

# *Cyclone*

informatica III - 2007/2008

## Cyclone - Materiale

---

- queste slides
- wikipedia
- la documentazione che si trova sul sito di cyclone:  
<http://cyclone.thelanguage.org/>
- sul sito web c'è un manuale in pdf

## Installazione per linux

---

1. scaricare i sorgenti dal sito di cyclone

- meglio prendere l'ultima versione dal repository cvs se avete gcc versione 4:

```
cvs -d  
:pserver:anonymous@cvs.eecs.harvard.edu:  
/home/cyclone/cvsroot co cyclone
```

2. (se non si usa cvs, scompattare con tar zxf)

3. aprire una shell e fare cd nelle directory dove c'è cyclone

4. ./configure

- (a) se vuoi metterlo in una directory particolare esistente, usa ./configure - -prefix=~ /cyclone

5. make

6. make install

## Installazione per windows

---

Come per linux però:

- installare cygwin (<http://www.cygwin.com/>)
- in windows salvarli in una directory senza spazi
- apri un terminale cygwin e fai configure, make, etc.

Nota: non funziona su AMD64

## Eseguire cyclone

- aggiungi la directory cyclone/bin nel tuo path o chiama cyclone con tutto il percorso
- compilatore e altri tool
- il il compilatore funziona più o meno come un normale compilatore:

```
cyclone opzioni <program.cyc>
-help Print a short description of the
      command-line options.
-v Print compilation stages verbosely.
--version Print version number and exit.
-o file Set the output file name to file.
-O Optimize.
-O2 A higher level of optimization.
-O3 Even more optimization.
...
```

## Primo Esempio

Se vogliamo compilare il solito esempio in C (hello.cyc):

```
#include <stdio.h>
int main() {
    printf("hello, world\n");
    return 0;
}
```

Per compilarlo ed eseguirlo, salva il programma in un file hello.cyc, ed esegui

```
cyclone -o hello hello.cyc
```

Nota che il return del main in Cyclone è obbligatorio

## Puntatori

- Cyclone accetta il normale uso dei puntatori (es1.cyc):

```
#include <stdio.h>
int main() {
    int x = 3;
    int *y = &x; // y punta alla cella di x
    *y = *y + 1;
    printf("%d\n", x);
    return 0;
}
```

in Cyclone è OK: che rischi ci sono?

## Situazioni in cui l'uso dei puntatori è "unsafe"

---

1. assegno un int ad un puntatore e faccio puntare quello che voglio
2. con l'aritmetica porto il puntatore fuori memoria



## NO Cast da int a puntatore

- in C posso scrivere, compilare ed eseguire

```
int main() {
    char * PTR; PTR = 1000; *PTR = 'a';
}
code/cyclone> gcc -o error1 error1.c
error_1.c: In function 'main': error_1.c:4:
warning: assignment makes pointer from integer
        without a cast
code/cyclone> error1
Segmentation fault
```

- In Cyclone NO: *you can't cast an integer to a pointer. Cyclone prevents this because it would let you overwrite arbitrary memory locations. In Cyclone, NULL is a keyword suitable for situations where you would use a (casted) 0 in C.*

## NO aritmetica dei puntatori

---

- in C posso scrivere

```
int main(){
    int x[100];
    int * ptr = x;
    ptr +=20000;
    *ptr = 2;
}
```

```
code/cyclone> gcc -o error2 error2.c
code/cyclone> error2
Segmentation fault
```

In cyclone NO: *You can't do pointer arithmetic on a \* pointer. Pointer arithmetic in C can take a pointer out of bounds, so that when the pointer is eventually dereferenced, it corrupts memory or causes a crash.*

## (auto) NO update di puntatore null

---

```
int main() {  
    int * ptr = 0;  
    *ptr = 2;  
}
```

(error3.c) There is one other way to crash a C program using pointers: you can dereference the NULL pointer or try to update the NULL location. Cyclone prevents this by inserting a null check whenever you dereference a \* pointer (that is, whenever you use the \*, ->, or subscript operation on a pointer.)

## Cosa puoi fare in Cyclone

---

vedremo alcune caratteristiche di cyclone dando esempi di violazione di sicurezza e spiegheremo che cyclone previene tale problema.

1. NULL pointers
2. buffer overflow
3. zero terminated strings
4. bounded pointers
5. dangling pointers

## NULL pointers

Considera la funzione `int getc(FILE *);`

- cosa succede se chiamo `int getc(NULL)`?
- non è specificato
  - se `getc` è scritta in modo che controlla ogni volta che non sia null, ho un rallentamento dell'esecuzione. Se `getc` non fa controlli (come in genere fa) posso avere errori
- in cyclone ho due alternative
  - se non faccio nulla, cyclone usa un suo `getc` definito nella sua libreria e che inserisce un controllo
  - oppure introduce un nuovo tipo di puntatori:

## Puntatori non nulli

---

- Un puntatore non-NULL è indicato da @nonnull, o brevemente mettendo @ al posto \*

```
int * @nonnull ptrA;  
int @ ptrB; // forma breve
```

- Un puntatore @nonnull non è mai NULL:
  - cyclone controlla quando lo usi (ad esempio non puoi assegnargli NULL)
  - quando dereferenzii un puntatore @nonnull o accedi al suo contenuto, cyclone può evitare di controllare che sia non nullo
- utili per documentazione ed efficienza

## Esempio di @nonnull pointer: i file

---

getc in cyclone è dichiarato

```
int getc(FILE *@nonnull); oppure int getc(FILE @)
```

Se apro un file e poi leggo

```
FILE *f = fopen("/etc/passwd", "r");  
// f will be NULL if the file /etc/passwd  
// doesn't exist or can't be read  
int c = getc(f)
```

cyclone inserisce un controllo che non sia nullo e dà un warning.  
Se voglio evitare il warning (ma non il check)

```
int c = getc((FILE *@nonnull)f);
```

## Esempio di @nonnull pointer: i file, alternative

---

Oppure, con check prima del getc

```
FILE @f = (FILE @) fopen("/etc/passwd", "r");  
int c = getc(f)
```

Oppure, con controllo esplicito (cyclone non mette check)

```
if (f == NULL) {  
    fprintf(stderr, "cannot open passwd file!");  
    exit(-1);  
}  
int c = getc(f); //OK so che f non è null
```



## Buffer overflows

---

- per prevenire buffer overflows, cyclone limita l'aritmetica dei puntatori su \*-pointer e su @-pointer
- si può però fare su “fat” puntatori, che mantengono un'informazione aggiuntiva sulla dimensione dell'array (a cui si può accedere con la funzione `numelts()`)
- fat pointers sono denotati con `@fat` o brevemente con ?

## Esempio buffer overflow con puntatori (overflow.c)

```
int strlen(const char *s) {
    int i = 0;
    if (!s) return 0;
    while(s[i] != 0) {
        i++;
    }
    return i;
}
```

- Assume che il buffer termini con 0, altrimenti va avanti oltre la fine del buffer
- Esempio se passo: `buf[] = {'h','e','l','l','o','!'}`; ho un risultato sbagliato
- `s` dovrebbe portarsi dietro la sua dimensione !!!

## versione strlen di cyclone

```
int strlen(const char ?s){
    int i,n;
    if (!s) return 0;
    n = numelts(s);
    for (i = 0; i <n; i++,s++)
        if (!*s) return i;
    return n;
}
```

(fat1.cyc) Quando faccio \*s cyclone inserisce un bound check e posso accedere alla dimensione con size

## Riepilogo: tipo di puntatori

- \* the normal type: no aritmetica !!
- @ the never-NULL pointer
- ? fat pointers: the only type with pointer arithmetic allowed

## Coverzioni

---

- gli array (e stringhe) vengono convertiti a ?-pointers.

```
char a[7] = "angelo";  
strlen(a);
```

- si possono convertire puntatori \* a ? con size 1;
- viceversa (da ? a \*) non c'è problema: casting from ? to \* invokes a bounds check, and casting from ? to @ invokes both a NULL check and a bounds check

## puntatori a puntatori

Il seguente programma scrive i comandi dati sulla stringa di comando in C

```
int main(int argc, char ** argv) {
```

argv è un char \*\*, a pointer to a pointer to a character, which is thought of as an array of an array of characters. In Cyclone:

```
#include <stdio.h>
int main(int argc, char **argv) {
    while (argc > 0) { /* print args space */
        printf("%s ", *argv);
        argc--;
        argv++; // <--- aritmetica OK
    }
    printf("\n"); return 0;
}
```

## Zero-Terminated

- Per rappresentare le stringhe si possono usare questo tipo di puntatori

```
char *@zeroterm
```

- @zeroterm qualifier indicates that the pointer points to a zero-terminated sequence.
- The qualifier is orthogonal to other qualifiers, such as @fat or @nonnull, so you can freely combine them.
- @nozeroterm: This qualifier is present by default on all pointer types except for char pointers. It is used to override the implicit @zeroterm qualifier for char pointers.
- puoi fare aritmetica: quando fai  $x+i$  controlla che in  $x$  tra 0 a  $i-1$  non ci siano null
- attenzione ad usarli: può diventare dispendioso

## Buonded pointers

- A pointer type can also specify that it points to a sequence of a particular (statically known) length using the @numelts qualifier

```
void foo(int *@numelts(4) arr);
```

- Bounded pointers are most often constructed from arrays.
- when you pass an array as a parameter to a function, it is promoted automatically to a pointer, This pointer will have a sequence bound that is the same as the length of the array:

```
int x[4] = {1, 2, 3, 4};  
foo(x);
```

- the parameter x being passed to foo is automatically cast to type int \*@numelts(4), which is the type expected by foo.



## Conversioni

```
void foo(int *@numelts(4) arr);  
int y[8] = {1,2,3,4,5,6,7,8};  
foo(y);
```

- the type of `y` is automatically cast to type `int *@numelts(8)`. Since  $8 \geq 4$ , the call is safe and so Cyclone accepts it but emits a warning “implicit cast to shorter array.”
- the following code will be rejected, because the pointer being passed is too short:

```
int bad[2] = {1,2};  
foo(bad); // does not typecheck
```

## Array cui lunghezza è nota come parametri

---

- in C puoi definire una procedura che prendere un array di const caratteri:

```
int f (char a[20]) {      a[19] = 'g' ; }
```

- però non c'è controllo quando chiamo f. Il seguente codice non da problemi

```
char nome[10] = " " ;      f (nome) ;
```

## Array cui lunghezza non è nota

- bounded pointers can also be used to correlate a pointer to an array whose length is not known statically with a variable that defines it. in C (num is the length of the array pointed at by p)

```
int sum(int num, int *p) {
    int a = 0;
    for (unsigned i = 0; i < num; i++) a += p[i];
}
```

- In Cyclone, this relationship can be expressed by giving sum the following type (the body of the function is the same):

```
int sum(tag_t<'n> num,
        int *@nonnull @numelts(valueof('n)) p) {
```

The type of num is specified as tag\_t<'n>. This simply means that num holds an integer value, called 'n, and the number of elements of p is equal to n

## In breve

in C:

```
int sum(int num, int *p)
```

in cyclone

```
int sum(tag_t num, int p[num]);
```

## Regions: dangling pointers

- **dangling pointer** = un puntatore che punta ad una zona di memoria che è stata deallocata (ad esempio una zona dello stack che viene liberata da un pop)
- Se dereferenzio un dangling pointer ho un **errore** (ma non sempre me ne accorgo)
- Esempio tipico (region.cyc)

```
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;
}
```

# BUG!!!

---

- per trovare questi errori cyclone associa una regione ad ogni puntatore.

*The code has an obvious bug: the function `newPoint` returns a pointer to a locally-defined variable (`result`), even though the storage for that variable is deallocated upon exit from the function. That storage may be re-used (e.g., by a subsequent procedure call) leading to subtle bugs or security problems. For instance, in the code above, after `bar` calls `newPoint`, the storage for the point is reused to store information for the activation record of the call to `foo`. This includes a copy of the pointer `p` and the return address of `foo`. Therefore, it may be that `p->y` actually points to the return address of `foo`. The assignment of the integer 1234 to that location could then result in `foo` “returning” to an arbitrary hunk of code in memory. Nevertheless, the C type-checker readily admits the code.*

## In Cyclone

```
region.cyc:7: returns value of type struct Point *`n
```

*In Cyclone, this code would be rejected by the type-checker to avoid the kind of problems mentioned above. The reason the code is rejected is that the Cyclone compiler tracks object lifetimes and ensures that a pointer to an object can only be dereferenced if that object has not been deallocated.*

'newPoint indica la "regione"

## Soluzione

---

La soluzione sta nell'usare malloc:

```
struct IntPair { int x; int y; };
struct IntPair * same(int z) {
    struct IntPair * ans =
        (struct IntPair *)malloc(sizeof(struct IntPair))
    /* il cast e' opzionale in cyclone*/
    ans->x = z;
    ans->y = z;
    return ans;
}
int main(){
    struct IntPair * p = same(3);
    printf("value %d", (*p).x);
    return 0;
}
```



## Altre caratteristiche

vedi la documentazione

## Esempi/esercizi

---

Dato il seguente codice, riscriverlo in cyclone

```
#include <stdio.h>
void foo(char *s) {
    printf(s);
}
int main(int argc, char **argv) {
    argv++;
    for (argc--; argc >= 0; argc--, argv++)
        foo(*argv);
}
```

## Esempi

Dato il seguente codice, riscriverlo in cyclone

```
#include <stdio.h>
void foo(char ?s) {
    printf(s);
}
int main(int argc, char ??argv) {
    argv++;
    for (argc--; argc >= 0; argc--, argv++)
        foo(*argv);
}
```

## Esercizio / Progetto

---

- prendi un programma scritto da te e convertilo in cyclone
- puoi usare cyclone -port foo.cyc
- oppure scrivi un programma che copia un array di caratteri in un altro array di caratteri e restituisce il nuovo array (clone)