, alcuni diagrammi UML) e **riferimenti incrociati** per migliorare la chiarezza della documentazione.



# doxygen

**Doxygen** è uno strumento molto utilizzato per **generare documentazione** dal codice sorgente annotato.

- Popolare nelle comunità di programmazione **C**, **C++** e **Java**
- Supporta anche **diversi altri linguaggi di programmazione**
- Può generare documentazione in vari formati di output, come **HTML**, **PDF** (tramite LaTeX), **Word** (tramite RTF) e **XML**.



# Installazione

- **Windows:** Scaricare il programma di installazione da <https://www.doxygen.nl/> ed eseguirlo.
- **macOS:** utilizzare Homebrew: `brew install doxygen`
- **Linux:** Installare tramite il gestore di pacchetti  
`sudo apt-get install doxygen`
- Assicurarsi che il file eseguibile si trovi nel path della shell del sistema operativo
- Mostra la **guida** del programma con il comando:  
`doxygen -h`

# Creare un file di configurazione

Nella **radice della cartella del progetto**, eseguire: `doxygen -g`

- Questo genera il file **Doxyfile** che contiene le opzioni di configurazione

## Impostazioni chiave:

- **PROJECT\_NAME:** Nome del progetto
- **OPTIMIZE\_OUTPUT\_FOR\_C:** impostare su YES per ottimizzare la documentazione per C
- **OPTIMIZE\_OUTPUT\_JAVA:** impostare YES per ottimizzare per Java.
- **FILE\_PATTERNS:** File da documentare (ad esempio, \*.java, \*.c, \*.h)
- **RECURSIVE:** impostare su YES per scansionare anche le sottodirectory.
- **GENERATE\_HTML:** Impostare su YES per generare documentazione HTML
- **GENERATE\_LATEX:** impostare NO a meno che non si abbia bisogno di documentazione in formato LaTeX.
- **HIDE\_UNDOC\_MEMBERS:** impostare a YES per nascondere i membri di strutture o classi che non hanno un commento di documentazione.

# Commenti di documentazione

Doxygen richiede che i commenti di documentazione abbiano una forma particolare, per distinguerli dagli altri commenti

- **Commenti a riga singola:**

```
/// This is a single-line comment for the following entity.  
void myFunction();
```

- **Commenti su più righe:**

```
/**  
 * This is a multi-line comment.  
 * It can span multiple lines and is typically used for more detailed descriptions.  
 */  
void myFunction();
```

- **Commenti in linea:**

```
int myVariable; ///  
This is an in-line comment for a variable.
```

# Documentazione a livello di file

All'**inizio di ogni file**, è necessario includere un commento di blocco che descriva lo scopo del file.

- Questo può includere annotazioni che specificano il **nome del file**, una **breve descrizione** e altri metadati come l'**autore**, la **data** e la **versione**.

```
/**  
 * @file myfile.c  
 * @brief This file contains the implementation of the XYZ feature.  
 *  
 * More detailed information about the file and its role in the project.  
 *  
 * @author Your Name  
 * @date August 31, 2024  
 * @version 1.0  
 */
```

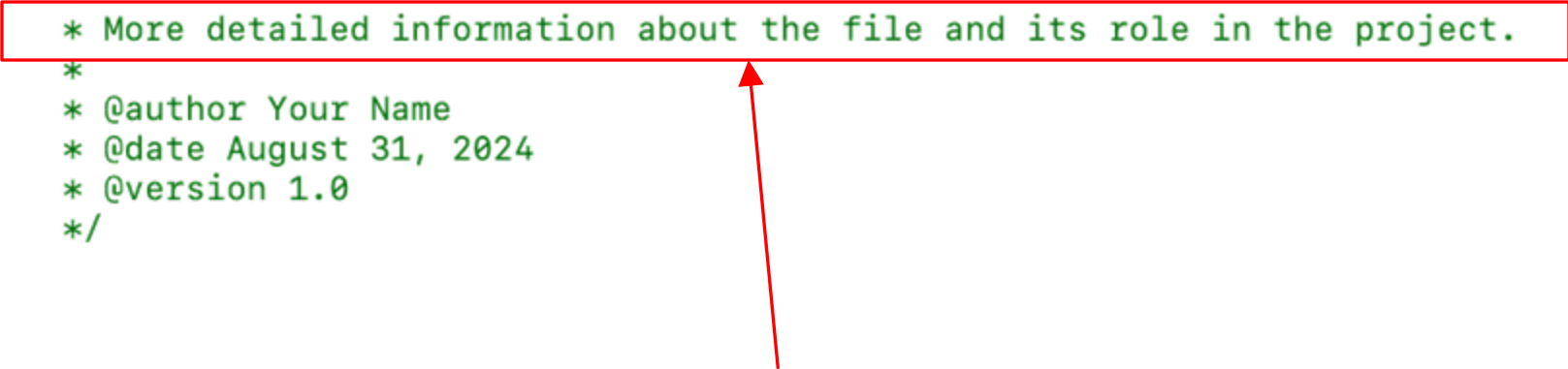
- **@brief** è un esempio di *tag* (marcatore), che introduce una particolare annotazione (il tag @brief aggiunge una "descrizione breve")

# Documentazione a livello di file

All'**inizio di ogni file**, è necessario includere un commento di blocco che descriva lo scopo del file.

- Questo può includere annotazioni che specificano il **nome del file**, una **breve descrizione** e altri metadati come l'**autore**, la **data** e la **versione**.

```
/**
 * @file myfile.c
 * @brief This file contains the implementation of the XYZ feature.
 *
 * More detailed information about the file and its role in the project.
 *
 * @author Your Name
 * @date August 31, 2024
 * @version 1.0
 */
```



- Un commento di documentazione di solito include anche un blocco di testo senza tag, che è usato per aggiungere una descrizione dettagliata dell'elemento commentato.

# Documentazione di una funzione

Ogni **funzione** può essere documentata con una descrizione del suo **scopo**, dei **parametri** che accetta e di ciò che **restituisce**.

- A tale scopo, utilizzare i tag **@param** e **@return**.
- **@param** può specificare anche il modo del parametro: **@param[in]**, **@param[out]** e **@param[inout]**

```
/**  
 * @brief Calculates the factorial of a number.  
 *  
 * This function uses a recursive algorithm to calculate the factorial of a given  
 * non-negative integer.  
 *  
 * @param[in] n The number for which the factorial is to be calculated. Must be non-negative.  
 * @return The factorial of the number.  
 */  
int factorial(int n);
```

# Documentazione di una funzione

Ogni **funzione** può essere documentata con una descrizione del suo **scopo**, dei **parametri** che accetta e di ciò che **restituisce**.

- A tale scopo, utilizzare i tag **@param** e **@return**.
- @param può specificare anche il modo del parametro: @param[in], @param[out] e @param[inout]

```
/**
 * @brief Calculates the factorial of a number.
 *
 * This function uses a recursive algorithm to calculate the factorial of a given
 * non-negative integer.
 *
 * @param[in] n The number for which the factorial is to be calculated. Must be non-negative
 * @return The factorial of the number.
 */
int factorial(int n);
```

@param deve essere seguito dal nome del parametro a cui l'annotazione si riferisce (n in questo caso)



# Precondizioni e postcondizioni

I tag **@pre**, **@post** e **@invariant** possono essere usati per indicare precondizioni, postcondizioni e invarianti

```
/**
 * @brief Calculates the factorial of a non-negative integer.
 *
 * This function computes the factorial of the input integer using a recursive approach.
 *
 * @pre `n >= 0` The input must be a non-negative integer.
 * @post The returned value will be the factorial of the input integer.
 *
 * @param[in] n The non-negative integer whose factorial is to be calculated.
 * @return The factorial of the input integer. If `n == 0`, the return value is 1.
 */
int factorial(int n) {
    assert(n >= 0); // Precondition

    if (n == 0) {
        return 1; // Base case
    }

    int result = n * factorial(n - 1);

    assert(result >= 1); // Postcondition
    return result;
}
```

# Classi e strutture

Quando si documentano classi o strutture, si può includere una breve descrizione (@brief) dello scopo della **classe/struttura**, seguita da **informazioni** più dettagliate su ciascun membro.

```
/**
 * @brief Represents a 2D point in Cartesian coordinates.
 *
 * This struct is used to store the coordinates of a point in a 2D space.
 */
typedef struct {
    double x; ///< The x-coordinate of the point.
    double y; ///< The y-coordinate of the point.
} Point2D;
```

# Riferimenti incrociati

Doxygen permette di fare riferimenti incrociati (link) tra le diverse parti della documentazione, sia usando il tag **@see** che inserendo il nome dell'elemento riferito nel testo di un'altra annotazione.

```
/**
 * @brief Example of a function that uses cross-referencing.
 *
 * See also: add(), subtract()
 *
 * @param[in] a First integer.
 * @param[in] b Second integer.
 * @see add()
 * @see subtract()
 */
int multiply(int a, int b);
```

# Generazione di documentazione

Dopo aver configurato il file **Doxyfile**, si può generare la documentazione eseguendo il comando:

```
doxygen Doxyfile
```

## Altre caratteristiche

- **Diagrammi:** Doxygen può generare diagrammi di classe e di collaborazione se è installato Graphviz.
- **Navigazione nel codice:** È possibile navigare nel codice direttamente dalla documentazione HTML generata.
- **Supporto IDE:** Molti IDE supportano Doxygen, sia in modo nativo che tramite plugin.