

# M2 Tipi e sicurezza dei tipi

Angelo Gargantini  
informatica III

Capitolo 2 su APPUNTI

# La sicurezza dei tipi nei linguaggi di programmazione

La sicurezza dei tipi in un programma è molto importante

- se l'esecutore (il PC o la macchina virtuale) non riesce a distinguere i tipi di un certo programma può facilmente causare errori
- molti attacchi sfruttano proprio debolezze nel controllo dei tipi di linguaggi diffusi come il C

*"Well-typed programs never go wrong."*

*Robert Milner*

# Tipo

**Tipo:** Insieme di valori omogenei + operazioni che si possono fare

**Esempi:**

- tipi semplici: Integers, String,
- tipi strutturati come classi, ...
- funzioni: int -> bool
  - Funzione che da un intero mi dà un boolean
  - Anche le funzioni e i metodi definisco un tipo

Esempi di non tipi:

- numeri dispari
- array contenenti String e Integer

Dipende però dal linguaggio di programmazione

# A cosa servono i tipi

Per **organizzare** e dare un nome ai concetti  
(documentazione)

- Spesso corrispondenti ai concetti nel dominio del problema che si vuole risolvere
- Indicare l'uso che si vorrà fare di certi identificatori (così il compilatore può controllare)

Per **assicurarsi** che sequenze di bit in memoria siano interpretate correttamente

- Per evitare errori come: 3 + true + "Angelo"

Per **istruire il compilatore** come rappresentare i dati

- Esempio short richiedono meno bit di int

# Errori di tipi a livello Hardware

Confondere **dati** con **programmi**

- Caricando quindi nei registri della CPU possibilmente codici non corretti

*Esempio:* cerco di eseguire un dato chiamando **x()** dove **x** non è una procedura ma un intero

Confondere **tipi di dati semplici**

*Esempio:* eseguo **float\_add(3,4.5)** con 3 int

`float_add`: operazione della CPU che chiama una routine della FPU, se la CPU prende 3 come sequenza di bit float, potrebbe generare un errore hardware

# Errori semantici

Il programma fa qualcosa che non è quello che dovrebbe fare

- Esempio con tipi primitivi: `int _add(3, 4.5)`

In questo caso la sequenza di bit che rappresenta 4.5 può essere interpretato come int ma non sarà uguale come valore

- Esempio con oggetti ed ereditarietà in Java

Sia Quadrato sottoclasse di Figura:

```
class Quadrato extends Figura
```

Se non riesco a distinguere istanze di Qu. e Fig.:

`Figura a1 = new Quadrato()` **OK**

`Quadrato b1 = new Figura()` **NO**: Quadrato potrebbe avere dei metodi in più che potrei invocare ma non trovare perché b1 è una Figura

# Type safety: sicurezza dei tipi

Un linguaggio di programmazione L si dice **type safe** se non esiste programma scritto in L che possa violare la distinzione di tipi in L

Esempi di violazioni dei tipi:

- confondere interi e float
- chiamare una funzione attraverso un intero
- accedere ad una zona di memoria sbagliata (**non memory safe**)

# Sicurezza di alcuni linguaggi

Ecco una tabella che riporta la sicurezza di alcuni linguaggi di programmazione molto diffusi

Safety	Linguaggio	Motivo
Non safe	C e C++	Type cast, aritmetica dei puntatori
Quasi safe	Pascal	Deallocazione esplicita e dangling pointers
Safe	Java, Lisp,Python	Controllo completo dei tipi

# Problemi del C/C++

Il C/C++ ha un sistema dei tipi non sicuro (posso facilmente violare la distinzione di tipi)

Alcuni tipi errori

- Type cast
- Dereferenziazione del null, ...
- Pointer arithmetic
- Accesso alla memoria non valida
  - Violazione **spaziale** come out of bound
  - Violazione **temporale** come dangling pointer

# Quando si fa il type checking?

Tra i linguaggi **type safe** distinguiamo due categorie a seconda del **momento** in cui avviene il controllo dei tipi

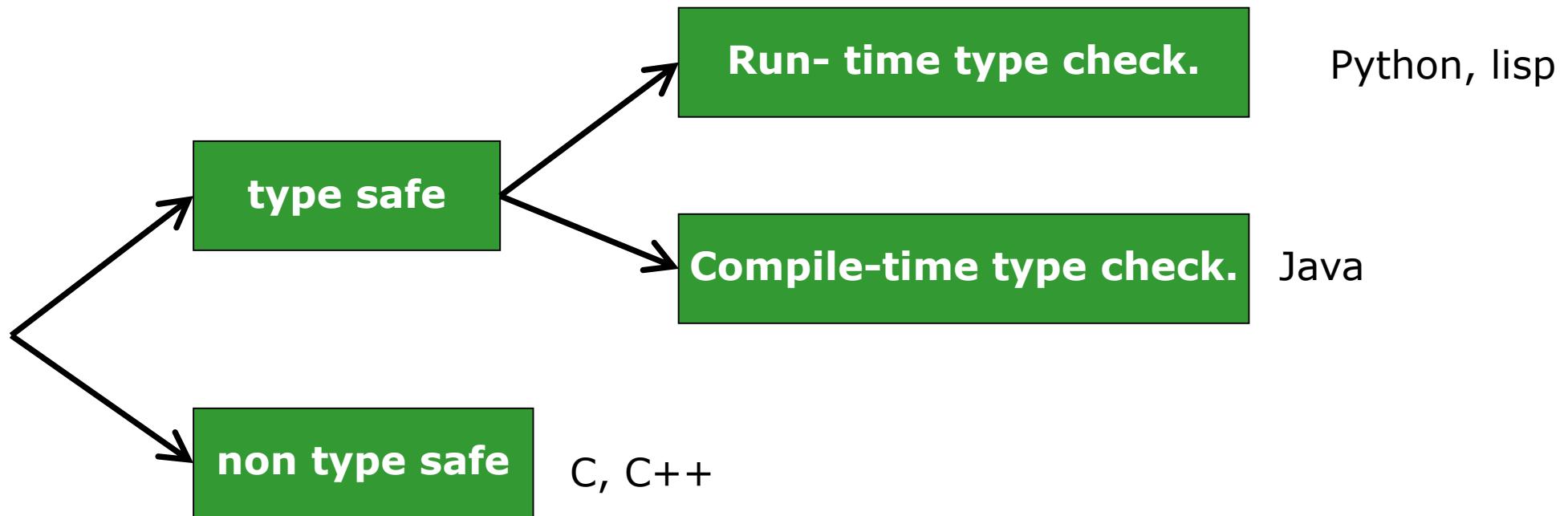
## **run-time type checking**

- Il controllo avviene durante l'esecuzione
- Esempio LISP: quando esegue l'istruzione `(car x)` - che applica `car` a `x` e `car` restituisce il primo elemento di una lista - controlla prima che `x` sia una lista

## **compile time type checking**

- Il controllo avviene durante la compilazione
- Esempio ML: se compila `f(x)` controlla che se `f` sia  $A \rightarrow B$  e `x : A`

# Classificazione dei linguaggi



Vedi syllabus per approfondimento

# Java

Java usa **compile time**, però dove il compilatore non è sicuro della sicurezza dei tipi, introduce un controllo run-time (**conversioni dei tipi controllate**)

considera la seguente istruzione

```
Quadrato a = (Quadrato) b
```

- con b dichiarato di classe Figura (padre di Quad.)
- la conversione al sottotipo Quadrato è corretta solo se **b è effettivamente una istanza di** Quadrato (o di una sottoclasse)
- tale controllo non si può fare in compilazione
- il compilatore introduce un controllo da fare durante l'esecuzione che b sia convertibile a Quadrato

## Pro e contro

Entrambi gli approcci (run-time e compile time) prevengono errori di tipo, però:

- run-time checking rallenta l'esecuzione
  - controlla le conversioni di tipo ogni volta
- compile-time checking limita la flessibilità dei programmi
  - tutte le istruzioni anche non eseguite devono essere corrette
  - Il controllo è **conservativo**

alcuni programmi che non sono corretti compile time sono invece run time corretti

# Flessibilità del run time checking

In Lisp, possiamo scrivere

```
(cond ((< x 10) x) (else (car x))) OK
```

alcune volte ci sarà' errore (catturato dal lisp stesso)

altre no - se x non è < 10 valuto car che si aspetta una lista

In Java, **non** posso scrivere

```
int x;
```

```
if (0>-1) { x++; } else { x = "ciao"; } NO
```

perchè assegna ad x int una String

eppure questo programma è type safe, perchè nessuna esecuzione causa errori di tipo (0 è sempre > -1)

# In sintesi

- Abbiamo visto:
  - l'importanza della sicurezza dei tipi
  - la definizione di linguaggio sicuro nei tipi
  - alcuni linguaggi sono safe altri no
- Ricordate che:
  - il C non è type safe – vedremo alcuni errori tipici
- Inoltre,
  - i linguaggi safe possono effettuare il controllo dei tipi o durante l'esecuzione (run-time) come il LISP o durante la compilazione (compile-time) come Java
  - i pro e contro dei due approcci sono: flessibilità (maggiore con runtime) e efficienza (maggiore con compile time)

# “C is not Safe”

Alcune caratteristiche del linguaggio C e C++ che **possono** dare errori:

1. dereferenziazione del null
2. type cast non controllato
3. pointer arithmetic
4. accesso alla memoria non valida
  - violazione **spaziale** come out of bound
  - violazione **temporale** come dangling pointers

Queste caratteristiche rendono il C molto **flessibile** e **veloce** a discapito della sua sicurezza

- è responsabilità del programmatore stare attento a non introdurre difetti

...one of the main causes of the fall of the Roman Empire was that, lacking zero, they had no way to indicate successful termination of their C programs.

Robert Firth

# Ripasso del C ....

```
#include <stdio.h>
```

Dichiarazioni di import /preprocessore

```
int main()
```

Dichiarazioni di funzioni

```
{
```

```
printf("Hello, World!\n");
```

```
return(0);
```

```
}
```

# Tipi in C

- Per dichiarare una variabile si scrive:  
var\_tipo elenco-variabili-separate-da-virgole ;
- Le variabili globali si definiscono al di sopra della funzione main(), nel seguente modo:

```
short number,sum;  
int bignum, bigsum;  
char letter;  
main() { }
```

- E' possibile preinizializzare una variabile utilizzando = (operatore di assegnazione).

# typedef

- Si possono definire nuovi propri tipi di variabili utilizzando "typedef"(questo risulta utile quando si creano strutture complesse di dati).
- Ad esempio:
  - `typedef float real;`
  - `typedef char letter;`
- variabili dichiarate:
  - `real sum=0.0;`
  - `letter nextletter;`
- Molto usato con le struct

# puntatori

- Un puntatore e' un tipo di dato, una variabile che contiene l'indirizzo in memoria di un'altra variabile. Si possono avere puntatori a qualsiasi tipo di variabile.
  - La dichiarazione di un puntatore include il tipo dell'oggetto a cui il puntatore punta.
  - L' operatore & (operatore unario, o monadico) fornisce l'indirizzo di una variabile.
  - L' operatore \* (operatore indiretto, o non referenziato) da' il contenuto dell'oggetto a cui punta un puntatore.
- Per dichiarare un puntatore ad una variabile, l'istruzione e':
- <tipo> \*<var>;
- es: int \*pointer;

# Esempi

- `int *pointer; /* dichiara pointer come un puntatore a int */`
- `int x=1,y=2;`
- `pointer= &x; /* assegna a pointer l'indirizzo di x */`
- `y= *pointer; /* assegna a y il contenuto di pointer */`
- `x=pointer /* assegna ad x l'indirizzo contenuto in pointer */`
- `*pointer=3; /* assegna al contenuto di pointer il valore 3 */`

# Aritmetica dei puntatori /array

- Un'array di elementi puo' essere pensato come disposto in un insieme di locazioni di memoria consecutive.

```
int a[10], x;  
int *ptr;  
ptr=&a[0]; /* ptr punta all'indirizzo di a[0] */  
x=*ptr; /* x = contenuto di ptr (in questo caso, a[0]) */
```

- A questo punto potremo incrementare ptr con successive istruzioni

`++ptr`

- ma potremo anche avere  
`(ptr + i)`
- che e' equivalente ad `a[i]`, con `i=0,1,2,3...9` .

# malloc

- La funzione malloc risituisce nuova memoria
- `char *malloc(int number_of_bytes)`
  - Es. `char *cp; cp = malloc(100);`
- Se si vuole avere un puntatore ad un altro tipo di dato, si deve utilizzare la coercizione. Inoltre solitamente viene utilizzata la funzione `sizeof()`
  - `int *ip;`
  - `ip = (int *) malloc(100*sizeof(int));`

# Deallocazione esplicita free

- La memoria allocata tramite malloc è persistente: ciò significa che continuerà ad esistere fino alla fine del programma o fino a quando non sarà esplicitamente deallocata dal programmatore (detto anche "liberata"). Questo risultato è ottenuto tramite la funzione free:
- `void free(void *pointer);`
- Alcuni linguaggi non hanno la free ed usano il garbage collector

# Type Cast non Safe

Il C permette la conversione **non controllata** da un tipo ad un altro:

- da un tipo ad un sopratipo con possibile perdita di informazioni.
- da intero ad una funzione per cercare di eseguire una certa locazione di memoria che potrebbe non essere un'istruzione corretta o fare qualcosa di non voluto

*Programma corretto in C ma con type cast non safe:*

```
double d;  
int i;  
...  
i = d; → possibile perdita di informazioni
```

# Dereferenziazione di null

- La dereferenziazione di un puntatore in C non viene controllata
- Se accedo ad una cella puntata da un puntatore nullo ho “segmentation fault”, cioè un **errore del sistema operativo**

*Programma con accesso tramite puntatore null*

```
int main() {  
    int * ptr; ...  
    ptr = NULL;  
    *ptr = 2;  
}
```

# Pointer arithmetic

Mediante l'aritmetica dei puntatori possiamo puntare a zone di memoria con tipi diversi

Esempio:

- se il puntatore  $p$  è definito di tipo  $A^*$
- l'espressione  $* (p+i)$  ha tipo  $A$
- poiché il valore memorizzato a  $p+i$  potrebbe avere qualsiasi tipo
- l'assegnamento  $x = * (p+i)$  con  $x$  di tipo  $A$ , permette di memorizzare un valore di qualsiasi tipo in  $x$

# C non è memory safe

Inoltre mediante i puntatori si può facilmente accedere a memoria in modo scorretto

```
void f(int* p, int i, int v) {  
    p[i] = v;  
}
```

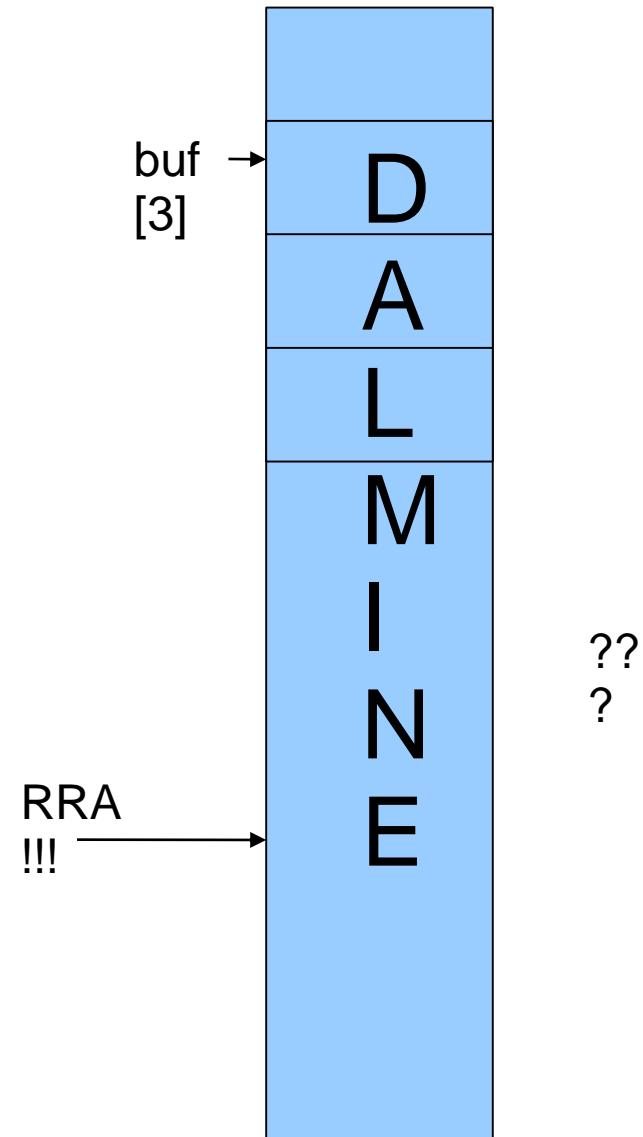
Accedo in questo modo all'indirizzo  $p+i$  e  $p+i$  potrebbe contenere dati importanti o altro codice

- Posso modificare il return address di una chiamata di una procedura ed eseguire altro codice, posso modificare dei diritti o leggere informazioni riservate
- Tipico “**buffer overflow**” /buffer overrun

# Buffer overrun

- A stack-based buffer overrun occurs when a buffer declared on the stack is overwritten by copying data larger than the buffer.
- For example copying the user input directly in the buffer using a `strcpy`,
- Variables declared on the stack are located next to the return address for the function's caller.
  - the result is that the return address for the function gets overwritten by an address chosen by the attacker.

```
strcpy(buf,"DALMINE");
```



# Esempio semplice

```
#include <string.h>

void foo (char *bar) {
    char c[12];
    strcpy(c, bar); // no bounds checking...
}

int main (int argc, char **argv) {
    foo(argv[1]);
}
```

# type cast e violazione memoria

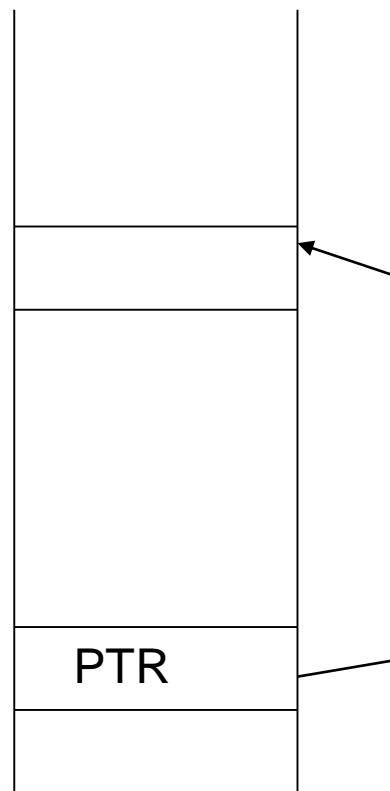
I puntatori in C sono assimilati a interi

Tramite cast di dati interi a puntatori, posso accedere ad una zona di memoria a piacere

*Programma (OK in compilazione)  
con conversione da int a char\**

```
int main() {
    char * PTR;
    PTR = 1000;
    *PTR = 'a';
}
```

1000



# Deallocazione esplicita e Dangling Pointers

In Pascal, C, ... una locazione puntata da un puntatore  $p$  può essere deallocata (liberata) dal programmatore:  $p$  è un “dangling pointer”

Ad esempio in C, faccio il **free** di un puntatore poi continuo ad usarlo

*Un puntatore è **dangling** se punta ad una zona di memoria che è stata liberata per essere riutilizzata*

- Il sistema operativo potrebbe allocare la stessa memoria nuovamente per memorizzare un altro tipo di valore
- Posso continuare ad usare  $p$  per accedere a questa memoria e rompere la type safety

# Uso di free

- Posso usare anche un puntatore dopo averne fatto il free
- Vedi esempio

# Memory leak

- In informatica, un **memory leak** ("perdita o fuoriuscita di memoria") è un particolare tipo di consumo non voluto di memoria dovuto alla mancata deallocazione dalla stessa, di variabili/dati non più utilizzati da parte dei processi. [wikipedia]

# Esempio

Funzione che converte un intero in stringa corrispondente, restituendo il puntatore alla stringa ottenuta:

```
char * itoa(int i){  
    char buf[20];  
    sprintf(buf,"%d",i);  
    return buf;  
}
```

Ricorda sprintf

A cosa punta buf ? buf viene restituito ma punta ad un **array locale che viene deallocated**

# Dangling Pointers sullo stack

Un esempio frequente di errore dovuto a dangling pointers è quando si usano puntatori a celle dello **stack**

Si verifica quando:

- si crea un puntatore p ad una zona A di memoria che è locale ad un metodo (ad esempio variabili locali)
- A è quindi allocata sullo stack
- A viene liberata all'uscita del metodo
- p è a questo punto un dangling pointer

# Esempio in C++

Esempio in C++:

```
struct Point {int x; int y;};
struct Point *newPoint(int x,int y) {
    struct Point result = {x,y};
    return &result;
}
void bar() {
    struct Point *p = newPoint(1,2);
    p -> y = 1234;
}
```

*newPoint* restituisce un puntatore ad un oggetto (*result*) locale: in *bar* *p* è un dangling pointer

# Soluzione

Come si possono evitare dangling pointers?

1. Evitare di puntare zone di memoria sullo stack ed usare la **malloc**:

```
struct Point * result =  
(struct Point*)malloc(sizeof(struct Point))
```

La malloc crea puntatori a zone sicure

Però attenzione che la sua gestione non è automatica come le variabili sullo stack

2. Uso del **garbage collector (gc)** invece che della deallocazione esplicita

- Il gcc marca lui le zone da liberare e che si possono riutilizzare
- Non usando free, il gc recura la memoria

# Cosa fare per avere evitare tali errori?

Se vogliamo scrivere codice safe cosa possiamo fare?

- Scrivere attentamente, progettare prima, documentare, etc.

Se vogliamo essere sicuri che il nostro codice è safe?

- Due soluzioni possibili
  - usare linguaggi type safe (Java, lisp;..) e linguaggi + astratti
  - usare linguaggi come C e dei tools che ci aiutano a rendere i programmi C safe

# In sintesi

- Abbiamo visto alcune fonti di violazioni di sicurezza del C:
  - dereferenziazione non controllata
  - typecast non controllato
  - aritmetica dei puntatori
    - Violazione “spaziale” della memoria, Buffer overflow, ...
  - deallocazione esplicita e dangling pointers
    - Violazione “temporale” della memoria, puntatori allo stack
- Le soluzioni proposte sono:
  - non usare C e passare a Java/C#, ...
  - usare C con tool e librerie che vedremo la prossima lezione



# Programmi sicuri in C

# Svantaggi ad usare linguaggi astratti

C'è un prezzo da pagare se si vogliono usare linguaggi safe come Java, ...

- **Prestazioni inferiori**
  - per il controllo dei limiti nell'accesso agli array, garbage collection per evitare dangling pointers
- **Impiego di maggiore memoria**
  - per tenere informazione sui tipi, sulla dimensione degli array
- **Annotazione dei tipi**
  - maggiore verbosità nelle dichiarazioni
- **Porting** di codice già esistente in C
  - (per quanto Java abbia sintassi simile al C)

# Vantaggi del C

Il C è tutt'oggi usato per molte applicazioni come il sistema operativo, i device drivers

- Ha prestazioni elevate
- Permette la gestione esplicita della memoria
- Permette il controllo della rappresentazione dei dati a basso livello
- Riuso del codice esistente già scritto in C

# Alcune violazioni di sicurezza del C

## Errori “spaziali” di accesso alla memoria

- Out of bound access, buffer overflow, ...

## Errori “temporali” di accesso alla memoria

- Dangling pointers,

## Errori di cast

- Tra diversi tipi di puntatori, tra puntatori e dati interi, tipi unione

## Memory leaks

- Programmi che non rilasciano la memoria anche quando non serve più

# Come rendere il C safe?

1. Tools per l'analisi statica e dinamica per trovare safety violations

- esempio **Purify** della Rational/IBM è un tool per l'analisi **dinamica** per scoprire errori di accesso alla memoria, valgrind
- Analisi statica: cppcheck, lint ...

2. Librerie per rendere il programmi C safe

3. Tools, e linguaggi per prevenire safety violation con due approcci distinti

1. rendere sicuri programmi C : SafeC, CCured
2. varianti safe del C: Cyclone, Vault

# Analisi statica con splint

- <http://www.splint.org/>
- Splint, short for Secure Programming Lint, is a programming tool for statically checking C programs for security vulnerabilities and coding mistakes. Formerly called LCLint, it is a modern version of the Unix lint tool.
- Splint has the ability to interpret special annotations to the source code, which gives it stronger checking than is possible just by looking at the source alone.
- Ultima versione 2007

# Esempio di splint

```
#include <stdio.h>
int main()
{
    char c;
    while (c != 'x');
    {
        c = getchar();
        if (c = 'x')
            return 0;
        switch (c) {
        case '\n':
        case '\r':
            printf("Newline\n");
        default:
            printf("%c", c);
        }
    }
    return 0;
}
```

Variable c used before definition

Suspected infinite loop. No value used in loop test (c) is modified by test or loop body.

Assignment of int to char: c = getchar()

Test expression for if is assignment expression: c = 'x'

Test expression for if not boolean, type char: c = 'x'

Fall through case (no preceding break)

# Analisi dinamica con Purify

approccio analisi dinamica

input programmi C/C++ (qualsiasi)

output eseguibili linkati con Purify

metodo inserimento di controlli per trovare durante  
l'esecuzione errori di accesso alla memoria o memoria non  
rilasciata

pro si applica a codice già esistente

contro rallenta l'esecuzione e non garantisce la scoperta di  
ogni errore

info <http://www-306.ibm.com/software/awdtools/purify/>

valutazione: ★★★

# Analisi con valgrind

- Valgrind is an instrumentation framework for building dynamic analysis tools.
- The Valgrind distribution currently includes six production-quality tools:
  - a memory error detector,
  - two thread error detectors,
  - a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler. It also includes three experimental tools: a heap/stack/global array overrun detector, a second heap profiler that examines how heap blocks are used, and a SimPoint basic block vector generator.
  - It runs on the following platforms: X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, X86/Darwin and AMD64/Darwin (Mac OS X 10.5 and 10.6).
- Is open source ([www.valgrind.org](http://www.valgrind.org))

# Esempio con valgrind

```
#include <stdlib.h>
void f(void)
{
    int* x =
        malloc(10 * sizeof(int));
    x[10] = 0;
}

int main(void)
{
    f();
    return 0;
}
```

Compile your program with `-g` to include debugging information so that Memcheck's error messages include exact line numbers.

```
valgrind --leak-check=yes
myprog
```

Che errori trova ...

Attenti non trova alcuni errori  
(overrun di variabili statiche)

Esempio uso con  
<https://code.google.com/p/drmemory/>

Per windows potete usare drmemory  
Cpp check  
.. Tanti altri

# Librerie Safe per le stringhe

Scopo: evitare i buffer overflow e altri problemi tipici delle stringhe e dei buffer di char

## 1. Safe C String Library

<http://www.zork.org/safestr/>

Autori: Matt Messier e John Viega

## 2. ISO/IEC TR 24731

Meyers, Randy. Specification for Safer, More Secure C Library Functions, ISO/IEC TR 24731, June 6, 2004

[www.open-std.org/jtc1/sc22/wg14/www/docs/n1172.pdf](http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1172.pdf)

Supportato da Microsoft

# SafeC

approccio	traduttore da C a C (C fatto sicuro)
input	programmi C (qualsiasi)
output	programmi C sicuri
metodo	garantisce la cattura delle violazioni di memoria e vari errori run time inserendo dei controlli e aggiungendo informazioni (ad esempio ai puntatori)
pro	si applica a C codice già esistente
contro	rallenta l'esecuzione e aumenta la memoria necessaria
info	<a href="http://www.eecs.umich.edu/~taustin/">http://www.eecs.umich.edu/~taustin/</a>
valutazione:	★

<http://sourceforge.net/projects/safeclib/>

- The Safe C Library provides bound checking memory and string functions per ISO/IEC TR24731. These functions are alternative functions to the existing Standard C Library. The functions verify that output buffers are large enough for the intended result, and return a failure indicator if they are not. Optionally, failing functions call a "runtime-constraint handler" to report the error. Data is never written past the end of an array. All string results are null terminated. In addition, the functions in ISO/IEC TR 24731-1:2007 are re-entrant: they never return pointers to static objects owned by the function.

# CCured

approccio traduttore da C a C (C fatto sicuro)

input programmi C con annotazioni particolari (opzionali)

output programmi C sicuri

metodo abbina analisi statica, controlli dinamici e garbage collector

pro si applica a C codice già esistente con minime modifiche

contro rallenta l'esecuzione

info <http://manju.cs.berkeley.edu/ccured/>

valutazione: ★★

# Cyclone

approccio safe C-like language

input programmi C modificati

output programmi C sicuri

metodo controlli dinamici solo dove necessario e garbage collector

pro minimo overhead di tempo e di memoria

contro richiede di modificare i programmi originali

info <http://cyclone.thelanguage.org/>

valutazione: ★★★



# Vault

approccio safe C-like language

input programmi C modificati

output COM objects

metodo linguaggio astratto simile a Java/C#

pro minimo overhead di tempo e di memoria

controricchiede di riscrivere i programmi originali

info <http://research.microsoft.com/vault/>

valutazione: ★★

# Confronto sui linguaggi

Controllo sui dettagli a basso livello

- SafeC e CCured: pieno utilizzo del C, operazioni limitate sui puntatori
- Cyclone: più restrittivo del C
- Vault: meno efficiente, più astratto

Opzioni sulla gestione della memoria

- Cyclone: diverse opzioni
- Vault: oggetti “lineari” e regioni
- SafeC: malloc() e free() esplicite
- CCured: garbage collection

# Confronto dei costi

Prestazioni

Cyclone  $\approx$  Vault < CCured << SafeC

Aumento memoria

Cyclone  $\approx$  Vault < CCured << SafeC

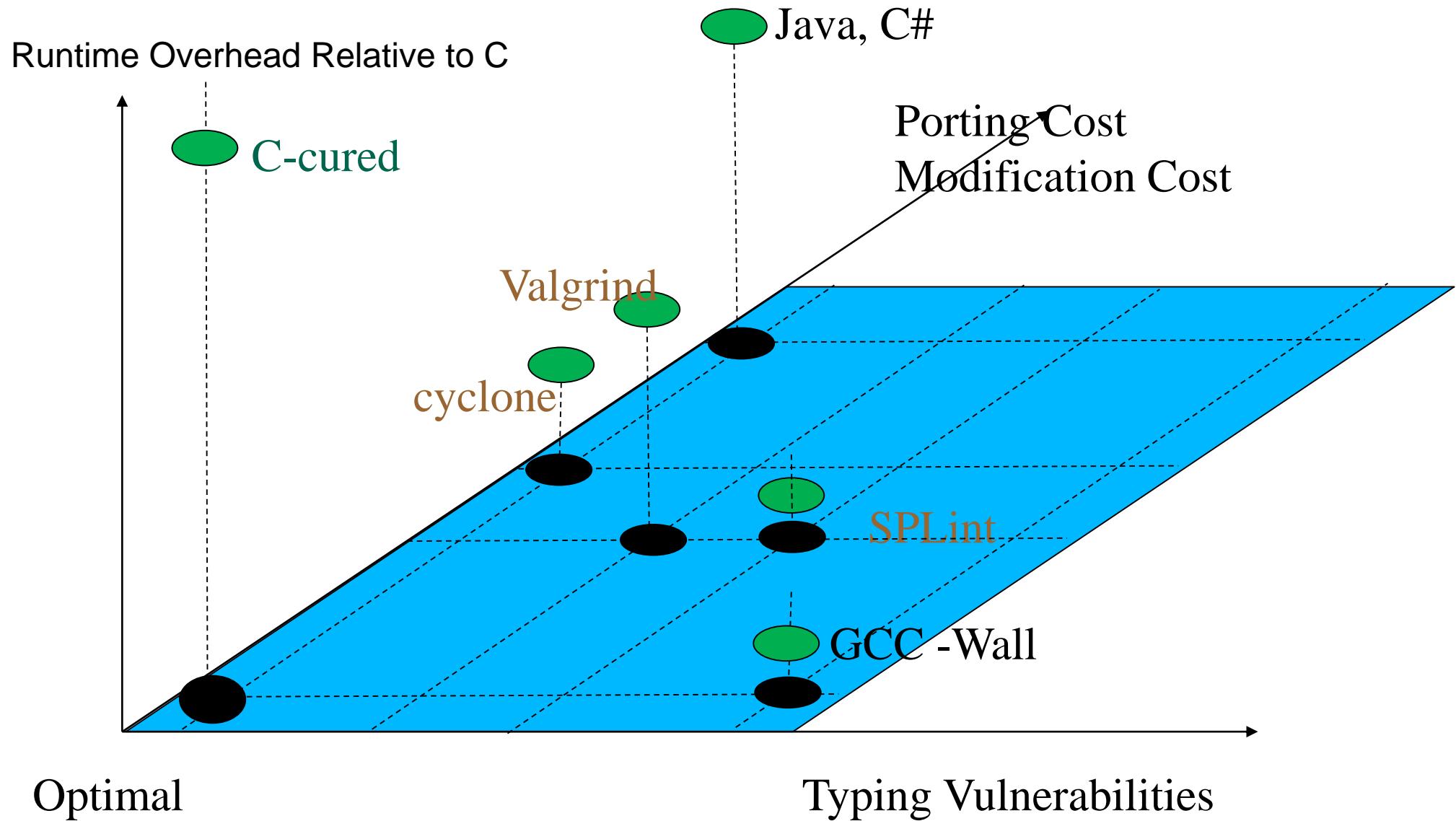
Sforzo per annotare i tipi

SafeC < CCured < Cyclone  $\approx$  Vault

Sforzo per portare codice esistente

SafeC < CCured < Cyclone << Vault

# Conceptual Space



# In sintesi

- Abbiamo visto:
  - quali sono i pro e contro ad usare i linguaggi safe ad alto livello invece che il C
  - possibili modi di rendere il C safe
- Ricordate che:
  - è possibile effettuare l'analisi dinamica con tool come purify
- Per essere certi di avere programmi in C safe abbiamo visto e confrontato questi tool:
  - SafeC, CCured, Cyclone, Vault

