Appunti di Laboratorio di Programmazione

Giacomo Simonetto

Primo semestre 2023-24

Sommario

Appunti del corso di Laboratorio di Programmazione della facoltà di Ingegneria Informatica dell'Università di Padova.

"Il codice è professionale se altri lo possono riutilizzare"

Indice

1 Variabili

Definizione

- Un oggetto è un contenitore di dati, situato in memoria in grado di memorizzare valori che possono essere letti e modificati durante l'esecuzione di un programma.
- Una variabile è un oggetto identificato con un nome (*named object*).

In C++ le variabili sono tipizzate, ovvero sono caratterizzate da un tipo che rimane costante per tutta l'esistenza della variabile.

1.1 Tipo di una variabile

Introduzione

Il tipo di una variabile definisce la natura di tale variabile, ovvero un range di valori che può assumere e un insieme di operazioni che possono essere eseguite. Specifica anche la dimensione in byte dell'area riservata in memoria. Il tipo può essere:

- **built-in**: tipi primitivi del linguaggio e inclusi nello standard
- **user-defined-type o UDT**: tipi definiti dall'utente in base alle esigenze del programma

Esistono delle funzioni di conversione tra tipi (casting) approfondite nel paragrafo 2.3.

Tipi built-in in C++

keyword	size	min	max
bool	1 bit	0	1
char	8 bit	-128	127
unisgned char	8 bit	0	255
short	16 bit	-32768	32767
unsigned short	16 bit	0	65535
int	32 bit	-2147483648	2147483647
unsigned int	32 bit	0	4294967295
long int	64 bit	-9223372036854775808	9223372036854775807
unsigned long int	64 bit	0	18446744073709551615
long long int	64 bit	-9223372036854775808	9223372036854775807
unsigned long long int	64 bit	0	18446744073709551615

keyword	size	min	nearest to zero	max	epsilon
float	32 bit	-3.40282e+38	1.17549e-38	3.40282e+38	1.19209e-07
double	64 bit	-1.79769e+308	2.22507e-308	1.79769e+308	2.22045e-16
long double	64 bit	-1.79769e+308	2.22507e-308	1.79769e+308	2.22045e-16

1.2 Nome variabile

Il nome di una variabile permette di richiamarla durante il programma in maniera univoca. Non ci possono essere più variabili con lo stesso nome. Alcune buone pratiche per nominare una variabile:

- evitare nomi troppo corti corti (eccetto per situazioni particolari es. contatori cicli)
- evitare acronimi difficili da interpretare
- usare nomi brevi che esplicitano il ruolo della variabile nel codice

1.3 Costanti

Le costanti sono particolari variabili il cui valore non può essere modificato durante l'esecuzione di un programma. Sono molto utili per identificare con un nome i valori costanti usati nel programma ed evitare di usare "magic numbers" di cui si potrebbe non comprenderne il significato. In base a come viene calcolato il valore, possono essere di due tipi:

- `const` se il valore assegnato è noto solo in fase di esecuzione
- `constexpr` se il valore assegnato è noto a tempo di compilazione

Esiste un terzo "caso" che utilizza la direttiva al preprocessore `#define`, solo che non è una vera variabile. Prima della fase di compilazione il preprocessore sostituisce tutte le occorrenze con il valore (o espressione) associato e, non avendo un vero tipo, non c'è type-safety.

```
const int i = a + b;           // valore noto a tempo di esecuzione
constexpr double pi = 3.14159; // valore noto a tempo di compilazione
#define w 12                   // direttiva al preprocessore
```

1.4 Literal

I literal sono particolari variabili costanti senza nome, contenute all'interno delle istruzioni del codice. Sono valori immediati utilizzati solo una volta. Un esempio le stringhe di testo da mandare in output o i valor i con cui le variabili sono inizializzate. I literal hanno un tipo definito automaticamente dal compilatore.

```
constexpr double pi = 3.14159; // 3.14159 e' literal di tipo float
string str = "Hello World";    // "Hello World" e' literal di tipo const char*
```

2 Istruzioni o statement

Le istruzioni o statement sono segmenti di codice che specificano un'azione che l'elaboratore dovrà eseguire al momento dell'esecuzione del programma. In genere terminano con `;` o con `{...}` (nel caso di istruzioni composte). Le direttive al preprocessore non sono istruzioni in quanto vengono risolte in fase di compilazione e non compaiono sull'eseguibile finale. Le istruzioni principali in `C++` sono:

- expression statements: espressioni da valutare
- compound statements: istruzioni composte, in genere seguite da un blocco `{...}`
- selection statements: selezioni (`if-else`, `switch-case`, ...)
- iteration statements: iterazioni (`for`, `while`, `do-while`, ...)
- declaration statements: dichiarazioni, inizializzazioni, definizioni
- try blocks: blocchi `try-catch`

2.1 Dichiarazioni, inizializzazioni, definizioni

Dichiarazioni

La dichiarazione è un'istruzione che introduce un nome in uno scope ne specifica il tipo. Le dichiarazioni possono essere di variabili o di funzioni. Non tutte le dichiarazioni hanno effetto sulla memoria (es. le dichiarazioni di funzioni non alterano la memoria).

```
extern int a; // dichiarazione di una variabile
int somma(int a, int b); // dichiarazione di una funzione
```

Definizioni

La definizione è un'istruzione che specifica completamente l'entità introdotta. Viene specificato il nome, il tipo, eventuali valori di inizializzazione e dettagli implementativi.

```
int a = 1; // definizione di una variabile
int somma(int a, int b) { // definizione di una funzione
    return a + b;
}
```

Inizializzazioni

L'inizializzazione di una variabile è una parte dell'istruzione di dichiarazione o definizione che assegna un valore iniziale alla variabile appena introdotta. È sempre opportuno inizializzare le variabili al momento della loro dichiarazione, altrimenti si potrebbe andare incontro a "bug oscuri".

```
int a; // dichiarazione di una variabile
int a = 1; // dichiarazione di una variabile con inizializzazione
```

Differenza tra dichiarazione e definizione

Nella dichiarazione si introduce soltanto il nome di un'entità all'interno di uno scope, ma questo risulta inutilizzabile senza una successiva definizione. Le variabili soltanto dichiarate non hanno uno spazio riservato in memoria e non possono, quindi, essere utilizzate. Le funzioni soltanto dichiarate hanno solo l'header (o intestazione) e necessitano di una definizione per essere utilizzate.

Nella definizione si specificano tutte le informazioni necessarie per l'entità introdotta, rendendola utilizzabile a pieno nel programma. Nel caso delle variabili viene inclusa anche l'assegnazione di uno spazio in memoria. Per le funzioni viene indicata l'implementazione all'interno di un blocco `{...}`.

2.2 Espressioni

Espressioni

L'espressione è il più piccolo segmento di codice usato per esprimere la computazione. Può essere semplice (literal, variabile, LValue, RValue), oppure composto (operazione con operandi).

```
    "Hello World";  // espressione semplice (literal)
    nome_variabile; // espressione semplice (nome variabile)
    a = b + c;      // espressione composta da operatore e operandi
```

Operatori

Gli operatori possono essere unari, binari o ternari in base al numero di operandi che coinvolgono e si dividono in base alla loro funzione:

- **operatori di assegnamento**

```
    a = b
    a += b  // a = a + b
    a -= b  // a = a - b
    a *= b  // a = a * b
    a /= b  // a = a / b
    a %= b  // a = a % b
    a &= b  // a = a && b
    a |= b  // a = a || b
    a ^= b  // a = a ^ b
    a <<= b // a = a << b
    a >>= b // a = a >> b
```

- **operatori di incremento e decremento**

```
    ++a; // preincremento  -> a = a + 1
    --a; // predecremento  -> a = a - 1
    a++; // postincremento -> a = a + 1
    a--; // postdecremento -> a = a - 1
```

- **operatori aritmetici**

```
    +a      // somma unaria
    -a      // sottrazione unaria (opposto)
    a + b   // somma
    a - b   // differenza
    a * b   // prodotto
    a / b   // divisione
    a % b   // modulo
    ~a      // NOR bitwise
    a & b   // AND bitwise
    a | b   // OR bitwise
    a ^ b   // XOR bitwise
    a << b // shift verso sinistra
    a >> b // shift verso destra
```

- **operatori logici**

```
    !a      // NOT logico
    a && b // AND logico
    a || b // OR logico
```

- **operatori di confronto**

```
    a == b  // uguaglianza
    a != b  // diverso
    a < b   // minore
    a > b   // maggiore
    a <= b  // minore o uguale
    a >= b  // maggiore o uguale
    a <=> b // segno della differenza a-b
```

- **operatori di accesso**

```
a[b]  // accesso in memoria
*a    // dereferenziamento
&a    // referenziamento
a.b   // membro di oggetto
a->b  // membro di puntatore
a.*b  // puntatore al membro di un oggetto
a->*b // puntatore al membro di un puntatore
```

- **altri operatori**

```
a(...)      // chiamata a funzione
a, b        // separazione tra parametri (puo' essere ridefinito)
a ? b : c   // confronto condizionale: return b if a == true, c if a == false
static_cast      // casting
dynamic_cast     // casting
const_cast       // casting
reinterpret_cast // casting
C_style_cast     // casting
new         // allocazione dinamica
delete      // deallocazione
sizeof      // dimensione del tipo di una variabile o oggetto
typeid      // ...
noexcept    // ...
alignof     // ...
```

Ordine di valutazione di un'espressione

In un'operazione non è possibile determinare con esattezza l'ordine di lettura degli operandi. Utilizzando la stessa variabile più volte in un'espressione si potrebbe andare incontro a risultati inattesi.

```
a = b + c       // non e' possibile stabilire se sara' letto prima b o c
a = v[i] + i++ // possibile errore -> viene prima letto v[i] o eseguito i++?
```

Side effect degli operatori

Il side effect di un operatore indica la modifica che l'operatore esegue agli operandi, indipendentemente dal risultato finale dell'operazione.

```
a++     // risultato: a+1, side effect: incremento di a
a -= b  // risultato: a-b, side effect: a = a-b
```

Overloading e significato degli operatori

Grazie all'overloading degli operatori (classi) è possibile che uno stesso operatore abbia diversi significati in base agli operatori su cui è invocato.

```
cout << "Hello"   // operator<< inserisce una stringa nell'output buffer
cout << 4.5       // operator<< inserisce un double nell'output buffer
a + b             // somma tra interi
"Hello" + "World" // giustapposizione di stringhe
```

Rvalue e Lvalue

- Un lvalue è l'elemento a sinistra di un operatore di assegnamento. Viene modificato durante l'esecuzione dell'operazione, per cui deve essere modificabile.
- Un rvalue è l'elemento a destra di un operatore di assegnamento. Non viene modificato (a meno di side effects) per cui può essere anche costante.
- Le variabili possono essere sia lvalue, sia rvalue, i literals, le costanti e i valori restituiti dalle funzioni (a meno di lvalue reference o puntatori) sono solo rvaule.

2.3 Casting

Casting esplicito e implicito

La conversione o casting può avvenire in due modi:

- **casting implicito**: quando non viene specificata l'operazione di casting
- **casting esplicito**: quando si esplicita l'operazione di casting

```
double x = 12 // casting implicito da int a double
double x = static_cast<double>(12) // casting esplicito da int a double
```

Type safety

Il cast implicito può essere di due tipi:

- **type-safe conversion**: quando si esegue la conversione senza perdita di dati, in generale da un tipo con minore capacità a uno con maggiore capacità
- **non-type-safe conversion**: quando si esegue la conversione con perdita di dati, in generale da un tipo con maggiore capacità a uno con minore capacità

Le conversioni non-type-safe (con perdita di dati) ed eventuali overflow non sono segnalati dal compilatore.

Narrowing conversion

Da `C++11` c'è la possibilità di eseguire le conversioni con controllo di narrowing utilizzando le `{}` per indicare al compilatore di segnalare errore nel caso di conversioni insicure:

```
int x (2);    // int x = 2           -> type-safe, nessun errore
int x (2.3);  // int x = 2.3         -> non-type-safe, nessun errore
int x {2};    // int x = 2   + check -> type-safe, nessun errore
int x {2.3};  // int x = 2.3 + check -> non-type-safe, errore
```

Funzioni di casting esplicito

In `C++` sono presenti funzioni per eseguire il cast in modo esplicito tra diversi tipi:

- `static_cast` per conversione tra tipi built-in
- `dynamic_cast` conversione tra classi derivate
- `const_cast` conversione da `const` a `non-const` e viceversa
- `reinterpret_cast` conversione tra due tipi non relazionati (es. lettura di dati binari da file, buffer, o sensore e conversione in dati built-in)

Nel cast esplicito, il compilatore non esegue nessun controllo su overflow o perdita di dati.

3 Scope, namespace e visibilità

Scope

Lo scope è una regione di codice di un programma con la caratteristica che le entità (variabili, funzioni, ...) dichiarate al suo interno hanno validità (esistono) solo all'interno di quello scope. L'esistenza dello scope permette di rendere i nomi locali e riduce i problemi di clash sui nomi. Il nome di una variabile deve essere tanto descrittiva quanto più è esteso lo scope su cui è dichiarata.

Tipi di scope

- **globale**: al di fuori di ogni altro scope
- **scope di classe**: codice all'interno della classe
- **scope locale**: codice all'interno di un blocco `{}`, esempio nel corpo di una funzione
- **scope di statement**: codice all'interno di un blocco `{}` di uno statement (`if-else`, `for`, ...)
- **namespace**: scope con un nome definito globalmente o in un altro namespace

Scope globale

Le entità che di solito sono dichiarate globalmente sono le funzioni e i namespace. Questi sono disponibili all'interno dell'intero file in cui sono dichiarati e in tutti i file che lo includono. È possibile dichiarare variabili globali, solo che sorgono numerosi problemi:

- le variabili globali sono accessibili a chiunque -¿ non c'è incapsulamento
- tutte le funzioni possono modificarne il valore -¿ non c'è incapsulamento
- il debug risulta più difficile perché l'errore potrebbe essere ovunque
- il codice risulta più astruso perché il passaggio di dati tra le funzioni non avviene esplicitamente attraverso parametri

Scope annidati

È possibile dichiarare più scope annidati ad esempio: blocchi di statement annidati (`if-else` annidati), funzioni all'interno di classi (funzioni membro), classi annidate (poco usato), classi in funzioni (poco usato). Non è possibile dichiarare funzioni in funzioni.

3.1 Namespace

Il namespace è uno scope con un nome. Non definisce nessuna entità (funzione o variabile). Risulta molto utile se si utilizzano più librerie con funzioni o classi con nomi uguali: è possibile confinare le dichiarazioni di classi o funzioni con nomi uguali in diversi namespace in modo da non avere collisione tra nomi e richiamarle con il fully-qualified-name `namespace::membro`.

```
namespace Graph_lib {
    struct Color {};
    ...
}
namespace Text_lib {
    struct Color {};
    ...
}
Graph_lib::Color // si riferisce alla struct all'interno del namespace Graph_lib
Text_lib::Color  // si riferisce alla struct all'interno del namespace Text_lib
```

Per evitare di usare sempre il fully-qualified-name si ricorre alle `using declaration` o `using directive`. Queste hanno validità globale, per cui è opportuno non inserirle negli header files, altrimenti verranno importate insieme agli header all'insaputa del programmatore.

```
using std::cout;      // using declaration -> valido solo per std::cout
using namespace std; // using directive -> valido per tutti i membri di std
```

4 Librerie

5 Funzioni

6 User Defined Type

7 Eccezioni

8 Overloading operatori

9 Puntatori

10 Array

11 Allocazione dinamica della memoria

12 Progettare interfaccia

13 Rule of 5

14 Template

15 Compilazione, CMake e Git

15.1 Compilazione

Il processo di compilazione avviene in tre fasi:

1. **preprocessore**: gestisce le direttive al preprocessore, es. `#include, #define, ...`
2. **compilatore**: traduce in codice macchina il singolo file
3. **linker**: risolve i riferimenti e le relazioni tra i diversi file compilati e aggiunge le librerie necessarie

Il processo di compilazione è gestito dal comando `g++ nome_file -o nome_eseguibile -parametri`

15.2 CMake

15.3 Git

16 Ereditarietà

17 Standard Template Library

18 Predicati

19 RAII e Smart Pointer