

Risposte alle verifiche

Modulo 1: Elementi fondamentali del C++

1. Il C++ è al centro della programmazione moderna in quanto deriva dal C ed è il progenitore di Java e C#. Questi quattro sono i linguaggi di programmazione più importanti.
2. Vero, un compilatore C++ produce codice che può essere eseguito direttamente dal computer.
3. I tre principi guida della programmazione a oggetti sono incapsulamento, polimorfismo ed ereditarietà.
4. I programmi C++ iniziano l'esecuzione alla funzione **main()**.
5. Un'intestazione contiene informazioni utilizzate dal programma.
6. **<iostream>** è l'intestazione che supporta l'I/O. L'istruzione include l'intestazione **<iostream>** in un programma.
7. Un namespace è una regione di dichiarazione nella quale è possibile collocare vari elementi di programma. Gli elementi dichiarati in un namespace sono separati da quelli dichiarati in un altro.
8. Una variabile è una locazione di memoria dotata di nome. Il contenuto di una variabile può essere cambiato durante l'esecuzione di un programma.
9. Le variabili non valide sono **d** ed **e**. I nomi delle variabili non possono iniziare con una cifra e non possono essere uguali a una parola chiave del C++.
10. Un commento a linea singola inizia con **//** e termina alla fine della linea. Un commento multilinea inizia con **/*** e termina con ***/**.
11. La forma generale dell'istruzione **if** è:
if(condizione) istruzione;
La forma generale del ciclo **for** è:
for(inizializzazione; condizione; incremento) istruzione;
12. Un blocco di codice inizia con **{** e termina con **}**.
13. **//** Mostra una tabella di pesi sulla terra e sulla luna.

```
#include <iostream>
using namespace std;

int main() {
    double earthweight; // peso sulla terra
    double moonweight; // peso sulla luna
    int counter;

    counter = 0;
```

```

for(earthweight = 1.0; earthweight <= 100.0; earthweight++) {
    moonweight = earthweight * 0.17;
    cout << earthweight << " chilogrammi terrestri equivalgono a " <<
        moonweight << " chilogrammi lunari.\n";
    counter++;
    if(counter == 25) {
        cout << "\n";
        counter = 0;
    }
}

return 0;
}

```

- 14.** // Conversione di anni di Giove in anni terrestri.

```

#include <iostream>
using namespace std;

int main() {
    double e_years; // anni terrestri
    double j_years; // anni di Giove

    cout << "Inserire il numero di anni di Giove: ";
    cin >> j_years;

    e_years = j_years * 12.0;

    cout << "Anni terrestri equivalenti: " << e_years;

    return 0;
}

```

- 15.** Quando una funzione viene chiamata, il controllo del programma viene trasferito alla funzione.

- 16.** // Calcola la media dei valori assoluti di 5 numeri.

```

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int i;
    double avg, val;

    avg = 0.0;

    for(i=0; i<5; i++) {
        cout << "Inserire un valore: ";

```

```

    cin >> val;

    avg = avg + abs(val);
}
avg = avg / 5;

cout << "Media dei valori assoluti: " << avg;

return 0;
}

```

Modulo 2: Introduzione a tipi di dati e operatori

1. I tipi interi del C++ sono:

int	short int	long int
unsigned int	unsigned short int	unsigned long int
signed int	signed short int	signed long int

Anche il tipo **char** può essere usato come tipo intero.

2. 12.2 è di tipo **double**.
3. Una variabile di tipo **bool** può essere **true** o **false**.
4. Il tipo intero lungo è **long int**, o semplicemente **long**.
5. La sequenza **\t** rappresenta una tabulazione. **\b** produce un segnale acustico.
6. Vero, una stringa è racchiusa tra apici doppi.
7. Le cifre esadecimali sono 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
8. Per inizializzare una variabile si usa questa forma generale:

tipo variabile = valore;
9. Il simbolo **%** rappresenta l'operatore di modulo, che restituisce il resto di una divisione intera. Non può essere utilizzato con i valori in virgola mobile.
10. Quando l'operatore di incremento o decremento precede il suo operando, il C++ eseguirà l'operazione corrispondente prima di ottenere il valore dell'operando in modo che sia utilizzato da parte del resto dell'espressione. Se l'operatore segue l'operando, il C++ otterrà invece il valore dell'operando prima di incrementarlo.
11. A, C ed E.
12. `x += 12;`
13. Un cast è una conversione di tipo esplicita.
14. Ecco un modo per trovare i numeri primi tra 1 e 100. Esistono ovviamente altre soluzioni.

```

// Trova i numeri primi tra 1 e 100.

#include <iostream>
using namespace std;

int main() {
    int i, j;
    bool isprime;

    for(i=1; i < 100; i++) {
        isprime = true;

        // verifica se il numero è divisibile esattamente
        for(j=2; j <= i/2; j++)
            // se lo è, non è primo
            if((i%j) == 0) isprime = false;

        if(isprime)
            cout << i << " è primo.\n";
    }

    return 0;
}

```

Modulo 3: Istruzioni di controllo

1. // Conteggio dei punti.

```

#include <iostream>
using namespace std;

int main() {
    char ch;
    int periods = 0;

    cout << "Digitare $ per interrompere.\n";

    do {
        cin >> ch;
        if(ch == '.') periods++;
    } while(ch != '$');

    cout << "Punti: " << periods << "\n";

    return 0;
}

```

2. Sì. Se manca un'istruzione **break** a conclusione di una sequenza **case**, l'esecuzione continuerà nel **case** successivo. Un'istruzione **break** evita che ciò accada.

3. `if(condizione)`

`istruzione;`

`else if(condizione)`

`istruzione;`

`else if(condizione)`

`istruzione;`

·

·

·

`else`

`istruzione;`

4. L'ultimo **else** è associato all'**if** esterno, che è l'**if** più vicino al medesimo livello dell'**else**.

5. `for(int i = 1000; i >= 0; i -= 2) // ...`

6. No. Secondo lo standard ANSI/ISO del C++ **i** non è nota solo al di fuori dell'istruzione in cui viene dichiarata (alcuni compilatori però potrebbero gestire la cosa in modo diverso).

7. Un **break** provoca la terminazione del ciclo o dell'istruzione **switch** che lo racchiude direttamente.

8. Dopo l'esecuzione di **break** viene visualizzato "dopo while".

9. 0 1

2 3

4 5

6 7

8 9

10. `/*`

 Uso di un ciclo for per generare la progressione

 1 2 4 8 16, ...

`*/`

`#include <iostream>`

`using namespace std;`

`int main() {`

`for(int i = 1; i < 100; i += i)`

`cout << i << " ";`

`cout << "\n";`

`return 0;`

`}`

11. // Conversione da maiuscolo a minuscolo e viceversa.

```
#include <iostream>
using namespace std;

int main() {
    char ch;
    int changes = 0;

    cout << "Digitare un punto per interrompere.\n";

    do {
        cin >> ch;
        if(ch >= 'a' && ch <= 'z') {
            ch -= (char) 32;
            changes++;
            cout << ch;
        }
        else if(ch >= 'A' && ch <= 'Z') {
            ch += (char) 32;
            changes++;
            cout << ch;
        }
    } while(ch != '.');

    cout << "\nConversioni eseguite: " << changes << "\n";

    return 0;
}
```

12. **goto** è l'istruzione di salto non condizionale del C++.

Modulo 4: Array, stringhe e puntatori

1. short int hightemps[31];

2. zero

3. // Trova i duplicati

```
#include <iostream>
using namespace std;

int main()
{
    int nums[] = {1, 1, 2, 3, 4, 2, 5, 4, 7, 7};

    for(int i=0; i < 10; i++)
        for(int j=i+1; j < 10; j++)
            if(nums[i] == nums[j])
                cout << "Duplicato: " << nums[i] << "\n";
}
```

```

    return 0;
}

```

4. Una stringa con terminatore NULL è un array di caratteri che termina con un NULL.

5. /* Ignora le differenze di maiuscolo e minuscolo
nel confronto di stringhe. */

```

#include <iostream>
#include <cctype>
using namespace std;

int main()
{
    char str1[80];
    char str2[80];
    char *p1, *p2;

    cout << "Immettere la prima stringa: ";
    cin >> str1;
    cout << "Immettere la seconda stringa: ";
    cin >> str2;

    p1 = str1;
    p2 = str2;

    // viene iterato fino a quando p1 e p2 puntano a caratteri non
    null
    while(*p1 && *p2) {
        if(tolower(*p1) != tolower(*p2)) break;
        else {
            p1++;
            p2++;
        }
    }

    /* le stringhe sono uguali se p1 e p2 puntano
    entrambe al terminatore NULL.
    */
    if(!*p1 && !*p2)
        cout << "Le stringhe sono uguali tranne che per " <<
            "le eventuali differenze di maiuscolo e minuscolo.\n";
    else
        cout << "Le stringhe sono diverse.\n";

    return 0;
}

```

6. Quando si usa **strcat()**, l'array di destinazione dev'essere grande abbastanza da accogliere il contenuto di entrambe le stringhe.

7. In un array multidimensionale ciascun indice viene specificato nel proprio insieme di parentesi quadre.
8. `int nums[] = {5, 66, 88};`
9. La dichiarazione di un array privo di limiti garantisce che un array inizializzato sarà sempre abbastanza grande da contenere gli inizializzatori specificati.
10. Un puntatore è un oggetto che contiene un indirizzo di memoria; gli operatori sui puntatori sono `&` e `*`.
11. Sì, è possibile indicizzare un puntatore come se fosse un array. Sì, è possibile accedere a un array attraverso un puntatore.
12. `// Conta le lettere maiuscole.`
`#include <iostream>`
`#include <cstring>`
`#include <cctype>`
`using namespace std;`

`int main()`
`{`
 `char str[80];`
 `int i;`
 `int count;`

 `strcpy(str, "Programma Di Prova");`

 `count = 0;`
 `for(i=0; str[i]; i++)`
 `if(isupper(str[i])) count++;`

 `cout << str << " contiene " << count << " lettere maiuscole.";`

 `return 0;`
`}`
13. *Indirizione multipla* è il termine utilizzato per la situazione in cui un puntatore punta a un altro.
14. Per convenzione si suppone che un puntatore NULL è inutilizzato.

Modulo 5: Introduzione alle funzioni

1. La forma generale di una funzione è:

```
tipo-restituzione nome(lista parametri)
{
    // corpo della funzione
}
```

2.

```
#include <iostream>
#include <cmath>
using namespace std;

double hypot(double a, double b);

int main() {

    cout << "Ipotenusa di un triangolo rettangolo 3 per 4: ";
    cout << hypot(3.0, 4.0) << "\n";

    return 0;
}

double hypot(double a, double b)
{
    return sqrt((a*a) + (b*b));
}
```
3. Sì, una funzione può restituire un puntatore. No, una funzione non può restituire un array.
4. // Versione personalizzata di strlen().

```
#include <iostream>
using namespace std;

int mystrlen(char *str);

int main()
{
    cout << "La lunghezza di Salve gente è: ";
    cout << mystrlen("Salve gente");

    return 0;
}

// Versione personalizzata di strlen().
int mystrlen(char, *str);
{
    int i;

    for(i=0; str[i]; i++) ; // trova la fine della stringa

    return i;
}
```
5. No, il valore di una variabile locale viene perso quando la funzione ritorna (o più in generale, il suo valore va perso all'uscita dal blocco).
6. I vantaggi principali delle variabili globali sono la loro disponibilità per tutte le altre funzioni del programma e la loro esistenza per tutta la durata del programma. Gli

svantaggi principali sono che occupano la memoria per tutta l'esecuzione del programma, che l'uso di una variabile globale laddove potrebbe andar bene una variabile locale rende meno generale la funzione, e che l'uso di un grande numero di variabili globali può portare a effetti collaterali imprevisi.

```
7. #include <iostream>
using namespace std;

int seriesnum = 0;

int byThrees();
void reset();

int main() {

    for(int i=0; i < 10; i++)
        cout << byThrees() << " ";

    cout << "\n";

    reset();

    for(int i=0; i < 10; i++)
        cout << byThrees() << " ";

    cout << "\n";

    return 0;
}
```

```
8. int byThrees()
{
    int t;

    t = seriesnum;
    seriesnum += 3;

    return t;
}

void reset()
{
    seriesnum = 0;
}

#include <iostream>
#include <cstring>
using namespace std;

int main(int argc, char *argv[])
```

```

{
  if(argc != 2) {
    cout << "Specificare la password!\n";
    return 0;
  }

  if(!strcmp("mypassword", argv[1]))
    cout << "Accesso consentito.\n";
  else
    cout << "Accesso negato.\n";

  return 0;
}

```

9. Vero. Un prototipo impedisce che una funzione venga chiamata con il numero non corretto di argomenti.
10. #include <iostream>
using namespace std;

```

void printnum(int n);

int main()
{
  printnum(10);

  return 0;
}

void printnum(int n)
{
  if(n > 1) printnum(n-1);
  cout << n << " ";
}

```

Modulo 6: Un'analisi più dettagliata delle funzioni

1. Un argomento può essere passato a una subroutine con una chiamata per valore o una chiamata per reference.
2. Un reference è un puntatore implicito, Un parametro reference viene creato anteponendo un simbolo **&** al nome del parametro.
3. f(ch, &i);
4. #include <iostream>
#include <cmath>
using namespace std;

```

void round(double &num);

```

```

int main()
{
    double i = 100.4;

    cout << i << " arrotondato è ";
    round(i);
    cout << i << "\n";

    i = -10.9;
    cout << i << " arrotondato è ";
    round(i);
    cout << i << "\n";

    return 0;
}

void round(double &num)
{
    double frac;
    double val;

    // scompono num nelle parti intera e frazionaria
    frac = modf(num, &val);

    if(frac < 0.5) num = val;
    else num = val+1.0;
}

```

5. #include <iostream>
using namespace std;

```

// Scambia gli argomenti e restituisce il minimo.
int & min_swap(int &x, int &y);

int main()
{
    int i, j, min;

    i = 10;
    j = 20;

    cout << "Valori iniziali di i e j: ";
    cout << i << ' ' << j << '\n';

    min = min_swap(j, i);

    cout << "Valori scambiati di i e j: ";
    cout << i << ' ' << j << '\n';

    cout << "Il valore minimo è " << min << "\n";
}

```

```

    return 0;
}

// Scambia gli argomenti e restituisce il minimo.
int &min_swap(int &x, int &y)
{
    int temp;

    // usa dei reference per scambiare i valori degli argomenti
    temp = x;
    x = y;
    y = temp;

    // restituisce un reference all'argomento minimo
    if(x < y) return x;
    else return y;
}

```

6. Una funzione non deve restituire un reference a una variabile locale, perché questa uscirà di ambito (ossia cesserà di esistere) quando la funzione ritorna.

7. Le funzioni soggette a overload devono differire nel tipo e/o nel numero dei parametri.

8. /*

Progetto 6-1 - aggiornato per la Verifica modulo

Creazione di funzioni println() soggette a overload
che visualizzano vari tipi di dati.

Questa versione include un parametro di rientro.

*/

```

#include <iostream>
using namespace std;

```

// Queste funzioni producono un carattere newline.

```

void println(bool b, int ident=0);
void println(int i, int ident=0);
void println(long i, int ident=0);
void println(char ch, int ident=0);
void println(char *str, int ident=0);
void println(double d, int ident=0);

```

// Queste funzioni non producono un carattere newline.

```

void print(bool b, int ident=0);
void print(int i, int ident=0);
void print(long i, int ident=0);
void print(char ch, int ident=0);
void print(char *str, int ident=0);
void print(double d, int ident=0);

```

```

int main()
{
    println(true, 10);
    println(10, 5);
    println("Questa è una prova");
    println('x');
    println(99L, 10);
    println(123.23, 10);

    print("Ecco alcuni valori: ");
    print(false);
    print(88, 3);
    print(100000L, 3);
    print(100.01);

    println(" Fatto!");

    return 0;
}

// Ecco le funzioni println().
void println(bool b, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    if(b) cout << "vero\n";
    else cout << "falso\n";
}

void println(int i, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << i << "\n";
}

void println(long i, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << i << "\n";
}

void println(char ch, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';
}

```

```
    cout << ch << "\n";
}

void println(char *str, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << str << "\n";
}

void println(double d, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << d << "\n";
}

// Ecco le funzioni print().
void print(bool b, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    if(b) cout << "vero";
    else cout << "falso";
}

void print(int i, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << i;
}

void print(long i, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << i;
}

void print(char ch, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';
```

```

    cout << ch;
}

void print(char *str, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << str;
}

void print(double d, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << d;
}

```

9. `myfunc('x');`
`myfunc('x', 19);`
`myfunc('x', 19, 35);`
10. L'overload delle funzioni può introdurre ambiguità quando il compilatore non riesce a decidere quale versione della funzione chiamare. Questo può avvenire quando si verificano delle conversioni di tipo automatiche e quando si usano gli argomenti di default.

Modulo 7: Altri tipi di dati e operatori

1. `static int test = 100;`
2. Vero. Lo specificatore **volatile** indica al compilatore che una variabile può essere modificata da fattori esterni al programma.
3. In un progetto a più file, affinché un file conosca una variabile globale dichiarata in un altro file si utilizza **extern**.
4. L'attributo più importante di una variabile locale **static** è che mantiene il proprio valore tra una chiamata alla funzione e l'altra.
5. `// Uso di static per contare le chiamate a funzione.`

```

#include <iostream>
using namespace std;

int counter();

int main()
{

```

```

int result;

for(int i=0; i<10; i++)
    result = counter();

cout << "Funzione chiamata " <<
     result << " volte." << "\n";

return 0;
}

int counter()
{
    static count = 0;

    count++;

    return count;
}

```

6. Il massimo effetto sulle prestazioni si avrà specificando **x** come **register**, seguita da **y** e poi da **z**. Il motivo è che l'accesso a **x** nel ciclo è il più frequente, il secondo per frequenza è **y**, mentre **z** viene usata solo quando il ciclo viene inizializzato.
7. **&** è un operatore bit-a-bit che agisce sui singoli bit di un valore. **&&** è un operatore logico che agisce su valori di tipo vero/falso.
8. L'istruzione moltiplica per 10 il valore corrente di **x** e assegna il risultato a **x**. Equivale a scrivere:

```
x = x * 10;
```

9. // Uso delle rotazioni per codificare un messaggio.

```

#include <iostream>
#include <cstring>
using namespace std;

unsigned char rrotate(unsigned char val, int n);
unsigned char lrotate(unsigned char val, int n);
void show_binary(unsigned int u);

int main()
{
    char msg[] = "Questa è una prova.";
    char *key = "xanadu";
    int klen = strlen(key);
    int rotnum;

    cout << "Messaggio originale: " << msg << "\n";
}

```

```

// Codifica il messaggio con una rotazione a sinistra.
for(int i = 0 ; i < strlen(msg); i++) {
    /* Rotazione a sinistra di ciascuna lettera di un valore
       derivato dalla stringa chiave. */
    rotnum = key[i%klen] % 8;
    msg[i] = lrotate(msg[i], rotnum);
}

cout << "Messaggio codificato: " << msg << "\n";

// Decodifica il messaggio con una rotazione a destra.
for(int i = 0 ; i < strlen(msg); i++) {
    /* Rotazione a destra di ciascuna lettera di un valore
       derivato dalla stringa chiave. */
    rotnum = key[i%klen] % 8;

    msg[i] = rrotate(msg[i], rotnum);
}

cout << "Messaggio decodificato: " << msg << "\n";

return 0;
}

// Rotazione a sinistra di un byte di n posizioni.
unsigned char lrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    for(int i=0; i < n; i++) {
        t = t << 1;

        /* Se un bit esce da un lato, si troverà nel bit 8
           dell'intero t. In tal caso,
           quel bit viene collocato sul lato destro. */
        if(t & 256)
            t = t | 1; // mette un 1 sul lato destro
    }

    return t; // restituisce gli 8 bit di ordine inferiore.
}

// Rotazione a destra di un byte di n posizioni.
unsigned char rrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

```

```

// Prima sposta il valore di 8 più in alto.
t = t << 8;

for(int i=0; i < n; i++) {
    t = t >> 1;

    /* Se un bit esce da un lato, si troverà nel bit 7
       dell'intero t. In tal caso,
       quel bit viene collocato sul lato sinistro. */
    if(t & 128)
        t = t | 32768; // mette un 1 all'estremità sinistra
}

/* Alla fine riporta il risultato agli
   8 bit di ordine inferiore di t. */
t = t >> 8;

return t;
}

```

10. // Visualizzazione dei bit all'interno di un byte.

```

void show_binary(unsigned int u)
{
    int t;

    for(t=128; t>0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}

```

Modulo 8: Classi e oggetti

1. Una classe è un costrutto logico che definisce la forma di un oggetto. Un oggetto è un'istanza di una classe, pertanto ha realtà fisica in memoria.
2. Per definire una classe si usa la parola chiave **class**.
3. Ciascun oggetto possiede la propria copia delle variabili membro di una classe.
4.

```

class Test {
    int count;
    int max;
    // ...
}

```
5. Un costruttore ha lo stesso nome della sua classe. Il distruttore ha lo stesso nome della sua classe preceduto da un simbolo ~.

6. Ecco tre modi per creare un oggetto che inizializza i a 10:

```
Sample ob(10);
Sample ob = 10;
Sample ob = Sample(10);
```

7. Quando una funzione membro viene dichiarata all'interno di una classe, viene automaticamente espansa in linea, se possibile.

8. // Crea una classe Triangle.

```
#include <iostream>
#include <cmath>
using namespace std;

class Triangle {
    double height;
    double base;
public:
    Triangle(double h, double b) {
        height = h;
        base = b;
    }

    double hypot() {
        return sqrt(height*height + base*base);
    }

    double area() {
        return base * height / 2.0;
    }
};

int main()
{
    Triangle t1(3.0, 4.0);
    Triangle t2(4.5, 6.75);

    cout << "Ipotenusa di t1: " <<
        t1.hypot() << "\n";
    cout << "Area di t1: " <<
        t1.area() << "\n";

    cout << "Ipotenusa di t2: " <<
        t2.hypot() << "\n";
    cout << "Area di t2: " <<
        t2.area() << "\n";

    return 0;
}
```

```

9. /*
    Progetto 8-1 potenziato

    Aggiunta di un ID utente alla classe Help.
*/

#include <iostream>
using namespace std;

// Una classe che incapsula una guida in linea.
class Help {
    int userID;
public:
    Help(int id) { userID = id; }

    ~Help() { cout << "Termine della guida per #" <<
        userID << ".\n"; }

    int getID() { return userID; }

    void helpon(char what);
    void showmenu();
    bool isvalid(char ch);
};

// Visualizzazione delle informazioni della guida.
void Help::helpon(char what) {
    switch(what) {
        case '1':
            cout << "Istruzione if:\n\n";
            cout << "if(condizione) istruzione;\n";
            cout << "else istruzione;\n";
            break;
        case '2':
            cout << "Istruzione switch:\n\n";
            cout << "switch(espressione) {\n";
            cout << "  case costante:\n";
            cout << "    sequenza di istruzioni\n";
            cout << "    break;\n";
            cout << "  // ... \n";
            cout << "}\n";
            break;
        case '3':
            cout << "Istruzione for:\n\n";
            cout << "for(inizializzazione; condizione; incremento)";
            cout << " istruzione;\n";
            break;
        case '4':
            cout << "Istruzione while:\n\n";
            cout << "while(condizione) istruzione;\n";
            break;
    }
}

```

```
case '5':
    cout << "Istruzione do-while:\n\n";
    cout << "do {\n";
    cout << "  istruzione;\n";
    cout << "} while (condizione);\n";
    break;
case '6':
    cout << "Istruzione break:\n\n";
    cout << "break;\n";
    break;
case '7':
    cout << "Istruzione continue:\n\n";
    cout << "continue;\n";
    break;
case '8':
    cout << "Istruzione goto:\n\n";
    cout << "goto etichetta;\n";
    break;
}
cout << "\n";
}

// Mostra il menu della guida.
void Help::showmenu() {
    cout << "Guida su:\n";
    cout << "  1. if\n";
    cout << "  2. switch\n";
    cout << "  3. for\n";
    cout << "  4. while\n";
    cout << "  5. do-while\n";
    cout << "  6. break\n";
    cout << "  7. continue\n";
    cout << "  8. goto\n";
    cout << "Scegliere un'istruzione (q per uscire): ";
}

// Restituisce true se una selezione è valida.
bool Help::isvalid(char ch) {
    if(ch < '1' || ch > '8' && ch != 'q')
        return false;
    else
        return true;
}

int main()
{
    char choice;
    Help hlpob(27); // crea un'istanza della classe Help.

    cout << "L'ID utente è " << hlpob.getID() <<
        ".\n";
}
```

```

// Usa l'oggetto Help per visualizzare le informazioni.
for(;;) {
    do {
        hlpob.showmenu();
        cin >> choice;
    } while(!hlpob.isvalid(choice));

    if(choice == 'q') break;
    cout << "\n";

    hlpob.helpon(choice);
}

return 0;
}

```

Modulo 9: Descrizione approfondita delle classi

1. Un costruttore di copia realizza una copia di un oggetto. Viene chiamato quando un oggetto ne inizializza un altro. Ecco la forma generale:

```

nomeclasse (const nomeclasse &oggetto) {
    // corpo del costruttore
}

```

2. Quando un oggetto è restituito da una funzione, viene creato un oggetto temporaneo come valore restituito. L'oggetto viene distrutto dal distruttore dell'oggetto una volta restituito il valore.
3.

```
int sum() {
    return this.i + this.j;
}
```
4. Una struttura è una classe i cui membri sono pubblici per default. Una union è una classe in cui tutti i membri dati condividono la stessa memoria. Anche i membri delle union sono pubblici per default.
5. ***this** si riferisce all'oggetto per il quale è stata chiamata la funzione.
6. Una funzione **friend** è una funzione non membro che può accedere ai membri privati della classe per la quale è friend.
7.

```
tipo nomeclasse::operator#(tipo op2)
{
    // operando di sinistra passato tramite "this"
}
```
8. Per consentire le operazioni tra un tipo classe e un tipo nativo dovete utilizzare due funzioni operatore **friend**, una con il tipo classe come primo parametro, l'altra con il tipo nativo come primo parametro.

9. No, l'overload di ? non è possibile. No, non è possibile modificare la precedenza di un operatore.
10. // Determina se un insieme è un sottoinsieme di un altro.

```
bool Set::operator <(Set ob2) {
    if(len > ob2.len) return false; // ob1 ha più elementi

    for(int i=0; i < len; i++)
        if(ob2.find(members[i]) == -1) return false;
    return true;
}

// Determina se un insieme è un sovrainsieme di un altro.
bool Set::operator >(Set ob2) {
    if(len < ob2.len) return false; // ob1 ha meno elementi

    for(int i=0; i < ob2.len; i++)
        if(find(ob2.members[i]) == -1) return false;
    return true;
}

```
11. // Imposta l'intersezione.

```
Set Set::operator &(Set ob2) {
    Set newset;

    // Aggiunge gli elementi comuni a entrambi gli insiemi.
    for(int i=0; i < len; i++)
        if(ob2.find(members[i]) != -1) // aggiunge l'elemento if
            // in entrambi gli insiemi
            newset = newset + members[i];

    return newset; // restituisce l'insieme
}

```

Modulo 10: Ereditarietà, funzioni virtuali e polimorfismo

1. Una classe che viene ereditata prende il nome di classe *base*. Quella che eredita viene chiamata classe *derivata*.
2. Una classe base *non* ha accesso ai membri delle classi derivate perché non ha alcuna conoscenza dell'esistenza di quest'ultima. Una classe derivata ha accesso ai membri non privati della sua classe base.
3. // Una classe Circle.

```
class Circle : public TwoDShape {
public:
    Circle(double r) : TwoDShape(r) { } // specifica il raggio
}

```

```
double area() {
    return getWidth() * getHeight() * 3.1416;
}
};
```

4. Per impedire che una classe derivata acceda a un membro di una classe base, occorre dichiarare **private** quel membro nella classe base.
5. Ecco la forma generale del costruttore di una classe derivata che chiama un costruttore della classe base:


```
classe-derivata() : classe-base() { //...
```
6. I costruttori vengono sempre chiamati in ordine di derivazione. Pertanto, quando viene creato un oggetto **Gamma**, i costruttori vengono chiamati in questo ordine: **Alpha, Beta, Gamma**.
7. A un membro **protected** in una classe base possono accedere la classe stessa e quelle da essa derivate. Per tutti gli altri elementi del programma è privato.
8. Quando la chiamata a una funzione virtuale avviene tramite un puntatore alla classe base, è il tipo dell'oggetto cui punta il puntatore che determina quale versione di quella funzione sarà chiamata.
9. Una funzione virtuale pura è una funzione che non ha corpo nella sua classe base e pertanto dev'essere sovrascritta dalle classi derivate. Una classe astratta è una classe che contiene almeno una funzione virtuale pura.
10. No, una classe astratta non può essere impiegata per creare un oggetto.
11. Una funzione virtuale pura rappresenta una descrizione generica alla quale tutte le implementazioni di tale funzione devono aderire. Pertanto nella frase "un'interfaccia, molti metodi" la funzione virtuale pura rappresenta l'*interfaccia*, mentre le singole implementazioni sono i *metodi*.

Modulo 11: Il sistema di I/O del C++

1. Gli stream predefiniti sono **cin**, **cout**, **cerr** e **clog**.
2. Sì, il C++ definisce stream sia a 8 bit sia stream di caratteri estesi.
3. La forma generale per l'overload di un inseritore è la seguente:

```
ostream &operator<<(ostream &stream, class_type obj)
{
    // scrivere qui il codice specifico della classe
    return stream; // restituisce lo stream
}
```

4. **ios::scientific** fa in modo che l'output numerico venga visualizzato in notazione scientifica.

5. La funzione **width()** imposta il campo **width**.
6. Vero, un manipolatore di I/O viene utilizzato in un'espressione di I/O.
7. Ecco un modo per aprire un file per l'input testuale:

```
ifstream in("test");
if(!in) {
    cout << "Impossibile aprire il file.\n";
    return 1;
}
```

8. Ecco un modo per aprire un file per l'output testuale:

```
ofstream out("test");
if(!out) {
    cout << "Impossibile aprire il file.\n";
    return 1;
}
```

9. **ios::binary** specifica che un file viene aperto per l'I/O binario invece che testuale.
10. Vero, alla fine del file la variabile stream verrà valutata falsa.
11. `while(strm.get(ch)) // ...`
12. Esistono molte soluzioni. Quella che segue è solo un esempio:

```
// Copia di un file.

#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=3) {
        cout << "Modalità d'uso: copy <source> <target>\n";
        return 1;
    }

    ifstream src(argv[1], ios::in | ios::binary);
    if(!src) {
        cout << "Impossibile aprire il file di origine.\n";
        return 1;
    }

    ofstream targ(argv[2], ios::out | ios::binary);
    if(!targ) {
        cout << "Impossibile aprire il file di destinazione.\n";
        return 1;
    }
}
```

```

do {
    src.get(ch);
    if(!src.eof()) targ.put(ch);
} while(!src.eof());

src.close();
targ.close();

return 0;
}

```

13. Esistono molte soluzioni. Quella che segue è un semplice esempio:

```

// Fusione di due file.

#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    char ch;

    if(argc != 4) {
        cout << "Modalità d'uso: merge <source1> <source2> <target>\n";
        return 1;
    }

    ifstream src1(argv[1], ios::in | ios::binary);
    if(!src1) {
        cout << "Impossibile aprire il primo file di origine.\n";
        return 1;
    }

    ifstream src2(argv[2], ios::in | ios::binary);
    if(!src2) {
        cout << "Impossibile aprire il secondo file di origine.\n";
        return 1;
    }

    ofstream targ(argv[3], ios::out | ios::binary);
    if(!targ) {
        cout << "Impossibile aprire il file di destinazione.\n";
        return 1;
    }

    // Copia il primo file di origine.
    do {
        src1.get(ch);

```

```

        if(!src1.eof()) targ.put(ch);
    } while(!src1.eof());

    // Copia il secondo file di origine.
    do {
        src2.get(ch);
        if(!src2.eof()) targ.put(ch);
    } while(!src2.eof());

    src1.close();
    src2.close();
    targ.close();

    return 0;
}

```

14. `MyStrm.seekg(300, ios::beg);`

Modulo 12: Eccezioni, template e altri elementi di programmazione avanzata

1. La gestione delle eccezioni del C++ si fonda su tre parole chiave: **try**, **catch** e **throw**. In termini molto generali si potrebbe dire che le istruzioni del programma che si desidera controllare per individuare le eccezioni sono contenute in un blocco **try**; se all'interno di questo blocco si verifica un'eccezione, ossia un errore, essa viene lanciata tramite **throw**; tale eccezione viene poi catturata con **catch** ed elaborata.
2. Quando si catturano le eccezioni della classe base e di quelle derivate, le classi derivate devono precedere la classe base in una lista **catch**.
3. `void func() throw(MyExcpt)`
4. Ecco un modo per aggiungere un'eccezione a **Queue**. È una delle molte soluzioni possibili.

```

/*
    Aggiunta di un'eccezione al Progetto 12-1

    Una classe template Queue.
*/
#include <iostream>
#include <cstring>
using namespace std;

// Questa è l'eccezione lanciata da Queue in caso di errore.
class QExcpt {
public:
    char msg[80];
};

```

```
const int maxQsize = 100;

// Crea una classe Queue generica.
template <class QType> class Queue {
    QType q[maxQsize]; // questo array contiene la coda
    int size; // il numero massimo di elementi che la coda può accogliere
    int putloc, getloc; // gli indici put e get
    QExcpt Qerr; // aggiunge un campo di eccezione
public:

    // Costruisca una coda di lunghezza specifica.
    Queue(int len) {
        // La coda dev'essere meno di max e positiva.
        if(len > maxQsize) len = maxQsize;
        else if(len <= 0) len = 1;

        size = len;
        putloc = getloc = 0;
    }

    // Inserisce dati nella coda.
    void put(QType data) {
        if(putloc == size) {
            strcpy(Qerr.msg, "La coda è piena.\n");
            throw Qerr;
        }

        putloc++;
        q[putloc] = data;
    }

    // Estrae dati dalla coda.
    QType get() {
        if(getloc == putloc) {
            strcpy(Qerr.msg, "La coda è vuota.\n");
            throw Qerr;
        }

        getloc++;
        return q[getloc];
    }
};

// Dimostra la classe Queue generica.
int main()
{
    // osservate che iQa è lunga solo 2 elementi
    Queue<int> iQa(2), iQb(10);
```

```

try {
    iQa.put(1);
    iQa.put(2);
    iQa.put(3); // overflow della coda!

    iQb.put(10);
    iQb.put(20);
    iQb.put(30);

    cout << "Contenuto della coda di interi iQa: ";
    for(int i=0; i < 3; i++) // underflow della coda!
        cout << iQa.get() << " ";
    cout << endl;

    cout << "Contenuto della coda di interi iQb: ";
    for(int i=0; i < 3; i++)
        cout << iQb.get() << " ";
    cout << endl;

    Queue<double> dQa(10), dQb(10); // crea due code di double

    dQa.put(1.01);
    dQa.put(2.02);
    dQa.put(3.03);

    dQb.put(10.01);
    dQb.put(20.02);
    dQb.put(30.03);

    cout << "Contenuto della coda di double dQa: ";
    for(int i=0; i < 3; i++)
        cout << iQa.get() << " ";
    cout << endl;

    cout << "Contenuto della coda di double dQb: ";
    for(int i=0; i < 3; i++)
        cout << dQb.get() << " ";
    cout << endl;

} catch(QExcp exc) {
    cout << exc.msg;
}

return 0;
}

```

5. Una funzione generica definisce la forma generale di una routine, ma non specifica il tipo preciso dei dati su cui opera. Viene creato con la parola chiave **template**.
6. Ecco un modo per trasformare **quicksort()** e **qs()** in funzioni generiche:

```
// Una classe Quicksort generica.

#include <iostream>
#include <cstring>

using namespace std;

// Imposta una chiamata alla funzione di ordinamento vera e propria.
template <class X> void quicksort(X *items, int len)
{
    qs(items, 0, len-1);
}

// Una versione generica di Quicksort.
template <class X> void qs(X *items, int left, int right)
{
    int i, j;
    X x, y;

    i = left; j = right;
    x = items[( left+right) / 2 ];

    do {
        while((items[i] < x) && (i < right)) i++;
        while((x < items[j]) && (j > left)) j--;

        if(i <= j) {
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    } while(i <= j);

    if(left < j) qs(items, left, j);
    if(i < right) qs(items, i, right);
}

int main() {

    // ordina i caratteri.
    char str[] = "jfmckldoelazlkper";

    cout << "Ordine originale: " << str << "\n";

    quicksort(str, strlen(str));

    cout << "Ordine dopo l'esecuzione: " << str << "\n";
```

```

// ordina interi
int nums[] = { 4, 3, 7, 5, 9, 8, 1, 3, 5, 4 };

cout << "Ordine originale: ";
for(int i=0; i < 10; i++)
    cout << nums[i] << " ";
cout << endl;

quicksort(nums, 10);

cout << "Ordine dopo l'esecuzione: ";
for(int i=0; i < 10; i++)
    cout << nums[i] << " ";
cout << endl;

return 0;
}

```

7. Ecco un modo per memorizzare oggetti **Sample** in una **Queue**:

```

/*
    Uso del Progetto 12-1 per memorizzare oggetti Sample.

    Una classe Queue template.
*/
#include <iostream>
using namespace std;

class Sample {
    int id;
public:
    Sample() { id = 0; }
    Sample(int x) { id = x; }
    void show() { cout << id << endl; }
};

const int maxQsize = 100;

// Crea una classe Queue generica.
template <class QType> class Queue {
    QType q[maxQsize]; // questo array contiene la coda
    int size; // il numero massimo di elementi che la coda può accogliere
    int putloc, getloc; // gli indici put e get
public:

    // Costruisca una coda di lunghezza specifica.
    Queue(int len) {
        // La coda dev'essere meno di max e positiva.
        if(len > maxQsize) len = maxQsize;
    }
};

```

```
        else if(len <= 0) len = 1;

        size = len;
        putloc = getloc = 0;
    }

    // Inserisce dati nella coda.
    void put(QType data) {
        if(putloc == size) {
            cout << " -- La coda è piena.\n";
            return;
        }

        putloc++;
        q[putloc] = data;
    }

    // Estrae dati dalla coda.
    QType get() {
        if(getloc == putloc) {
            cout << " -- La coda è vuota.\n";
            return 0;
        }

        getloc++;
        return q[getloc];
    }
};

// Dimostrazione della classe Queue generica.
int main()
{
    Queue<Sample> sampQ(3);

    Sample o1(1), o2(2), o3(3);

    sampQ.put(o1);
    sampQ.put(o2);
    sampQ.put(o3);

    cout << "Contenuto di sampQ:\n";
    for(int i=0; i < 3; i++)
        sampQ.get().show();
    cout << endl;

    return 0;
}
```

8. Qui gli oggetti **Sample** vengono allocati:

```
/*
    Uso del Progetto 12-1 per memorizzare oggetti Sample.

    Allocazione dinamica di oggetti Sample.

    Una classe Queue template.
*/
#include <iostream>
using namespace std;

class Sample {
    int id;
public:
    Sample() { id = 0; }
    Sample(int x) { id = x; }
    void show() { cout << id << endl; }
};

const int maxQsize = 100;

// Crea una classe Queue generica.
template <class QType> class Queue {
    QType q[maxQsize]; // questo array contiene la coda
    int size; // il numero massimo di elementi
                // che la coda può accogliere
    int putloc, getloc; // gli indici put e get
public:

    // Costruisca una coda di lunghezza specifica.
    Queue(int len) {
        // La coda dev'essere meno di max e positiva.
        if(len > maxQsize) len = maxQsize;
        else if(len <= 0) len = 1;

        size = len;
        putloc = getloc = 0;
    }

    // Inserisce dati nella coda.
    void put(QType data) {
        if(putloc == size) {
            cout << " -- La coda è piena.\n";
            return;
        }

        putloc++;
        q[putloc] = data;
    }
};
```

```

// Estrae dati dalla coda.
QType get() {
    if(getloc == putloc) {
        cout << " -- La coda è vuota.\n";
        return 0;
    }

    getloc++;
    return q[getloc];
}
};

// Dimostrazione della classe Queue generica.
int main()
{
    Queue<Sample> sampQ(3);

    Sample *p1, *p2, *p3;

    p1 = new Sample(1);
    p2 = new Sample(2);
    p3 = new Sample(3);

    sampQ.put(*p1);
    sampQ.put(*p2);
    sampQ.put(*p3);

    cout << "Contenuto di sampQ:\n";
    for(int i=0; i < 3; i++)
        sampQ.get().show();
    cout << endl;

    delete(p1);
    delete(p2);
    delete(p3);

    return 0;
}

```

9. Per dichiarare un namespace di nome **RobotMotion** usate questa istruzione:

```

namespace RobotMotion {
    // ...
}

```

10. La libreria standard del C++ è contenuta nel namespace **std**

11. No, una funzione membro **static** non può accedere ai dati non **static** di una classe.

12. L'operatore **typeid** ottiene il tipo di un oggetto in fase di esecuzione.
13. Per determinare la validità di un cast polimorfo in fase di esecuzione si usa **dynamic_cast**.
14. **const_cast** sovrascrive **const** o **volatile** in un cast.