# Dispense Universitarie: Fondamenti di Programmazione in C per la Cybersecurity

[cite_start]Queste dispense sono un'elaborazione tecnica e accademica delle slide del corso "Cyber Security & Ethical Hacking - Linguaggi di programmazione: C"[cite: 1, 2]. Sono state corrette, ampliate e strutturate per fornire una base solida necessaria per l'analisi di malware e lo sviluppo di exploit.

---

## 1. Introduzione e Classificazione dei Linguaggi di Programmazione

### 1. 📖 Spiegazione Approfondita

La programmazione è il fondamento di ogni componente informatico. [cite_start]Per un Ethical Hacker, comprendere come funziona un programma non è opzionale, ma essenziale per sviluppare una mentalità analitica, analizzare malware e comprendere gli exploit[cite: 18, 19, 20].

I linguaggi di programmazione si classificano storicamente e funzionalmente in tre livelli di astrazione principali:

1. **Linguaggi Macchina:** Costituiscono il livello più basso. Gli algoritmi sono codificati esclusivamente tramite sequenze di bit (0 e 1). [cite_start]Il computer esegue queste istruzioni direttamente senza intermediari, ma per l'essere umano risultano illeggibili e complessi da gestire[cite: 44, 48, 49].
2. **Linguaggi Assembly:** Rappresentano un primo livello di astrazione. Definiscono un set di operazioni elementari utilizzando codici mnemonici (più vicini al linguaggio umano rispetto ai bit). [cite_start]Tuttavia, la CPU non può eseguirli direttamente; necessitano di un traduttore per essere convertiti in linguaggio macchina[cite: 46, 50, 51].
3. **Linguaggi ad Alto Livello:** Sono progettati per essere molto vicini al linguaggio naturale umano (es. inglese). Le istruzioni sono astratte, leggibili e di facile utilizzo. [cite_start]Anche in questo caso, è indispensabile un traduttore per convertire la logica ad alto livello in istruzioni macchina comprensibili dall'hardware[cite: 47, 52, 53].

### 2. 🧠 Analisi Tecnica: Compilatore vs Interprete

[cite_start]La traduzione dal codice sorgente al codice macchina avviene tramite due metodologie distinte[cite: 56, 57]:

- **Compilatore:** Traduce l'**intero** programma scritto in alto livello in un file eseguibile in linguaggio macchina *prima* dell'esecuzione. [cite_start]Il risultato è un file (es. `.exe` o binario ELF) che può essere eseguito indipendentemente dal codice sorgente[cite: 60, 71, 73].
- **Interprete:** Analizza e traduce il programma istruzione per istruzione *durante* l'esecuzione stessa. [cite_start]Non produce un file eseguibile intermedio; il codice sorgente viene processato in tempo reale[cite: 61, 64].

**Definizione Chiave:** Il linguaggio C è un linguaggio **compilato**. Questo implica efficienza e controllo diretto sulle risorse, caratteristiche critiche per lo sviluppo di sistemi operativi ed exploit.

---

# 2. Gli Errori di Programmazione

## 1. 📖 Spiegazione Approfondita

Durante lo sviluppo software, è inevitabile incontrare errori. [cite_start]In C, questi si categorizzano in tre famiglie distinte[cite: 81]:

- **Errori di Sintassi (Compile-time):** Sono violazioni delle regole grammaticali del linguaggio (es. dimenticare un punto e virgola, parentesi non bilanciate). [cite_start]Sono i più semplici da risolvere perché il compilatore li rileva immediatamente, impedendo la creazione dell'eseguibile e segnalando spesso la riga esatta dell'errore[cite: 82, 87, 88].
- **Errori Logici (Design-time):** Si verificano quando la sintassi è corretta, ma l'algoritmo progettato è errato (es. calcolare l'area di un triangolo con la formula del rettangolo). [cite_start]Sono insidiosi perché il compilatore genera l'eseguibile senza avvisi, ma il programma produce risultati sbagliati.[cite: 84, 89, 90].
- **Errori di Esecuzione (Runtime):** Avvengono dopo la compilazione, mentre il programma è in funzione. Sono spesso causati da operazioni illegali come la divisione per zero o l'accesso a memoria non allocata. [cite_start]Il compilatore non può prevederli tutti staticamente, rendendoli difficili da identificare preventivamente[cite: 86, 91, 92].

## 5. ⚠ Focus Esame

- **Domanda:** Quale errore non viene rilevato dal compilatore?
- **Risposta:** Sia gli errori logici che quelli di runtime (es. divisione per zero condizionale) sfuggono al controllo sintattico del compilatore.

---

# 3. Il Linguaggio C: Storia e Caratteristiche

## 1. 📖 Spiegazione Approfondita

Il linguaggio C è stato progettato e implementato da **Dennis Ritchie** presso gli **AT&T Bell Laboratories**. [cite_start]È definito come un linguaggio imperativo ad alto livello, ma con capacità di basso livello uniche[cite: 99].

**Caratteristiche distintive:**

- **Sviluppo OS:** È nato per sviluppare il sistema operativo UNIX. [cite_start]La sua capacità di manipolare direttamente le risorse hardware (memoria, registri) lo rende insostituibile per kernel e driver[cite: 100, 103].
- [cite_start]**Librerie:** Offre un ampio set di librerie (insiemi di funzioni predefinite) per gestire input/output, networking e memoria[cite: 101, 102].
- [cite_start]**Portabilità:** Sebbene nato su UNIX, il codice C ben scritto può essere compilato ed eseguito su diversi sistemi operativi (DOS, Windows, OS X, Linux) con modifiche minime[cite: 103].

---

# 4. Struttura di un Programma in C

## 1. 📖 Spiegazione Approfondita

Un programma C segue una struttura rigida e sequenziale. Analizziamo le componenti fondamentali:

1. **Direttive del Preprocessore:** Le righe che iniziano con `#` (es. `#include <stdio.h>`) vengono elaborate prima della compilazione vera e propria. Dicono al compilatore di caricare librerie esterne. [cite_start]`<stdio.h>` (Standard Input Output) è essenziale per funzioni come leggere da tastiera o scrivere a schermo[cite: 113, 114, 117].
2. **Funzione `main()`:** È il punto di ingresso (entry point) di qualsiasi programma C. L'esecuzione inizia sempre da qui. [cite_start]È definita come `int main()` perché deve restituire un valore intero al sistema operativo al termine dell'esecuzione[cite: 146, 147, 148].
3. **Blocchi di Codice:** Le parentesi graffe `{` e `}` delimitano l'inizio e la fine di una funzione o di un blocco di controllo. [cite_start]Omettere la chiusura è un errore di sintassi[cite: 209, 215].
4. **Terminatore di Istruzione:** Ogni singola istruzione deve terminare con un punto e virgola `;`. [cite_start]La sua assenza impedisce la compilazione[cite: 228, 229].
5. **Commenti:**
    - [cite_start]`/* commento */`: Per commenti multilinea[cite: 175].
    - [cite_start]`// commento`: Per commenti su una singola riga[cite: 176].
6. [cite_start]**Return:** L'istruzione `return 0;` alla fine del main indica al sistema operativo che il programma è terminato con successo (senza errori)[cite: 284].

### 2. 🛠 Sintassi & Comandi (Cleaning)

Esempio corretto di "Hello World" (rispetto agli esempi frammentati delle slide):

```c
#include <stdio.h> // Include la libreria standard I/O

/* Programma principale */
int main() {
    printf("***-----------***\n"); // Stampa decorazione e va a capo
    printf("Hello World!!\n");      // Stampa il messaggio
    printf("***-----------***\n");

    return 0; // Termina con successo
}
```

### 3. 🧠 Analisi Tecnica

- [cite_start]**Caratteri Speciali (Sequenze di Escape):** All'interno delle stringhe, alcuni caratteri modificano il flusso del testo[cite: 311, 313]:
    - `\n`: Newline (ritorno a capo).
    - `\t`: Tabulazione orizzontale.
    - `\b`: Backspace (cancella ultimo carattere).
    - `\a`: Alert (emette un suono di sistema).
    - `\f`: Form feed (salta pagina).

---

## 5. Variabili, Tipi di Dati e Input/Output

### 1. 📖 Spiegazione Approfondita

Le variabili sono contenitori di memoria destinati a ospitare dati. [cite_start]In C, il **tipo** della variabile determina la quantità di memoria allocata e come quei bit vengono interpretati[cite: 322, 334].

[cite_start]**Tabella dei Tipi Principali e Dimensioni (Architettura 32-bit standard):** [cite: 333, 335, 336]

| Tipo | Dimensione (Byte) | Descrizione | Range approssimativo (signed) | Specifier |
|---|---|---|---|---|
| **char** | 1 | Singolo carattere ASCII | -128 a +127 | `%c` |
| **short** | 2 | Intero corto | -32.768 a +32.767 | `%hd` |
| **int** | 4 | Intero standard | ± 2 miliardi (circa) | `%d` |
| **long** | 4 o 8 | Intero lungo (dipende da OS) | Molto ampio (usato in crittografia) | `%ld` |
| **float** | 4 | Reale a virgola mobile | Precisione singola | `%f` |
| **double** | 8 | Reale a doppia precisione | Precisione doppia | `%lf` |

[cite_start]**Nota sulla sicurezza:** I tipi `LONG` sono cruciali nella generazione di chiavi crittografiche per la loro capacità di gestire numeri primi molto grandi[cite: 337].

## 2. 🛠 Sintassi & Comandi (Input/Output)

Le funzioni `printf` e `scanf` (dalla libreria `stdio.h`) gestiscono l'interazione con l'utente.

- **Output (`printf`):** Stampa a video. [cite_start]Usa i placeholder (es. `%d`) per inserire valori variabili nella stringa[cite: 417, 418].
- **Input (`scanf`):** Legge da tastiera. [cite_start]È fondamentale passare l'**indirizzo di memoria** della variabile dove scrivere il dato, usando l'operatore `&` (address-of)[cite: 366, 420].

**Esempio Corretto (Somma di due numeri):**

```c
#include <stdio.h>

int main() {
    int primo, secondo, somma;

    printf("Inserisci il primo numero: ");
    scanf("%d", &primo); // NOTA: & prima della variabile

    printf("Inserisci il secondo numero: ");
    scanf("%d", &secondo);

    somma = primo + secondo;

    printf("La somma è: %d\n", somma);
    return 0;
}
```

### 5. ⚠ Focus Esame

- **Errore comune:** Dimenticare `&` nella `scanf` (es. `scanf("%d", variabile)` invece di `&variabile`). Questo causa spesso un **Segmentation Fault** (errore di runtime) perché il programma cerca di scrivere a un indirizzo di memoria casuale.

---

## 6. Operatori Aritmetici, Relazionali e Logici

### 1. 📖 Spiegazione Approfondita

Il C fornisce set di operatori per manipolare i dati e valutare condizioni.

[cite_start]**Operatori Aritmetici:** [cite: 436]

- `+`, `-`, `*`, `/`: Operazioni standard.
- `%` (Modulo): Restituisce il **resto** della divisione intera. Es. `10 % 7` restituisce `3`. Fondamentale in crittografia e controlli ciclici.
- `++` / `--`: Incremento e decremento unitario (es. `a++` equivale a `a = a + 1`).

[cite_start]**Operatori Relazionali (Confronto):** [cite: 444]

- `==` (Uguale a), `!=` (Diverso da).
- `>`, `<`, `>=`, `<=`.

[cite_start]**Operatori Logici (Booleani):** [cite: 444]

- `&&` (AND): Vero solo se *entrambe* le condizioni sono vere.
- `||` (OR): Vero se *almeno una* condizione è vera.
- `!` (NOT): Inverte il valore di verità.

---

## 7. Istruzioni di Controllo Condizionali

### 1. 📖 Spiegazione Approfondita

[cite_start]Le istruzioni di controllo rompono l'esecuzione sequenziale lineare, permettendo al software di prendere decisioni[cite: 450, 451].

**Costrutto IF-ELSE:** Valuta un'espressione. Se vera, esegue il blocco `if`. [cite_start]Se falsa, esegue il blocco `else` (se presente)[cite: 465].

- *Applicazione:* Evitare errori critici, come la divisione per zero.

[cite_start]**Esempio (Prevenzione crash):** [cite: 469, 470, 471]

```
if (denominatore != 0) {
    float divisione = numeratore / denominatore;
    printf("Risultato: %f", divisione);
} else {
    printf("Errore: Il denominatore non può essere 0");
}
```

**Costrutto SWITCH:** Utile per la "selezione multipla" quando una variabile può assumere valori discreti specifici. [cite_start]È più pulito di molti `if-else` concatenati[cite: 491, 492].

- `case X:` Definisce il blocco per il valore X.
- `break;`: Fondamentale per uscire dallo switch dopo aver eseguito un case. [cite_start]Senza `break`, il codice continuerebbe ad eseguire anche i case successivi (fall-through)[cite: 496].
- [cite_start]`default:` Eseguito se nessuno dei case corrisponde[cite: 497].

---

## 8. Istruzioni di Controllo Iterative (Cicli)

### 1. 📖 Spiegazione Approfondita

I cicli permettono di ripetere un blocco di codice finché una condizione rimane vera.

[cite_start]**Ciclo WHILE:** [cite: 508] Valuta la condizione *prima* di eseguire il codice. Se la condizione è falsa all'inizio, il codice non viene mai eseguito.

[cite_start]**Ciclo DO-WHILE:** [cite: 509, 510] Esegue il codice *prima* e valuta la condizione *dopo*. Garantisce che il blocco venga eseguito **almeno una volta**.

[cite_start]**Ciclo FOR:** [cite: 533, 534] Ideale quando si conosce a priori il numero di iterazioni. Compatta in un'unica riga:

1. Inizializzazione (`int i = 0`)
2. Test condizionale (`i < 10`)
3. Incremento (`i++`)

```
for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

---

## 9. Gli Array (Vettori)

### 1. 📖 Spiegazione Approfondita

[cite_start]Un array è una struttura dati che memorizza una sequenza di elementi dello **stesso tipo** in un blocco di memoria **contiguo** (celle adiacenti)[cite: 549].

[cite_start]**Caratteristiche Fondamentali:** [cite: 551, 555, 556, 557]

- **Omogeneità:** Un array di `int` contiene solo interi; non si possono mischiare tipi.
- **Dimensione Fissa:** La dimensione va dichiarata alla creazione e non può cambiare (in C statico).
- **Indicizzazione:** Si accede agli elementi tramite indice, partendo rigorosamente da **0**. L'ultimo elemento di un array di dimensione N è all'indice N-1.

**Sintassi:** `tipo nome_array[dimensione];` [cite_start]Es: `int numeri[5];` crea spazio per 5 interi[cite: 564, 568].

---

## 10. Le Funzioni

### 1. 📖 Spiegazione Approfondita

Le funzioni sono sottoprogrammi autonomi che ricevono parametri in input, eseguono istruzioni e possono restituire un valore. [cite_start]Favoriscono la modularità e il riutilizzo del codice[cite: 591, 608].

[cite_start]**Struttura:** [cite: 593, 600]

```
tipo_ritorno nome_funzione(tipo param1, tipo param2) {
    // corpo
    return valore;
}
```

- [cite_start]**Void:** Se una funzione non restituisce nulla, il tipo di ritorno è `void`[cite: 686].
- [cite_start]**Chiamata:** Si invoca la funzione nel `main` passando i valori attuali (argomenti)[cite: 613].

---

## 11. Compilazione in ambiente Linux (Kali)

### 1. 🛠 Sintassi & Comandi (Cleaning)

[cite_start]Per trasformare il codice sorgente (`.c`) in un software eseguibile su Kali Linux, si usa la suite `GCC` (GNU Compiler Collection)[cite: 637].

[cite_start]**Procedura Operativa:** [cite: 641, 643]

1. **Scrittura:** Creare il file sorgente (es. `nano esempio.c`).
2. **Compilazione:**

   ```
   gcc esempio.c -o esempio
   ```

   - `esempio.c`: File di input.
   - `-o esempio`: Flag per specificare il nome del file di output (eseguibile).
3. **Esecuzione:**

   ```
   ./esempio
   ```

   - `./`: Indica alla shell di cercare l'eseguibile nella cartella corrente.

---

## 12. I Puntatori

### 1. 📖 Spiegazione Approfondita

[cite_start]I puntatori sono l'argomento più complesso e potente del C. Sono variabili che non contengono un valore numerico (come un `int` normale), ma un **indirizzo di memoria** che punta alla posizione di un'altra variabile[cite: 655, 659].

**Concetti Chiave:**

- [cite_start]**Dichiarazione:** `int *ptr;` (ptr è un puntatore a un intero)[cite: 666].
- **Referenziazione (`&`):** Ottiene l'indirizzo di una variabile.
  - [cite_start]`ptr = &var;` (ptr ora "punta" a var)[cite: 675].
- **Dereferenziazione (`*`):** Accede al *valore* contenuto all'indirizzo puntato.
  - [cite_start]`*ptr = 5;` (Va all'indirizzo puntato da ptr e scrive 5)[cite: 676].

## 3. 🧠 Analisi Tecnica: Puntatori e Array

Esiste una stretta relazione tra array e puntatori. [cite_start]Il nome di un array (senza parentesi quadre) è di fatto un puntatore al suo primo elemento (indice 0)[cite: 705].

**Aritmetica dei Puntatori:** Se `ptr` punta a `vettore[0]`, l'istruzione `ptr++` sposta il puntatore alla cella di memoria successiva (`vettore[1]`). [cite_start]Questo permette di scorrere un array senza usare indici numerici espliciti (`[i]`), ma manipolando direttamente gli indirizzi di memoria[cite: 738, 739].

[cite_start]**Esempio avanzato (ricostruito dalle slide):** [cite: 693, 737]

```
void leggivettore(int *ptr) {
    for(int i=0; i<10; i++) {
        scanf("%d", ptr); // Legge e scrive all'indirizzo corrente
        ptr++;            // Sposta il puntatore al prossimo intero in
memoria
    }
}
```

## 4. 🔗 Cross-Connection Master (Pentesting)

Comprendere i puntatori è fondamentale per l'**Exploit Development** (es. Buffer Overflow).

- In un attacco Buffer Overflow, si sovrascrivono aree di memoria adiacenti superando i limiti di un array.
- Capire come il puntatore scorre in memoria (`ptr++`) e come i dati sono contigui spiega perché un input troppo lungo può sovrascrivere variabili critiche o l'indirizzo di ritorno della funzione (EIP), permettendo l'esecuzione di codice arbitrario.

## 5. ⚠ Focus Esame

- **Domanda:** Cosa succede se incremento un puntatore `int *p` di 1?
- **Risposta:** L'indirizzo di memoria aumenta di **4 byte** (la dimensione del tipo `int`), non di 1 byte. Il puntatore si sposta alla "prossima cella logica".

---