


LEZIONE 1

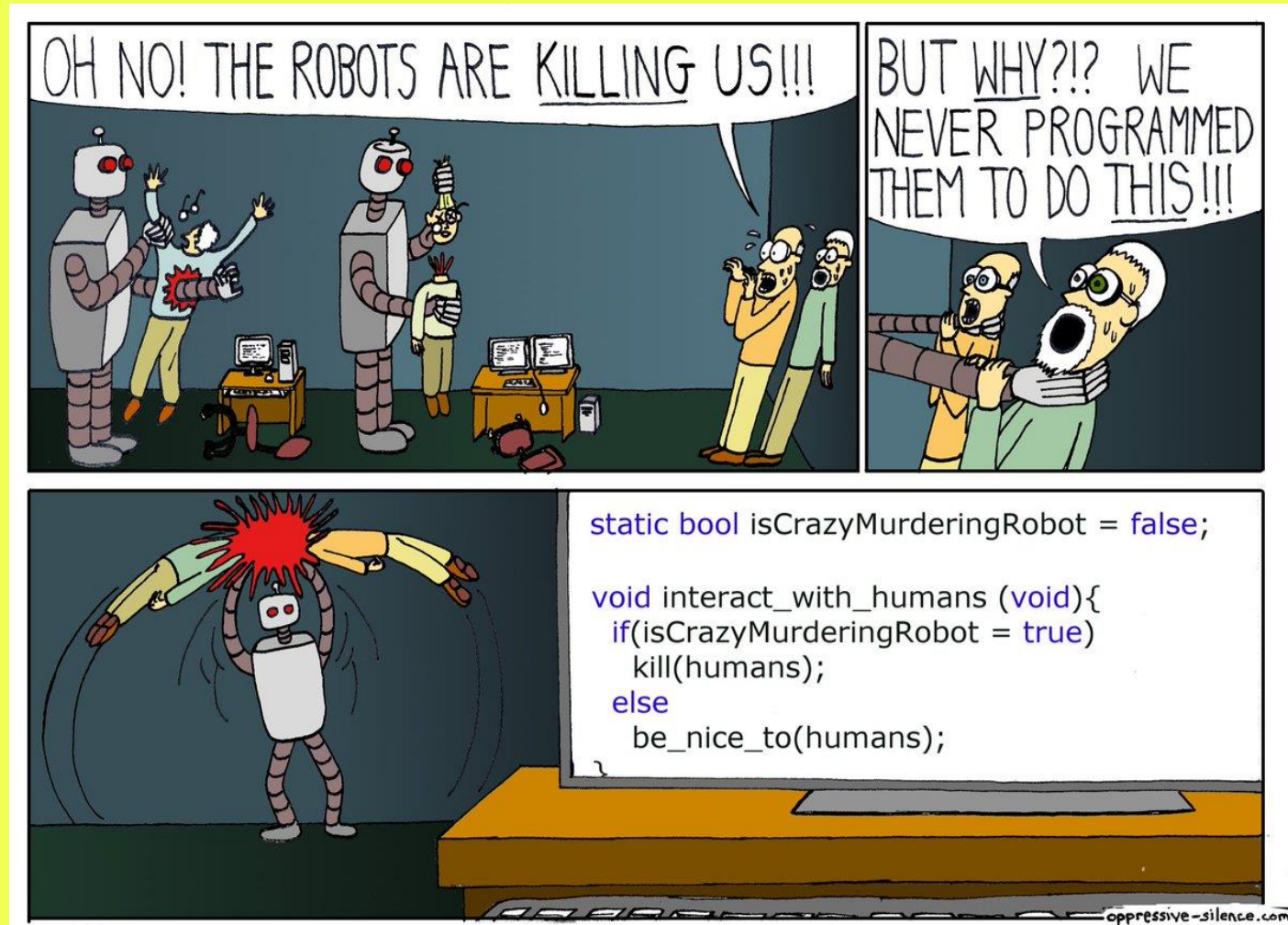
C \rightarrow C++

INTRO

- ▶ Confronto tra C e C++
 - ▶ Ulteriori considerazioni su questi linguaggi
 - ▶ Costrutti che C++ mette a disposizione in fase di scrittura e compilazione per eliminare alcune delle trappole presenti in C
- 

VERSO C++: R-VALUE E L-VALUE

- In C è possibile spararsi su un piede



VERSO C++: R-VALUE E L-VALUE

- ▶ Domande:
 - perché `if (isCrazyMurderingRobot = true)` mi fa entrare nel blocco di condizione soddisfatta?
 - perché `if (x = 7)` mi fa entrare nel blocco di condizione soddisfatta?
- ▶ - come fare in modo che questo macello venga intercettato durante la compilazione?

VERSO C++: R-VALUE E L-VALUE

- ▶ LVALUE (locator value) rappresenta un'entità che occupa una posizione identificabile in memoria [in origine: "values suitable for left-hand-side of assignment"]



RVALUE: ehm... Per esclusione. Es: risultati temporanei di una valutazione di espress.

(<http://eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c>)

- ▶ TL;DR
- ▶ Lvalue: un valore di cui possiamo stabilire l'indirizzo in memoria
- ▶ Rvalue: un valore di cui non possiamo stabilire l'indirizzo in memoria

- ▶ Es:

- ▶ `foo() = 4;` `\\` errore
- `7 = a;` `\\` errore

VERSO C++: R-VALUE E L-VALUE

- ▶ Domande:
 - perché `if (isCrazyMurderingRobot = true)` mi fa entrare nel blocco di condizione soddisfatta?
 - perché `if (x = 7)` mi fa entrare nel blocco di condizione soddisfatta?
- ▶ In C, ogni espressione viene valutata, persino gli assegnamenti. Da standard: "The result of an assignment expression in C is an **rvalue** the same as the value newly assigned to the left hand side of the assignment."
- ▶

```
int main() {  
    int a, b, c;  
    std::cout << (a = 7) << std::endl;    // Stampa 7  
    c = b = a = 12;  
    std::cout << b << std::endl;          // Stampa 12  
}
```

VERSO C++: R-VALUE E L-VALUE

come fare in modo che questo macello venga intercettato durante la compilazione?

- ▶ Nelle espressioni booleane, scrivere il valore che è esplicitamente un RVALUE a sinistra:
- ▶ `if (true == isCrazyMurderingRobot) // Ok!`
- ▶ `if (true = isCrazyMurderingRobot) // Errore di compilazione`
- ▶ `if (isCrazyMurderingRobot == true) // Ok!`
- ▶ `if (isCrazyMurderingRobot = true) // Ok! Ma, ehm...`

VERSO C++: R-VALUE E L-VALUE

- ▶ Pro: sfruttiamo regola di assegnamento lvalue per sollevare errore di compilazione

VERSO C++: R-VALUE E L-VALUE

- Pro: sfruttiamo regola di assegnamento lvalue per sollevare errore di compilazione



VERSO C++: R-VALUE E L-VALUE

- ▶ Pro: sfruttiamo regola di assegnamento lvalue per sollevare errore di compilazione
- ▶ Contro: ...



RIPASSO: PUNTATORI

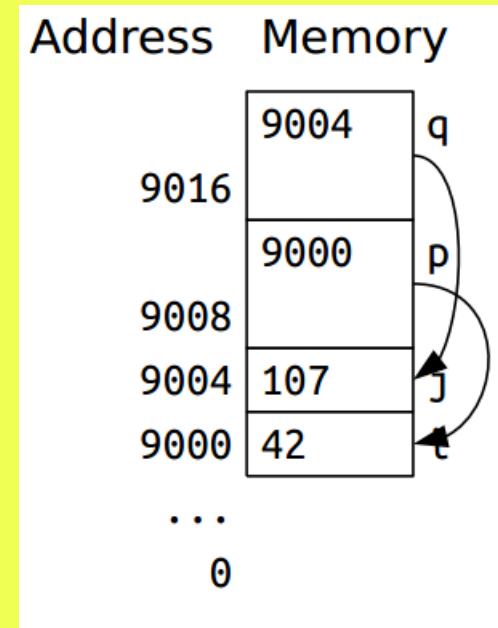
- ▶ Puntatori: tipo di dato. Variabile che contiene indirizzo di memoria di un'altra variabile.
- ▶ Si possono avere puntatori a qualsiasi tipo di variabile e la dichiarazione di un puntatore include il tipo di variabile a cui punta.
- ▶ `int a;` dichiarazione di una variabile di tipo `int`
- ▶ `int *a;` dichiarazione di un puntatore ad una variabile di tipo `int`
- ▶ In C ogni variabile ha due tipi di valore:
 - una locazione
 - valore contenuto in quella locazione
- ▶ Due operatori particolari:
 - `&` (operatore unario) fornisce l'indirizzo di una variabile
 - `*` (operatore indiretto) fornisce il contenuto dell'oggetto a cui punta il puntatore

RIPASSO: PUNTATORI

- ▶ Due operatori particolari:
 - & (operatore unario) fornisce l'indirizzo di una variabile
 - * (operatore indiretto) fornisce il contenuto dell'oggetto a cui punta il puntatore
- ▶ Mnemonico:
 - &: « l'indirizzo di »
 - *: « il valore in »

- ▶

```
int i = 42, j = 107;  
int *p = &i;  
int *q = &j;  
  
printf("%d\n", *p);
```

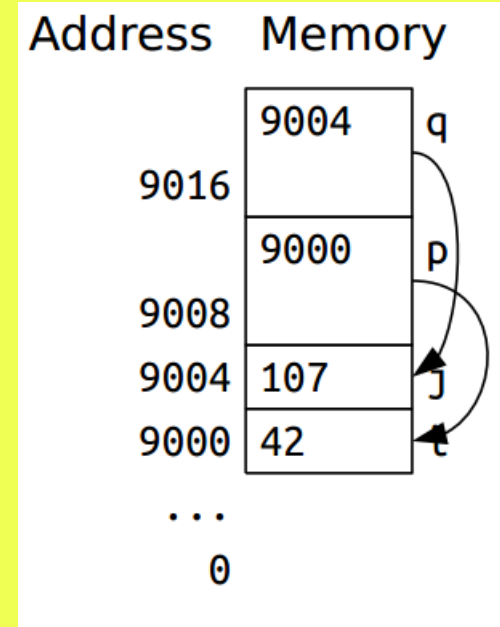


RIPASSO: PUNTATORI

- ▶ `int i = 42, j = 107;`
`int *p = &i;`
`int *q = &j;`
- ▶ `printf("%d\n", *p);`

Cosa eseguono queste istruzioni?

- ▶ `p = q;`
`*p = *q;`
`*p = q;`
`p = *q;`

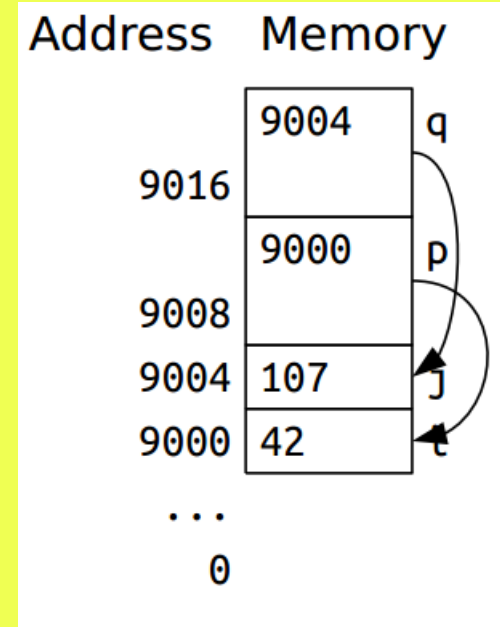


RIPASSO: PUNTATORI

- ▶ `int i = 42, j = 107;`
`int *p = &i;`
`int *q = &j;`
- ▶ `printf("%d\n", *p);`

Cosa eseguono queste istruzioni?

- ▶ `p = q;`

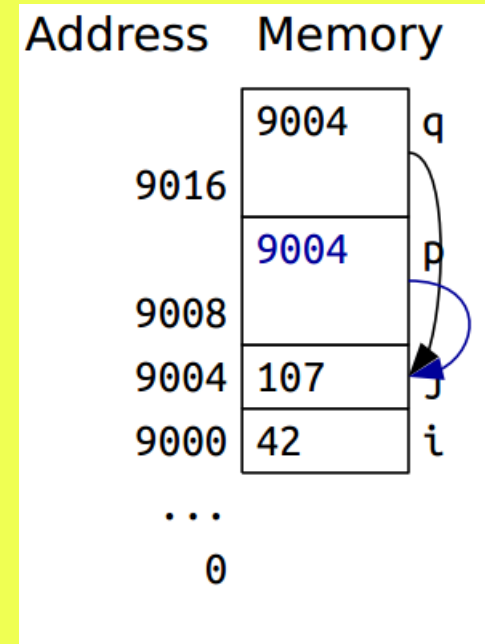


RIPASSO: PUNTATORI

- ▶ `int i = 42, j = 107;`
`int *p = &i;`
`int *q = &j;`
- ▶ `printf("%d\n", *p);`

Cosa eseguono queste istruzioni?

- ▶ `p = q;`
- ▶ `p` e `q` sono «alias»

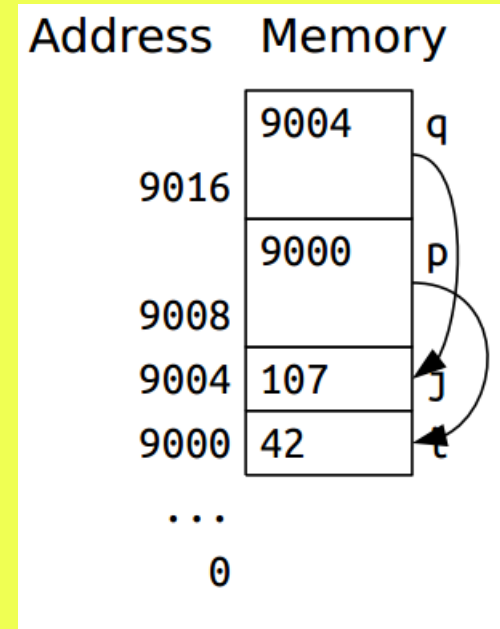


RIPASSO: PUNTATORI

- ▶ `int i = 42, j = 107;`
`int *p = &i;`
`int *q = &j;`
- ▶ `printf("%d\n", *p);`

Cosa eseguono queste istruzioni?

- ▶ `*p = *q;`

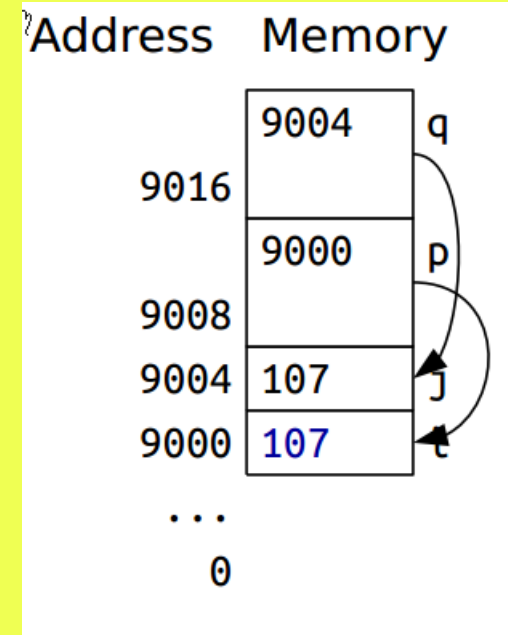


RIPASSO: PUNTATORI

- ▶ `int i = 42, j = 107;`
`int *p = &i;`
`int *q = &j;`
- ▶ `printf("%d\n", *p);`

Cosa eseguono queste istruzioni?

- ▶ `*p = *q;`

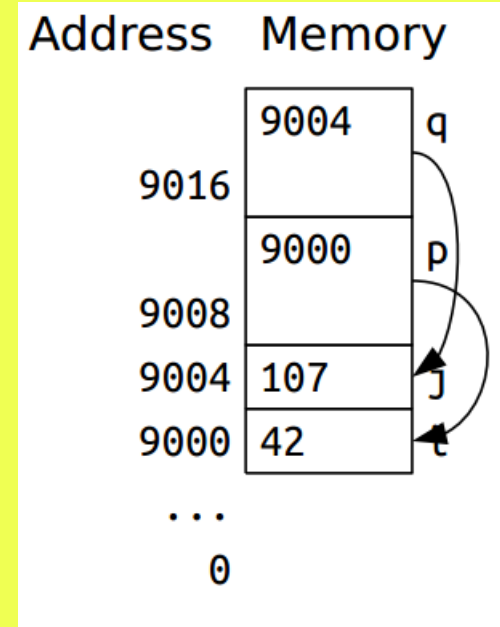


RIPASSO: PUNTATORI

- ▶ `int i = 42, j = 107;`
`int *p = &i;`
`int *q = &j;`
- ▶ `printf("%d\n", *p);`

Cosa eseguono queste istruzioni?

- ▶ `*p = q;`

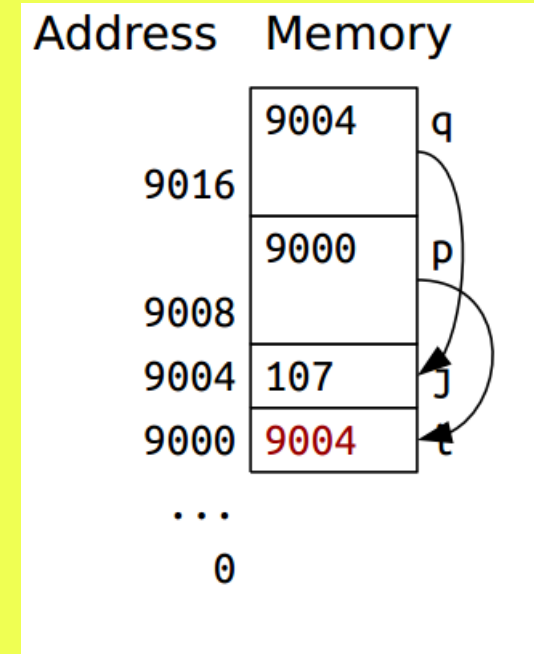


RIPASSO: PUNTATORI

- ▶ `int i = 42, j = 107;`
`int *p = &i;`
`int *q = &j;`
- ▶ `printf("%d\n", *p);`

Cosa eseguono queste istruzioni?

- ▶ `*p = q;`
- ▶ WARNING: assignment makes integer from pointer without a cast

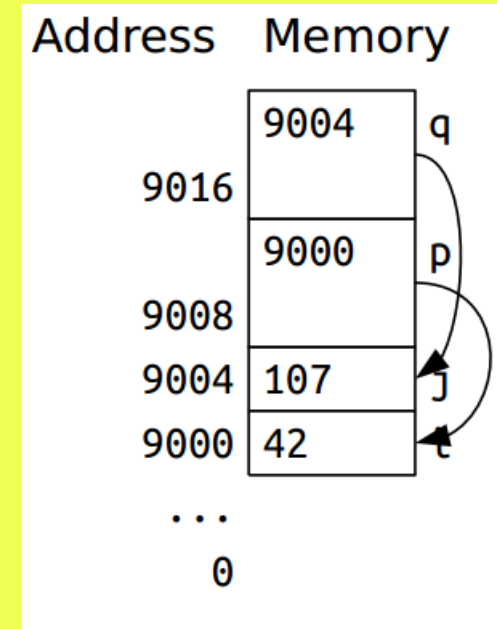


RIPASSO: PUNTATORI

- ▶ `int i = 42, j = 107;`
`int *p = &i;`
`int *q = &j;`
- ▶ `printf("%d\n", *p);`

Cosa eseguono queste istruzioni?

- ▶ `p = *q;`

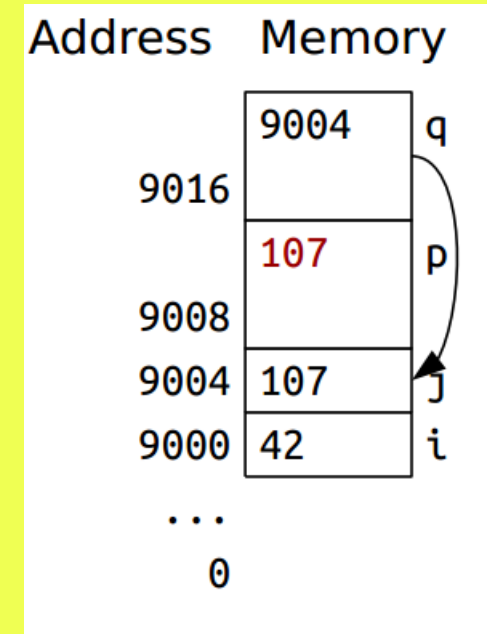


RIPASSO: PUNTATORI

- ▶ `int i = 42, j = 107;`
`int *p = &i;`
`int *q = &j;`
- ▶ `printf("%d\n", *p);`

Cosa eseguono queste istruzioni?

- ▶ `p = *q;`
- ▶ WARNING: assignment makes pointer from integer without a cast
- ▶ Cosa succede se:
`printf("%d\n", *p);`
?

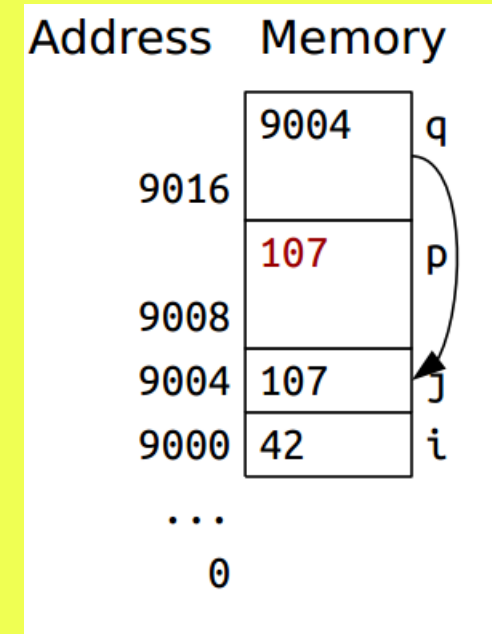


RIPASSO: PUNTATORI

- ▶ `int i = 42, j = 107;`
`int *p = &i;`
`int *q = &j;`
- ▶ `printf("%d\n", *p);`

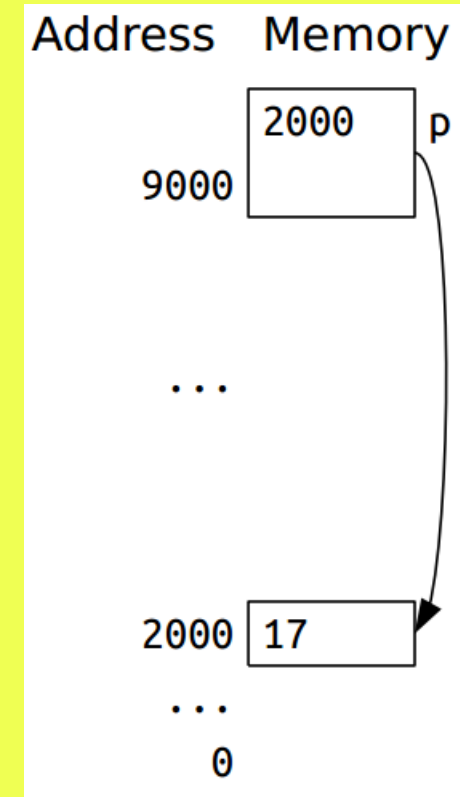
Cosa eseguono queste istruzioni?

- ▶ `p = *q;`
- ▶ WARNING: assignment makes pointer from integer without a cast
- ▶ Cosa succede se:
`printf("%d\n", *p);`
?
- ▶ Probabile Segmentation Fault!



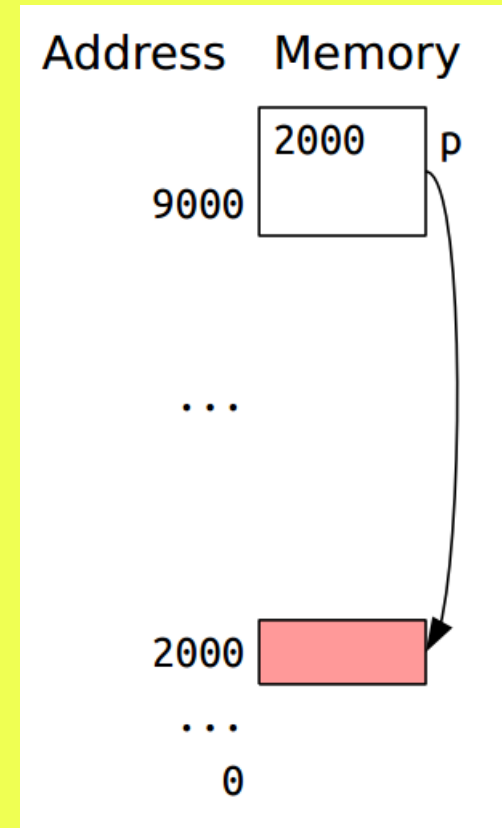
RIPASSO: PUNTATORI

- ▶ `malloc`: permette allocazione di quantità di memoria su heap. Dimensione della memoria allocata viene definita staticamente o a runtime.
- ▶ `malloc(size)`: argomento è numero di byte richiesti; restituisce un puntatore nella memoria heap; memoria allocata rimane valida fino esplicito rilascio.
- ▶ `int *p = (int *) malloc(sizeof(int));`
- ▶ `*p = 17;`



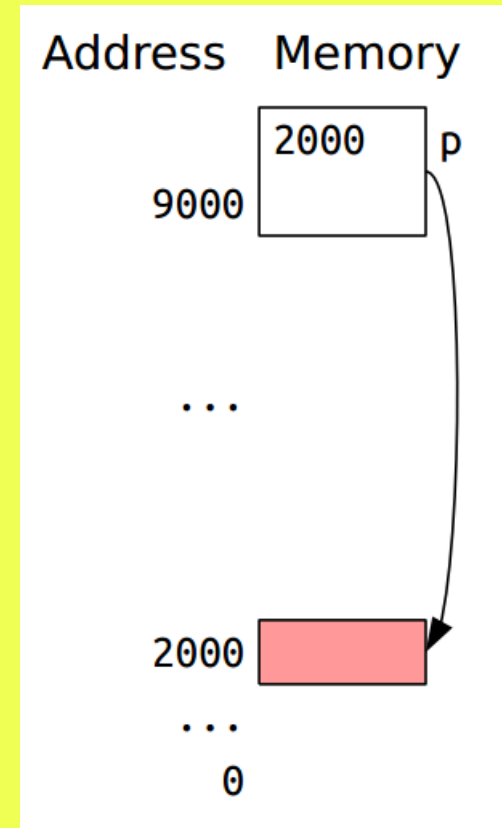
RIPASSO: PUNTATORI

- ▶ `malloc`: permette allocazione di quantità di memoria su heap. Dimensione della memoria allocata viene definita staticamente o a runtime.
- ▶ `Malloc(size)`: argomento è numero di byte richiesti; restituisce un puntatore nella memoria heap; memoria allocata rimane valida fino esplicito rilascio.
- ▶ `int *p = (int *) malloc(sizeof(int));`
- ▶ `*p = 17;`
- ▶ ...
- ▶ `free(p);`



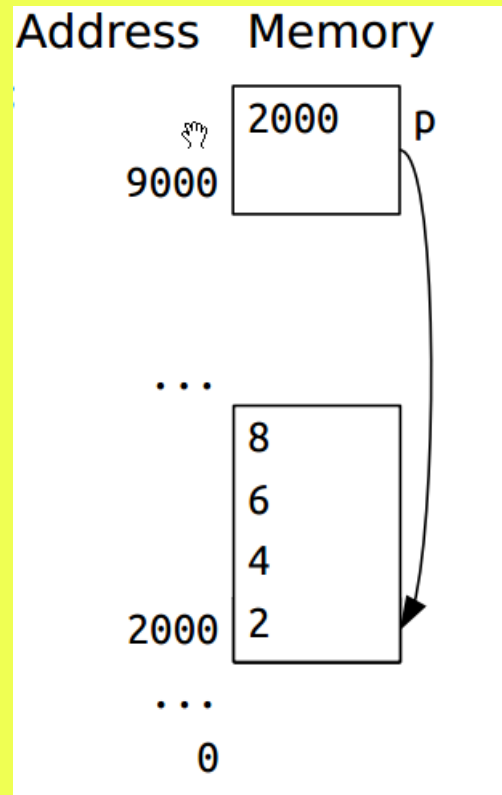
RIPASSO: PUNTATORI

- ▶ `malloc`: permette allocazione di quantità di memoria su heap. Dimensione della memoria allocata viene definita staticamente o a runtime.
- ▶ `Malloc(size)`: argomento è numero di byte richiesti; restituisce un puntatore nella memoria heap; memoria allocata rimane valida fino esplicito rilascio.
- ▶ `int *p = (int *) malloc(sizeof(int));`
- ▶ `*p = 17;`
- ▶ ...
- ▶ `free(p);`
- ▶ Occhio: in C (e C++) la gestione della memoria allocata sullo heap è completamente affidata al programmatore.
E' possibile fare danni che compromettono la stabilità dell'applicazione, fino a minare il funzionamento dell'intero sistema operativo!



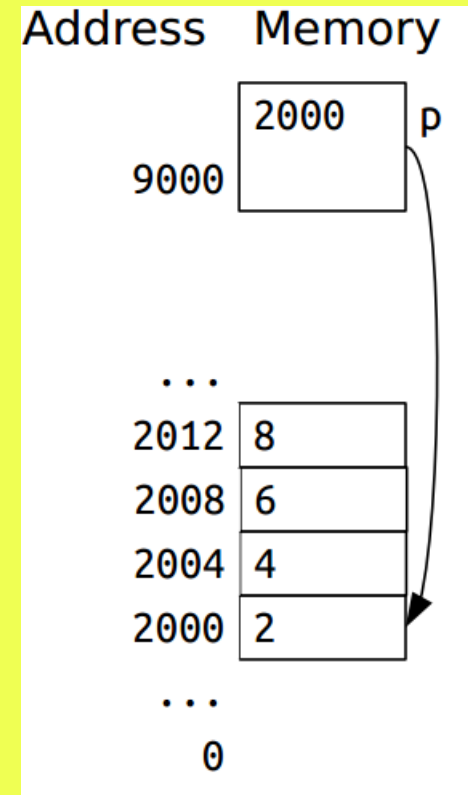
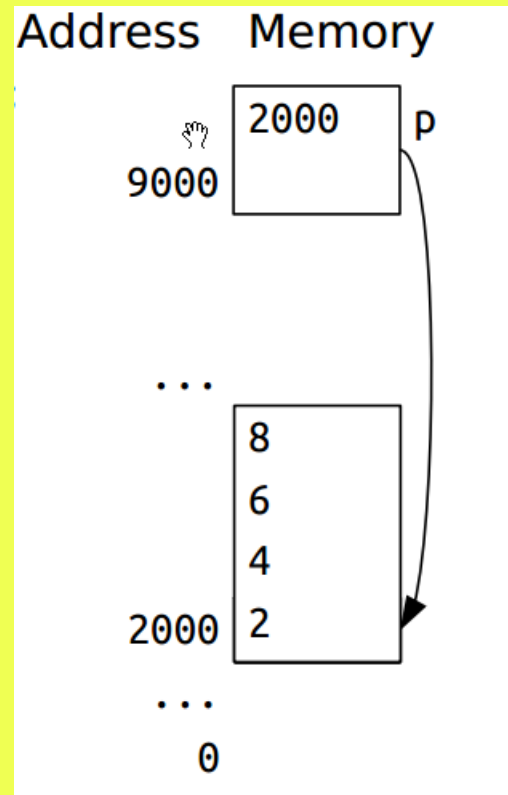
RIPASSO: PUNTATORI

- ▶ Se volessimo allocare più spazio? E gli array?
- ▶ `int *p = (int *) malloc(4 * sizeof(int));`
- ▶ `P[0] = 2;`
`p[1] = 4;`
`p[2] = 6;`
`p[3] = 8;`



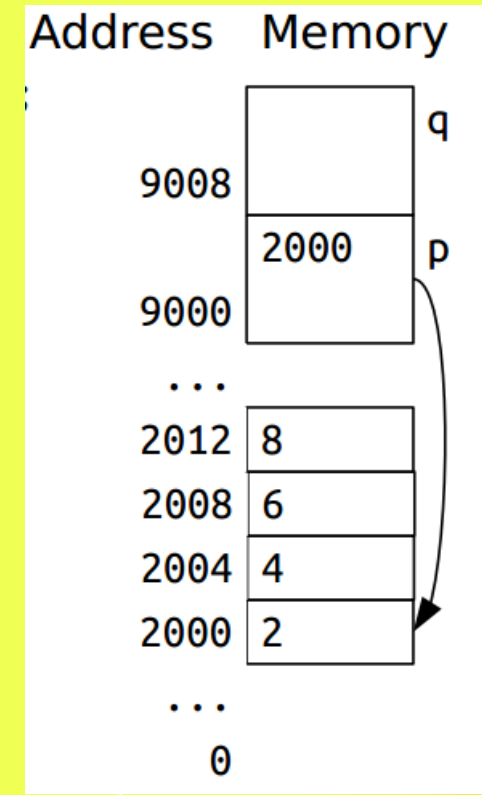
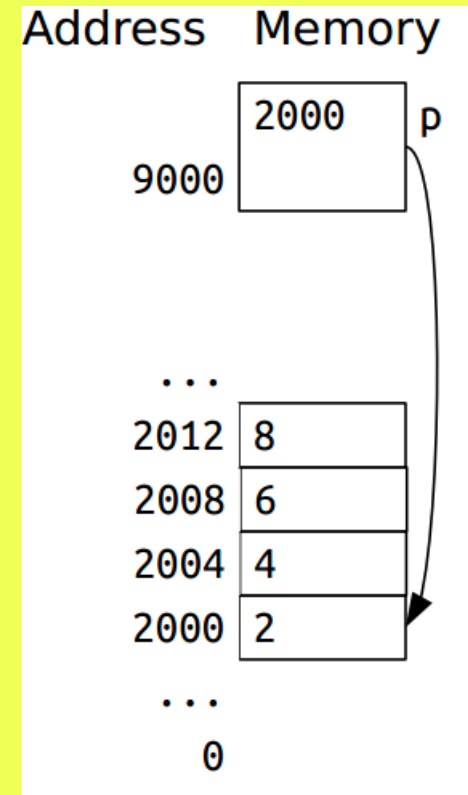
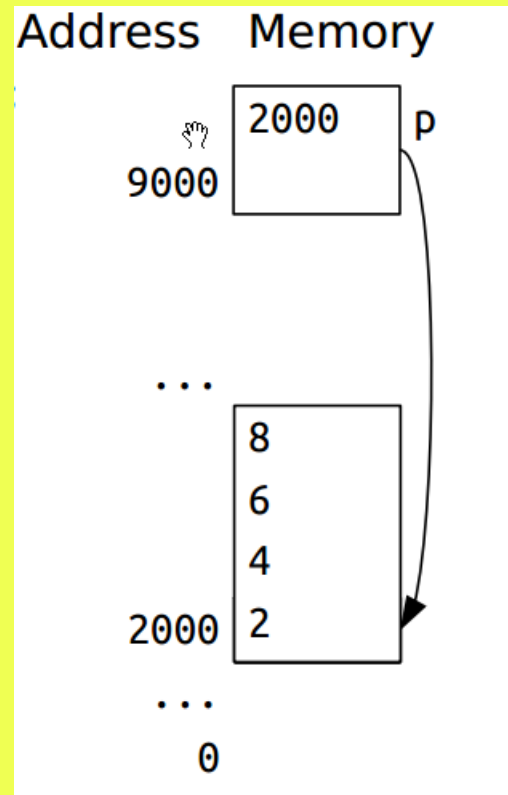
RIPASSO: PUNTATORI

- ▶ Se volessimo allocare più spazio? E gli array?
- ▶ `int *p = (int *) malloc(4 * sizeof(int));`
- ▶ `P[0] = 2;`
`p[1] = 4;`
`p[2] = 6;`
`p[3] = 8;`



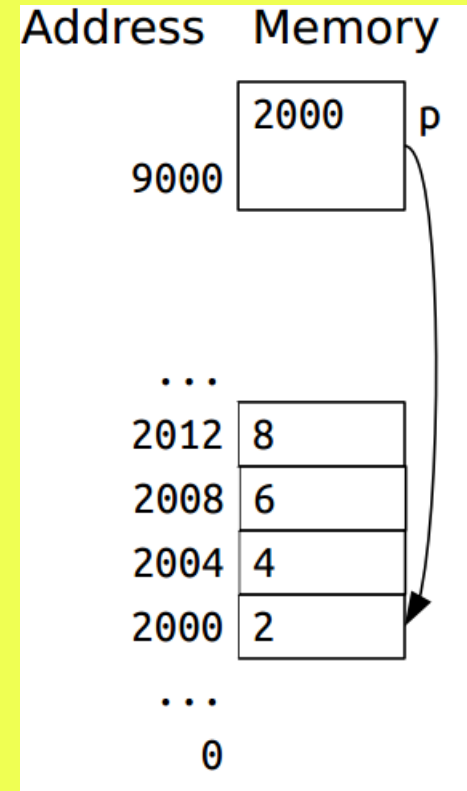
RIPASSO: PUNTATORI

- ▶ Se volessimo allocare più spazio? E gli array?
- ▶ `int *p = (int *) malloc(4 * sizeof(int));`
- ▶ `p[0] = 2;`
`p[1] = 4;`
`p[2] = 6;`
`p[3] = 8;`
- ▶ `int *q = p + 2;`



RIPASSO: PUNTATORI

- ▶ `[]` è un operatore equivalente a `*(.. + ..)` :
- ▶ `P[2]` equivale a `*(p + 2)`
- ▶ `P[0]` equivale a `*(p + 0) == *p`
- ▶ `&p[2]` equivale a `p + 2`
- ▶ `&p[0]` equivale a `p`



RIPASSO: PUNTATORI

- ▶ Array bidimensionali?
- ▶ `int bidim[10][5];`
- ▶ È un array lineare di 50 elementi!
- ▶ `int *bidim = (int *) malloc(10 * 5 * sizeof(int));`
- ▶ `bidim[i][j];`

RIPASSO: PUNTATORI

- ▶ Array bidimensionali?
- ▶ `int bidim[10][5];`
- ▶ È un array lineare di 50 elementi!
- ▶ `int *bidim = (int *) malloc(10 * 5 * sizeof(int));`
- ▶ `bidim[i][j] = *bidim(i * 10 + j) = bidim[i * 10 + j]`
- ▶ Confronta con array di array
- ▶ `int **arrOfArrs = (int**) malloc(5 * sizeof(int*));`
- ▶ `for (int i = 0; i < 5; i++) {`
- ▶ `arrOfArrs[i] = (int*) malloc(15 * sizeof(int));`
- ▶ `}`

RIPASSO: PUNTATORI

- ▶ Micro digressione, ma è necessaria dirla ora!
malloc in C++ non dovrebbe essere usata
- ▶ Al suo posto, new:

```
int *p = (int *) malloc( 4 * sizeof( int ) );
```
- ▶ diventa:

```
int *p = new int[4];  
std::string *s = new std::string( "Ciao!" );
```
- ▶ Gli oggetti allocati con new possono essere usati nella stessa maniera rispetto ad una allocazione con malloc

RIPASSO: PUNTATORI

- ▶ Micro digressione, ma è necessario dirla ora!
malloc in C++ non dovrebbe essere usata
- ▶ Al suo posto, new:

```
int *p = (int *) malloc( 4 * sizeof( int ) );
```
- ▶ diventa:

```
int *p = new int[4];  
std::string *s = new std::string( "Ciao!" );
```
- ▶ Gli oggetti allocati con new possono essere usati nella stessa maniera rispetto ad una allocazione con malloc
- ▶ Per distruggerli, usare keyword delete o delete[]:

```
delete s;  
delete[] p;    // sì, fa differenza!
```
- ▶ Non mischiare free(), delete e delete[]

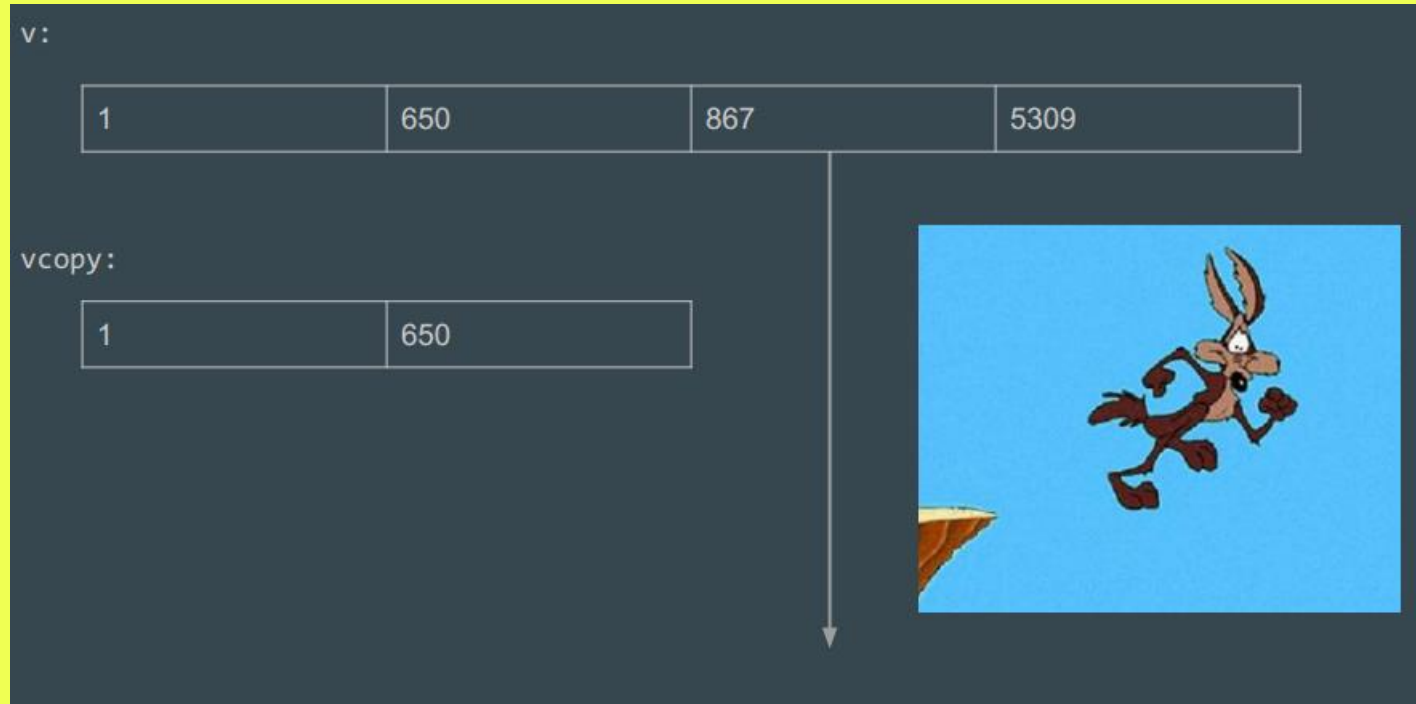
RIPASSO: PUNTATORI

- ▶ - Uso `new` per allocare un oggetto e `delete` per rimuoverlo; nel caso di array, userò `new[]` e `delete[]`. Non mischierò questi operatori!
- ▶ - Uso `new` per allocare un oggetto e `delete` per rimuoverlo; nel caso di array, userò `new[]` e `delete[]`. Non mischierò questi operatori!
- ▶ - Uso `new` per allocare un oggetto e `delete` per rimuoverlo; nel caso di array, userò `new[]` e `delete[]`. Non mischierò questi operatori!
- ▶ - Uso `new` per allocare un oggetto e `delete` per rimuoverlo; nel caso di array, userò `new[]` e `delete[]`. Non mischierò questi operatori!
- ▶ - Uso `new` per allocare un oggetto e `delete` per rimuoverlo; nel caso di array, userò `new[]` e `delete[]`. Non mischierò questi operatori!
- ▶ - Uso `new` per allocare un oggetto e `delete` per rimuoverlo; nel caso di array, userò `new[]` e `delete[]`. Non mischierò questi operatori!
- ▶ - Uso `new` per allocare un oggetto e `delete` per rimuoverlo; nel caso di array, userò `new[]` e `delete[]`. Non mischierò questi operatori!
- ▶ - Uso `new` per allocare un oggetto e `delete` per rimuoverlo; nel caso di array, userò `new[]` e `delete[]`. Non mischierò questi operatori!



RIPASSO: C

- Spararsi su un piede? No run-time array bound check



RIPASSO: C

- ▶ Spararsi su un piede?
- ▶ Ricordiamoci che $P[2]$ equivale a $\ast(p + 2)$
- ▶ `p_array[3]`

RIPASSO: C

- ▶ Spararsi su un piede?
- ▶ Ricordiamoci che $P[2]$ equivale a $*(p + 2)$
- ▶ `p_array[3] = *(p_array + 3)`

RIPASSO: C

- ▶ Spararsi su un piede?
- ▶ Ricordiamoci che $P[2]$ equivale a $*(p + 2)$
- ▶ `p_array[3] = *(p_array + 3) = *(3 + p_array)`

RIPASSO: C

- ▶ Spararsi su un piede?
- ▶ Ricordiamoci che $P[2]$ equivale a $*(p + 2)$
- ▶ `p_array[3] = *(p_array + 3) = *(3 + p_array) = 3[p_array]`

VERSO C++: DEFINE

- ▶ Direttiva del preprocessore che crea una MACRO, ovvero l'associazione di un identificatore o di un identificatore con parametri con una stringa di token.
- ▶ Dopo che la macro è stata definita, il compilatore può sostituire la stringa di token per ogni occorrenza dell'identificatore presente nel file di origine
- ▶ Non definisce una variabile!

VERSO C++: DEFINE

- ▶

```
#define X 8
int main( void ) {
    ++X;
    printf("%d\n", X);
    return 0;
}
```
- ▶ Qual è il valore di X durante l'esecuzione? Cosa viene stampato?

VERSO C++: DEFINE

- ▶

```
#define X 8
int main( void ) {
    ++X;
    printf("%d\n", X);
    return 0;
}
```
- ▶ Qual è il valore di X durante l'esecuzione? Cosa viene stampato?
- ▶ X viene espansa dal preprocessore in ++8, quindi 8 è l'operando dell'operatore unario di incremento ++.

VERSO C++: DEFINE

- ▶

```
#define X 8
int main( void ) {
    ++X;
    printf("%d\n", X);
    return 0;
}
```
- ▶ Qual è il valore di X durante l'esecuzione? Cosa viene stampato?
- ▶ X viene espansa dal preprocessore in ++8, quindi 8 è l'operando dell'operatore unario di incremento ++.
- ▶ L'espressione "++8" è equivalente a "8 = 8 + 1". Per come la macro è stata espansa, 8 è un rvalue, non può essere "a sinistra" in un assegnamento, quindi il codice non compila.

VERSO C++: DEFINE

- ▶ La direttiva di preprocessore `#define` viene usata anche per definire delle funzioni macro
- ▶ `#define SQUARE(x) ((x) * (x))`
- ▶ `SQUARE(10)` espansa in `(10 * 10)`

VERSO C++: DEFINE

- ▶ La direttiva di preprocessore `#define` viene usata anche per definire delle macro
- ▶ `#define SQUARE(x) ((x) * (x))`
- ▶ `SQUARE(10)` espansa in `(10 * 10)`
- ▶ Ma occhio:
- ▶ `int x = 2`
- ▶ `SQUARE(x++)` espansa in `((x++) * (x++))`

VERSO C++: DEFINE

- ▶ La direttiva di preprocessore `#define` viene usata anche per definire delle macro
- ▶ `#define SQUARE(x) ((x) * (x))`
- ▶ `SQUARE(10)` espansa in `(10 * 10)`
- ▶ Ancora:
- ▶ `#define DIV(a,b) a / b`
`printf("25 / (3+2) = %d", DIV(25,3+2)); // Cosa stampa?`

VERSO C++: DEFINE

- ▶ La direttiva di preprocessore `#define` viene usata anche per definire delle macro
- ▶ `#define SQUARE(x) ((x) * (x))`
- ▶ `SQUARE(10)` espansa in `(10 * 10)`
- ▶ Ancora:
 - ▶ `#define DIV(a,b) a / b`
`printf("25 / (3+2) = %d", DIV(25,3+2)); // Cosa stampa?`
- ▶ Espansa in:
 - ▶ `printf("25 / (3+2) = %d", 25 / 3 + 2);`

VERSO C++: DEFINE

► Abuso della direttiva...

► `#define else`
`#define break`

`#define while if`

`#define return if (std::random(1000) < 2) throw std::exception(); else return`

VERSO C++: DEFINE

▶ Abuso della direttiva...

▶ `#define false true`

VERSO C++: DEFINE

► Abuso della direttiva...

► `#define ... se true`



VERSO C++: DEFINE

- ▶ Come evitare guai? `constexpr` e `inline`
- ▶ `constexpr` definisce delle espressioni che possono essere valutate come costanti in fase di compilazione. Qualificatore che può venire assegnato a variabili, oggetti e funzioni.
- ▶ `constexpr int N = 27 + 5;`
- ▶ `#define DIV(a,b) (a/b)`
...
`std::cout << "25 / (3+2) = " << DIV(25,3+2) << std::endl;`
- ▶ `constexpr int div_new(int a, int b) { return a / b; }`
...
`std::cout << "25 / (3+2) = " << div_new(25,3+2) << std::endl;`
- ▶ `int b = 37;`
`std::cout << "25 / (3+2) = " << div_new(b,b+2) << std::endl;`

VERSO C++: DEFINE

- ▶ Come evitare guai? `constexpr` e `inline`
- ▶ `inline` è un qualificatore di funzioni e indica al compilatore di "copiare ed incollare" il corpo della funzione laddove è richiesto, invece di utilizzare il meccanismo di CALL della funzione.
- ▶ Nota: ogni funzione definita con `constexpr` è implicitamente una funzione `inline`.
- ▶ Serve a bypassare i limiti delle funzioni `constexpr`, per es:

```
constexpr int divv(int a, int b) {  
    if (a>b)  
        return a / b;  
    else  
        return 0;  
}
```

Non è una `constexpr` valida (ha due return)

VERSO C++: OCC

- Obfuscated C contest: <http://www.ioccc.org/2015/mills1/prog.c>

```

        #define P(a,b,c) a##b##c
        #include/*+++++*/<urses.h>
        int      c,h,      v,x,y,s,      i,b; int
        main      () {      initscr(      ); P(cb,
        rea,      k) ()      ;///
        P(n,      oec,      ho) (
        )/*      */      ;for      (curs_set(0); s=      x=COLS/2
        ; P(      flu,      shi,      np) (){ timeout(y=c=      v=0);///
        P(c,      lea,      r) ()      ;for      (P (
        mva,      d,      dstr      ) (2,      3+x,
        G) ;      ; P(      usl,      eep,      ) (U)) {//
        P(m,      vad,      dstr      ) ( y      >>8,x,//
        "      "); for(i=LINES; /*      */ i
        ; mvinsch(i,0,0>(~c|i-h-H      &h-i      )?' '
        : (i-      h|h-      i+H)      <0?'|'      : '=' ));
        if((      i=( y      +=v=      getch(      )>0?I:v+
        A)>>8)>=LINES||mvinch(i*=      0<i,      x)!=' '|'|' '
        !=mvinch(i,3+x))break/*&%      &*/;      mvaddstr(y
        >>8,      x,0>v      ?F:B      ); i--s
        /-W;      P(m,      vpr,      intw) (0,
        COLS-9," %u/%u ",(0<i)*      i,b=b<i?i:
        b); refresh(); if(++      c==D){ c
        -=W; h=rand()% (LINES-H-6
        )+2; } } flash(); }}
```

VERSO C++: CONST

- ▶ Imparare a leggere complicate dichiarazioni
 - * puntatore a sempre a sx
 - [] array di sempre a dx
 - () funzione che restituisce sempre a dx
- ▶ Partire da identificatore e leggere ciò che sta immediatamente a destra
Continuare fino a che si incontra una ") " o si finisce la dichiarazione
Ricominciare da dove si è partiti e proseguire a sx fino a " ("
Ripetere 2 e 3!
- ▶ Es.
- ▶

```
int *p[];  
int *( *func() )();  
double (*( *fun_one)( char *, double )) [9][20];
```


VERSO C++: CONST

- ▶ `const`: uno dei qualificatori più importanti di tutto il linguaggio (lo vedremo più avanti... const-correctness!)
- ▶ Qualificatore che modifica tipo immediatamente alla sua sinistra
- ▶

```
int i = 0;  
int const j = 4;  
const int k = 123;
```
- ▶ Anche qui è utile regola «right-left»
- ▶

```
int * const i;  
int const * i;  
const int * i;  
const int * const i;
```

VERSO C++: CONST

- ▶ C possiede 4 type qualifier: `const`, `volatile`, `restrict`, `_Atomic`, mentre C++ solo i primi 2
- ▶ `volatile`: indica un valore che potrebbe cambiare tra diversi accessi. In tal caso, compilatore non ottimizza accesso a tale valore

```
▶ static int counter;  
void funzione(void) {  
    counter = 0;  
    while (counter != 255)  
        {};  
}
```

VERSO C++: CONST

- ▶ C possiede 4 type qualifier: `const`, `volatile`, `restrict`, `_Atomic`, mentre C++ solo i primi 2
- ▶ `volatile`: indica un valore che potrebbe cambiare tra diversi accessi. In tal caso, compilatore non ottimizza accesso a tale valore

```
▶ static int counter;  
void funzione(void) {  
    counter = 0;  
    while (counter != 255)  
        {};  
}
```

```
static int counter;  
void funzione_ottimizzata(void) {  
    counter = 0;  
    while (true)  
        {};  
}
```

- ▶ Originariamente questo avveniva durante accesso a periferiche mappate nello spazio di memoria.

VERSO C++: STORAGE CLASS

- ▶ Storage class specifiers: insieme allo scope, definiscono la durata e il linking di ogni oggetto.
- ▶ Storage duration:
 - automatic: oggetto allocato all'inizio dello scope e deallocato alla fine
- ▶ - static: oggetto allocato ad inizio esecuzione del programma e deallocato alla sua conclusione. Può esistere solo un'istanza di tale oggetto nel corso dell'esecuzione!
- ▶ - thread: oggetto allocato ad inizio esecuzione del thread e deallocato alla sua conclusione. Ogni thread ha la propria istanza di tale oggetto
- ▶ - dynamic: oggetto allocato e deallocato dinamicamente nel corso dell'esecuzione