


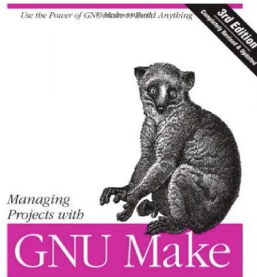
```
g++ -c orario.cpp // compila orario.cpp e non linka  
                  // produce il file orario.o
```



All'utilizzatore basta fornire il file "**orario.h**" da includere in "**main.cpp**" per la compilazione ed il file "**orario.o**" da aggiungere al codice compilato di "**main.cpp**" affinché il linker generi l'eseguibile.

```
g++ main.cpp orario.o // compila main.cpp e linka con  
                      // orario.o
```

Compilazione automatizzata mediante make (e Makefile)



O'REILLY®

Copyrighted Material

Robert M. Koenig

Il **make** è un'**utility**, sviluppata sui **sistemi operativi** della famiglia **UNIX**, ma disponibile su un'ampia gamma di sistemi, che

automatizza il processo di creazione di **file** che dipendono da altri file, risolvendo le **dipendenze** e invocando **programmi** esterni per il lavoro necessario.

La gestione delle dipendenze in **make** è molto semplice, e si basa sulla data e ora di ultima modifica dei file interessati.

L'**utility** è usata soprattutto per la **compilazione** di **codice sorgente** in **codice oggetto**, unendo e poi **linkando** il codice oggetto in programmi **eseguibili** o in **librerie**. Esso usa file chiamati *makefile* per determinare il grafo delle dipendenze per un particolare output, e gli **script** necessari per la **compilazione** da passare alla **shell**. Il termine *makefile* deriva dal nome dato tipicamente al file di input di **make**.

make	
Sviluppatore	Stuart Feldman
Ultima versione	4.0 (9 ottobre 2013)
Sistema operativo	Unix-like
Linguaggio	
Genere	Automazione dello sviluppo
Licenza	GNU General Public License (Licenza libera)
Sito web	www.gnu.org/software/make/ 

Un makefile consiste di alcune regole così descritte:

TARGET : DEPENDENCIES ...
COMMAND

Di solito TARGET è il nome dell'eseguibile o del file oggetto da ricompilare, ma può essere anche una azione (es. **clean**). È una sorta di identificatore dell'azione da compiere: alla chiamata **bash\$ make clean** verrà eseguito il TARGET ``**clean**".

Le DEPENDENCIES sono usate come input per generare l'azione TARGET e di solito sono più di una. Più genericamente vengono citati i file o le azioni, cioè i TARGET, da cui dipende il completamento dell'azione TARGET.

Un COMMAND è invece il comando da eseguire; può essere più di uno, e di solito si applica sulle DEPENDENCIES

```
# Commento: variabile CC è il comando che invoca il compilatore
CC=g++
# CCFLAGS: flags per il compilatore
CCFLAGS=-Wall -march=x86-64

my_prog : main.o kbd.o video.o help.o print.o
$(CC) $(CCFLAGS) -o my_prog main.o kbd.o video.o \\\
help.o print.o

main.o : main.cpp config.h
$(CC) $(CCFLAGS) -c main.cpp -o main.o

kbd.o : kbd.cpp config.h
$(CC) $(CCFLAGS) -c kbd.cpp -o kbd.o

video.o : video.cpp config.h defs.h
$(CC) $(CCFLAGS) -c video.cpp -o video.o

help.o : help.cpp help.h
$(CC) $(CCFLAGS) -c help.cpp -o help.o

print.o : print.cpp
$(CC) $(CCFLAGS) -c print.cpp -o print.o

clean :
rm *.o
echo "pulizia completata"
```

Program Development Flow



Integrated development environment

Da Wikipedia, l'enciclopedia libera.

In **informatica** un **ambiente di sviluppo integrato** (in **lingua inglese** **integrated development environment** ovvero **IDE**, anche *integrated design environment* o *integrated debugging environment*, rispettivamente *ambiente integrato di progettazione* e *ambiente integrato di debugging*) è un **software** che, in fase di **programmazione**, aiuta i **programmatore** nello sviluppo del **codice sorgente** di un **programma**. Spesso l'IDE aiuta lo sviluppatore segnalando **errori di sintassi** del codice direttamente in fase di scrittura, oltre a tutta una serie di strumenti e funzionalità di supporto alla fase di sviluppo e **debugging**.



Caratteristiche [\[modifica \]](#) [modifica wikitesto](#)]

Normalmente è uno strumento software che consiste di più componenti, da cui appunto il nome *integrato*:





- un **editor** di codice sorgente;
- un **compilatore** e/o un **interprete**;
- un tool di **building automatico**;
- (solitamente) un **debugger**.

svn, git

A volte è integrato anche con un **sistema di controllo di versione** e uno o più tool per semplificare la costruzione di una **GUI**.

Alcuni IDE, rivolti allo sviluppo di software orientato agli **oggetti**, comprendono anche un navigatore di **classi**, un *analizzatore di oggetti* e un *diagramma della gerarchia delle classi*.

IDE più noti ed usati:

- Eclipse (open source)  eclipse
- NetBeans (open source)  NetBeans
- Visual Studio (Microsoft, free per studenti UniPD)  Visual Studio
- XCode (Apple) 

Top IDE index

The Top IDE Index is created by analyzing how often IDEs' download page are searched on Google

The more an IDE is searched, the more popular the IDE is assumed to be. The raw data comes from Google Trends.

If you believe in collective wisdom, the Top IDE index can help you decide which IDE to use for your software development project.



Worldwide, Oct 2020 compared to a year ago:

Rank	Change	IDE	Share	Trend
1	↑	Visual Studio	25.0 %	+3.4 %
2	↑	Eclipse	16.7 %	-1.0 %
3	↓↓	Android Studio	12.18 %	-11.6 %
4	↑	Visual Studio Code	8.49 %	+3.4 %
5	↑	pyCharm	7.58 %	+2.5 %
6	↑	IntelliJ	5.93 %	+1.2 %
7	↓↓↓	NetBeans	5.1 %	-0.3 %
8		Xcode	4.55 %	+0.6 %
9	↑	Atom	3.87 %	+0.8 %
10	↓	Sublime Text	3.75 %	+0.1 %
11		Code::Blocks	1.84 %	+0.4 %
12		Vim	0.96 %	+0.1 %
13	↑	PhpStorm	0.65 %	+0.0 %
14	↓	Xamarin	0.63 %	-0.1 %
15		Komodo	0.49 %	+0.1 %
16		Qt Creator	0.39 %	+0.1 %
17		Emacs	0.3 %	+0.1 %

- Preprocessore
- Direttive al preprocessore
- **#include**: direttiva di inclusione
- **#define**: definizione di una macro

C preprocessor

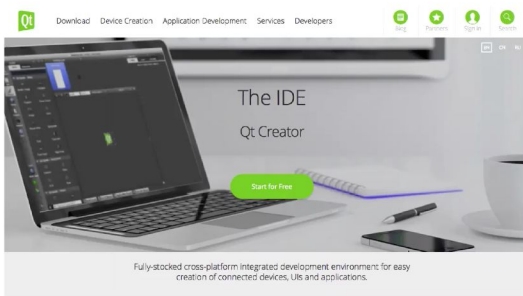
From Wikipedia, the free encyclopedia

The **C preprocessor** or **cpp** is the [macro preprocessor](#) for the C and C++ computer [programming languages](#). The preprocessor provides the ability for the inclusion of [header files](#), [macro expansions](#), [conditional compilation](#), and [line control](#).

In many C implementations, it is a separate [program](#) invoked by the [compiler](#) as the first part of [translation](#).

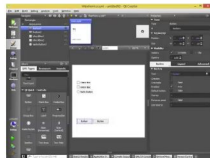
The language of preprocessor [directives](#) is only weakly related to the grammar of C, and so is sometimes used to process other kinds of [text files](#).





Beyond the Code Design and Create

We believe that delivering connected devices, UIs and applications that meet and exceed end user demands takes more than just clean code. You can't live on intuitive and comprehensive APIs alone. We want you to be able to not only code, but to also design and create. You've surely heard us say "code less, create more"? Well, this is where "create" comes into play.



Valgrind

Da Wikipedia, l'enciclopedia libera.

Valgrind (pronuncia: /ˈvælɡrɪnd/) è uno strumento [open source](#) per il [debug](#) di problemi di memoria, la ricerca dei [memory leak](#) ed il [profiling](#) del software.

È stato sviluppato come una versione liberamente distribuibile e modificabile di [Purify](#) per [Linux](#) su [x86](#), ma è divenuto un'infrastruttura per la creazione di strumenti di analisi dinamica, come la profilazione ed il controllo dei programmi.

Ha un'eccellente reputazione fra i programmatori Linux.



[<< Home Page](#)

[Information](#) [Source Code](#) [Documentation](#) [Contact](#) [How to Help](#) [Gallery](#)

| [Current Releases](#) | [Release Archive](#) | [Variants / Patches](#) | [Code Repository](#) | [Valkyrie / GUIs](#) |

Graphical User Interfaces

One of the most requested features for Valgrind is a graphical user interface to help with use and configuration.

Valkyrie is a Qt4-based GUI for the Memcheck and Helgrind tools in the Valgrind 3.6.X line, developed and maintained by the Valgrind Developers.

- Modularizzazione delle classi



- Modularizzazione delle classi

- Relazione has-a

Has-a

From Wikipedia, the free encyclopedia

In [database design](#), [object-oriented programming](#) and [design](#) (see [object oriented program architecture](#)), **has-a** (**has_a** or **has a**) is a [composition](#) relationship where one object (often called the constituted object, or part/constituent/member object) "belongs to" (is [part or member of](#)) another object (called the composite type), and behaves according to the rules of ownership. In simple words, **has-a** relationship in an object is called a member field of an object. Multiple **has-a** relationships will combine to form a possessive hierarchy.


This is to be contrasted with an [is-a](#) (*is_a* or *is a*) relationship which constitutes a taxonomic hierarchy ([subtyping](#)).

Esercizio (P1): quale è l'invariante?






```
// file telefonata.h
#ifndef TELEFONATA_H
#define TELEFONATA_H
#include <iostream>
#include "orario.h"

class telefonata {
private:
    orario inizio, fine; // relazione has-a
    int numero; 
```

```
// file telefonata.h
#ifndef TELEFONATA_H
#define TELEFONATA_H
#include <iostream>
#include "orario.h"

class telefonata {
private:
    orario inizio, fine; // relazione has-a
    int numero; 
public:
    telefonata(orario, orario, int); //p.valore (per ora)
    telefonata();
    orario Inizio() const;
    orario Fine() const;
    int Numero() const;
    bool operator==(const telefonata&) const;
};

ostream& operator<<(ostream&, const telefonata&);
#endif
```

In memoria

orario inizio	orario fine	int numero
--------------------------	------------------------	-----------------------

telefonata

Esempio:

```
#define COST 123
#define max(A,B) ((A) > (B) ? (A) : (B))

int main() {
    x = COST;
    y = max(4, z+2);
}
```

viene espanso dal preprocessore come:

```
int main() {
    x = 123;
    y = ((4) > (z+2) ? (4) : (z+2));
}
```

Come si comporta il costruttore di telefonata?

<code>orario inizio</code>	<code>orario fine</code>	<code>int numero</code>	<code>telefonata</code>
--------------------------------	------------------------------	-----------------------------	-------------------------

Comportamento del costruttore

Consideriamo una classe **C** con campi dati $\mathbf{x_1}, \dots, \mathbf{x_k}$ di qualsiasi tipo, possibilmente qualche altra classe. L'ordine dei campi dati è determinato dall'ordine in cui essi appaiono nella definizione della classe **C**.

Supponiamo di definire nella classe **C** un qualsiasi costruttore come segue:

```
C(Tipo_1, ..., Tipo_n) { //codice }
```

Il comportamento di tale costruttore è il seguente:

- (1) Per ogni campo dati $\mathbf{x_j}$ di tipo non classe **T_j** (ovvero di tipo primitivo o derivato), viene allocato un corrispondente spazio in memoria per contenere un valore di tipo **T_j** ma il valore viene lasciato indefinito;
- (2) Per ogni campo dati $\mathbf{x_i}$ di tipo classe **T_i** viene invocato il costruttore di default **T_i ()** ;
- (3) Alla fine, viene eseguito il codice del corpo del costruttore.

Nota: i punti (1) e (2) vengono eseguiti per tutti i campi dati $\mathbf{x_1}, \dots, \mathbf{x_k}$ seguendo il loro ordine di dichiarazione.



```
class C {
private:
    int x;
public:
    C() {cout << "C0 "; x=0;}
    C(int k) {cout << "C1 "; x=k;}
};
class D {
private:
    C c;
public:
    D() {cout << "D0 "; c = C(3);}
};
class E {
private:
    char c;
    C c1;
public:
    D d;
    C c2;
};
int main() {
    E x; cout << "UNO\n";
    D y; cout << "DUE\n";
    E* p = &x; cout << "TRE\n";
    D& a = y; cout << "QUATTRO";
}
```

Cosa stampa?





```
class B {
private:
    int x;
public:
    B(int a) {x=a;}
};

int main(){
    C z;
}

// ERRORE in compilazione:
// In method C::C() no matching function for call
// to B::B(). Candidates are: B::B(int)  B::B(const B&)
```

Definizione dei metodi di **telefonata**

```
// file telefonata.cpp
#include "telefonata.h"

telefonata::telefonata(orario i, orario f, int n) {
    inizio = i;
    fine = f;
    numero = n;
}

telefonata::telefonata() {numero = 0;}
// Attenzione: gli altri campi dati di tipo orario vengono
// inizializzati tramite il costruttore di default

orario telefonata::Inizio() const {return inizio;}

orario telefonata::Fine() const {return fine;}

int telefonata::Numero() const {return numero;}
```

```
bool telefonata::operator==(const telefonata& t) const {  
    return inizio == t.inizio &&  
        fine == t.fine &&  
        numero == t.numero;  
}
```

```
bool telefonata::operator==(const telefonata& t) const {  
    return inizio == t.inizio &&  
        fine == t.fine &&  
        numero == t.numero;  
}  
  
ostream& operator<<(ostream& s, const telefonata& t) {  
    return s << "INIZIO " << t.Inizio()  
        << " FINE " << t.Fine()  
        << " NUMERO CHIAMATO " << t.Numero();  
}
```

Esercizio

La classe `telefonata` presenta l'inconveniente di non poter rappresentare numeri telefonici che iniziano con lo 0.




Scegliere una **rappresentazione alternativa** per il numero telefonico che risolva l'inconveniente e riscrivere le definizioni dei metodi per tale rappresentazione.

Problema dell'inclusione multipla di file header




Esempio

```
// file "C.h"
class C {
public:
    int x;
    C(int k=4) {x=k;}
};
```

```
// file "D.h"
#include<iostream>
#include "C.h" 

class D {
public:
    int x;
    D(int k=6) {x=k;}
    void print(const C& c) const {std::cout << x + c.x;}
};
```

```
// file "main.cpp"
#include "C.h" 
#include "D.h"

int main() {
    C c(3); D d(4);
    d.print(c);
}
// main.cpp non compila!
// g++: "redefinition of class C"
```

Direttive di compilazione condizionale

```
#ifdef identificatore
    text
#endif

#ifndef identificatore
    text
#endif

#ifdef identificatore1
    text1
#elif defined identificatore2
    text2
#endif
```

Operatore **defined** : verifica se un simbolo è stato definito o meno.

```
#ifdef identificatore  
#ifndef identificatore  
// sono equivalenti a  
#if defined identificatore  
#if !defined identificatore
```

Esempio standard:

```
// file orario.h
#ifndef ORARIO_H
#define ORARIO_H
class orario {
    ...
    ...
};
#endif
```

Altro esempio

```
#define LINUX
#ifdef LINUX
// istruzioni per versione Linux
#elif defined MacOS
// istruzioni per versione MacOS
#endif
```