# INFORMATICA GENERALE II

Ingegneria delle Telecomunicazioni
Università di Trento
A.A. 2003/2004
II Bimestre

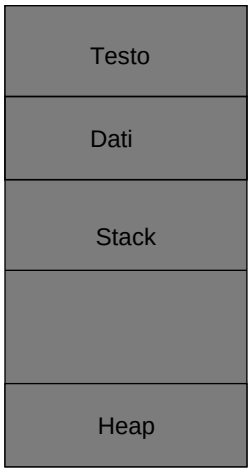
Marco Roveri
roveri@irst.itc.it

Memoria e Puntatori

## Struttura della memoria di un programma

- Testo: contiene il programma. Dimensione dipende dalla dimensione del programma.
- Dati: contiene variabili globali e statiche. La dimensione dipende dal programma.
- Stack: contiene argomenti delle funzioni, punti di ritorno, variabili locali ad una funzione…. Di solito ha una dimensione fissa massima.
- Heap: contiene area dati dinamica. La dimensione è limitata dal sistema operativo (e dalla memoria fisica disponibile).

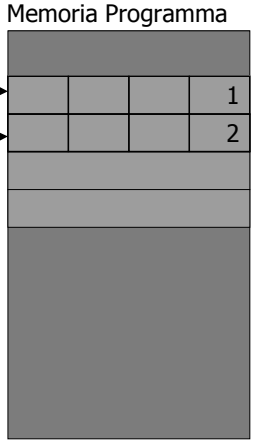
# Struttura della memoria di un programma




# Programmi, variabili, memoria

```
int main () {
 int x = 1;
 int y = x +1;

 cout << “x = “ << x << endl;
 cout << “y = “ << y << endl;
 cout << endl;
 return 0;
}
```



## Programmi, variabili, memoria

- Ogni volta che si dichiara una variabile, viene allocata (riservata) una zona di memoria per essa.
- La grandezza di questa zona dipende dal tipo della variabile.
- Tutte la variabili di un certo tipo occupano esattamente lo stesso numero di byte.
- Esempio:
  - Le variabili di tipo char occupano un solo byte.
  - Gli interi occupano 4 byte.
  - Le variabili di tipo double occupano 8 byte.

## Programmi, variabili, memoria

- Per sapere quali byte una variabile occupa occorrono due numeri:
  - La posizione del primo byte (detto *indirizzo).*
  - Il numero di byte occupati.
- Il C++ mette a disposizione primitive per conoscere questi due valori.
- Esiste la possibilità di scrivere programmi senza sapere quanto spazio occupa una variabile.
- Esistono delle situazioni in cui è necessario (gestione di dati non noti a priori, e.g. array, …).

## Programmi, variabili, memoria

- Per sapere l'indirizzo di una variabile si usa l'operatore **&**

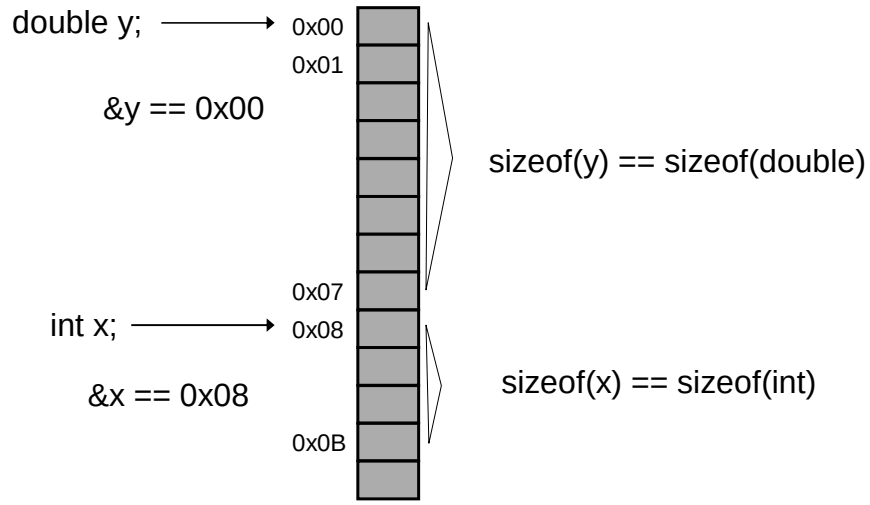
  &x rappresenta indirizzo della variabile x
- Per sapere il numero di byte occupati da una variabile si usa l'operatore **sizeof**

  sizeof(x)  bytes occupati da x

## Programmi, variabili, memoria

- Sintassi:
  - **&**var // restituisce indirizzo della
    // variabile *var*
  - **sizeof**(var) // ritorna numero byte
    // occupati dalla variabile *var*
  - **sizeof**(*type*) // ritorna il numero di byte
    // occupati da una variabile di
    // tipo *type* (e.g. **int**, **double**, **char**, …)

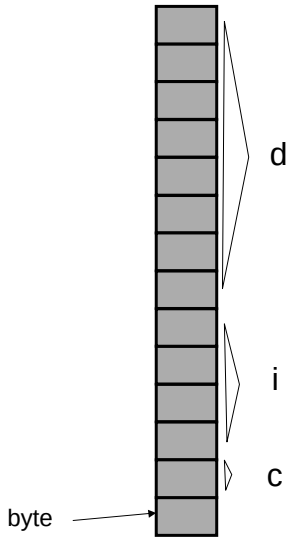
# Programmi, variabili, memoria




# Programmi, variabili, memoria

```
int main () {
  double d;
  int i;
  char c;

  cout << “double = “ << sizeof(d);
  cout << “ add = ” << &d << endl;
  cout << “int = “ << sizeof(i);
  cout << “ add = “ << &i << endl;
  cout << “char = “ << sizeof(c);
  cout << “ add = “ << &c << endl;
  return 0;
}
```



## Programmi, variabili, memoria

- È importante notare la differenza tra il *valore* di una variabile e il suo *indirizzo*.
  - L'indirizzo di una variabile è l'indirizzo del primo byte della zona di memoria occupata dalla variabile.
  - Il valore di una variabile è il contenuto di tale zona.

## Programmi, variabili, memoria

- Esempio:

```
int main () {
  int x = 10;
  double c = `a`;

  cout "Indirizzo di x = " << &x << " valore di x = " << x;
  cout << endl;
  cout "Indirizzo di c = " << &c << " valore di c = " << c;
  cout << endl;
  return 0;
}
```

# Esercizi

- Provare ad implementare per ogni tipo noto un programma simile al precedente che stampi indirizzo e valore di una variabile.
- Quale è il risultato dell'esecuzione del seguente frammento di codice C++?

```
int  main () {
  int x = 10;
  cout << "x = " << x << " &x = " << &x << endl;
   x = 20;
  cout << "x = " << x << " &x = " << &x << endl;
  return 0;
}
```

# Puntatori

- L'indirizzo di una variabile di un tipo *T* viene detto *puntatore a T.*
  - &x è un puntatore ad un intero (x è di tipo intero)
- In C++ si possono dichiarare delle variabili di tipo puntatore.
- Il tipo di una variabile puntatore a T è *T* *
- Ad ogni tipo che è possibile definire in C++ è associato il corrispondente tipo puntatore.

# Puntatori

- Una variabile *puntatore* rappresenta l'indirizzo di un'altra variabile o funzione.
- Hanno come valore gli indirizzi di memoria di locazioni di memoria.
- Il tipo puntatore è un tipo come tutti gli altri, quindi la sua dichiarazione avviene nel modo solito.
- Sintassi

  *tipo * identificativo;*

  Esempio:

  int * p;

# Puntatori

- Le variabili puntatore possono essere confrontate, assegnate come qualunque altra variabile.

```
int main () {
  int x = 10;
  int *y, *z; // notare differenza tra int * y, z;

  y = &x;  z = y;
  if (z == &x)  cout << "Ok\n";
  else cout << "Ko\n";
  return 0;
}
```

# Operatore di *dereference*

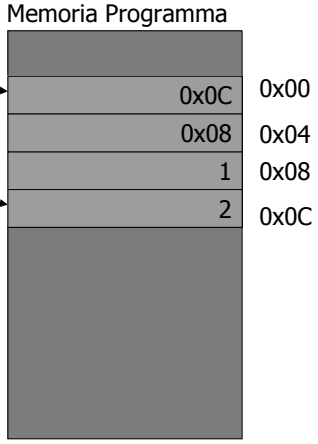
- Per accedere all'oggetto puntato da una variabile puntatore occorre usare operatore di *dereference* "*****"

```
int x = 1;          // variabile di tipo intero
int *px;            // variabile di tipo puntatore
px = &x;            // assegno a px l'indirizzo di x
*px = *px + 1; // incrementa di uno il contenuto
               // della memoria puntata da px
```

# Variabili puntatori e memoria

```
int main () {
 int * px, int * py;
 int x = 1;
 int y = x +1;

 px = &x;
 py = &y;
 cout << "x = " << *px;
 cout << "px = " << px << endl;
 cout << "y = " << *py;
 cout << "py = " << py << endl;
 return 0;
}
```



# Variabili puntatori e memoria

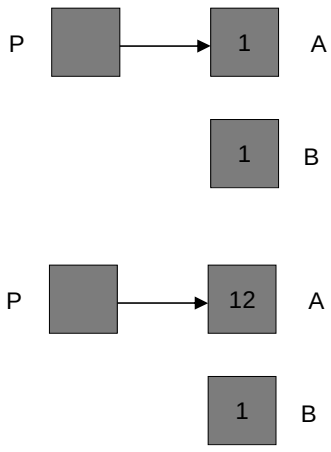
```
int main () {
 int * p, num;

 p = &num;
 *p = 100;
 cout << num << ‘ ‘;
 (*p)++;
 cout << num << ‘ ‘;
 (*p)--;
 cout << num << ‘\n‘;
 return 0;
}
```

# Copia del valore e copia dell’indirizzo

```
int main () {
  int a, b, *p;

  a = 1;    // ad a assegniamo valore 1
  b = a;    // copiamo in b il valore di a
  p = &a;  // l’indirizzo di a è copiato in p
  a = 12;  // cambiamo il valore di a
  // quale è il valore di *p e di b?
  cout << “ *p vale = “ << *p << endl;
  cout << “ b vale = “ << b << endl;
  return 0;
}
```



## Memoria e puntatori

- Ad una variabile puntatore viene associato una spazio di memoria atto a contenere un indirizzo di memoria, ma non viene riservato spazio di memoria per l'oggetto puntato.
- Lo spazio allocato per una variabile di tipo puntatore è sempre uguale, indipendentemente dal tipo dell'oggetto puntato.
- Per inizializzare una variabile puntatore ad un indirizzo costante è necessario effettuare un casting (conversione esplicita):

```
double *px=(double *)321;
```

## Gestione dinamica della memoria

- Quando non si puo stabilire a priori (in maniera statica) la dimensione delle strutture dati, occorre gestire la memoria *dinamicamente* durante l'esecuzione del programma.
- La gestione dinamica della memoria consente di allocare porzioni di memoria nella ***heap***, ovvero in un'area di memoria esterna allo stack di esecuzione del programma.
- L'accesso a questa area avviene tramite puntatori.

## Allocazione dinamica della memoria

- L'allocazione avviene mediante l'operatore ***new***, che alloca un'area di memoria atta a contenere un oggetto del tipo specificato, e ritorna un puntatore a tale area di memoria.
- Sintassi:

  ***new*** *tipo;*

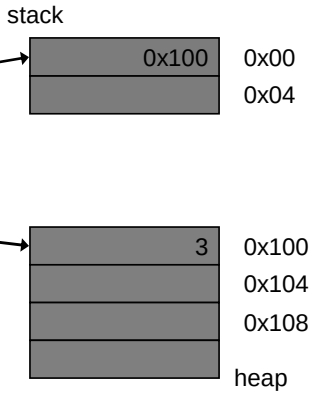
  ***new*** *tipo [dimensione]; // (per gli array)*

## Allocazione dinamica della memoria

- Esempio:

```
int main () {
  int *p;

  p = new int;
  *p = 3;
  cout << "p = " << p;
  cout <<  " *p = " << *p << endl;
  return 0;
 }
```



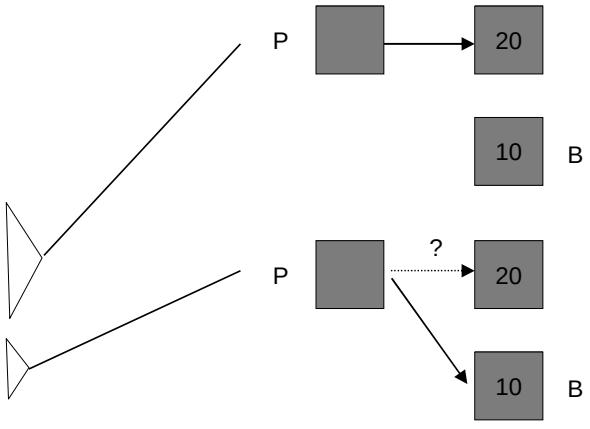
# Allocazione dinamica della memoria

- Una variabile creata dinamicamente resta allocata finchè:
  - il programma non termina
  - non viene esplicitamente deallocata
- La memoria deallocata viene resa disponibile al programma per allocazioni successive, ma la dimensione della heap *non* diminuisce.
- La memoria allocata dinamicamente **deve** essere deallocata quando non più utilizzata.
- La non deallocazione causa il cosiddetto problema del *memory leak*, e può risultare non più disponibile al programma e agli altri programmi.

# Allocazione dinamica della memoria

```
int main () {
  int *p; int b;

  p = new int;
  *p = 20;
  b = 10;
  p = &b;
  return 0;
}
```



## Allocazione dinamica della memoria

- La deallocazione esplicita di una variabile si effettua con l'operatore ***delete***
- Sintassi:

  **delete** var;

  **delete** var [dimensione]; // (per gli array)
- Esempio:
  - Nel programma precedente aggiungere l'istruzione "**delete** p;".
  - Dove la devo posizionare per evitare memory leak?

## Allocazione dinamica della memoria

- È un errore deallocare memoria non precedentemente allocata mediante l'operatore new.
  - Esempio:

```
void main () {
   int x, *px;
   x = 2;
   px = &x;
   delete px;
}
```

  - Se eseguito genera un *core dump.*

# Puntatori a Puntatori

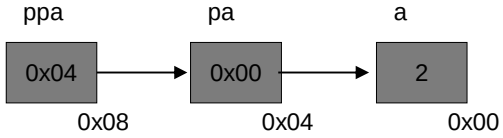
- Una variabile puntatore è una variabile con un tipo (similmente a qualunque altra variabile), per cui è possibile definire puntatori a tali variabili.
- Il suo indirizzo è un puntatore ad un puntatore.
- Sintassi:
  - int **p; // puntatore a puntatore ad intero
  - char **c // puntatore a puntatore a carattere

# Puntatori a Puntatori

- Esempio:

```
void main () {
  int a, *pa, ** ppa;
  a = 9; pa = &a; ppa = &pa;
  cout << “Ind. di a = “ << &a << “ valore di a = “ << a << endl;
  cout << “Ind. di di pa = “ << &pa << “ valore di pa = “ <<  pa << endl;
  cout << “Ind. di ppa = “ << &ppa << “ valore di ppa = “ << ppa << endl’;
}
```

- Valore di ppa coincide con indirizzo di pa.
- Valore di pa coincide con indirizzo di a.



# Funzioni: passaggio valore/riferimento

```
void swap_v(int x, int y) {
   int z = y;
   y = x;
   x = z;
  cout << “Indirizzo di x = “ << &x << endl;
  cout << “Indirizzo di y = “ << &y << endl;
}

void swap_r(int &x, int &y) {
   int z = y;
   y = x;
   x = z;
  cout << “Indirizzo di x = “ << &x << endl;
  cout << “Indirizzo di y = “ << &y << endl;
}
```

```
int main() {
  int x = 0; y = 1;
  cout << “Indirzzo di x = “ << &x << endl;
  cout << “Indirizzo di x = “ << &y << endl;
  swap_v(x,y);
  swap_r(x,y);
}
```

# Funzioni: passaggio valore/riferimento

- La funzione swap_r quando invocata, si vede che gli indirizzi corrispodenti ai parametri formali corrispondono agli indirizzi delle corrispondenti variabili nella procedura padre.
- Questo è il motivo per cui lo chiamiamo passaggio per riferimento, le variabili si *riferiscono* alle variabili della procedura padre. *Non viene copiato il valore* nello stack di attivazione, ma I parametri formali sono dei semplici “*alias*” (sinonimi) per l’area di memoria a cui puntano.
- Quindi risulta evidente che eventuali modifiche a parametri formali passati per riferimento possono comportare modifiche del valore.

## Vantaggi memoria dinamica

- Gestione efficiente delle risorse in modo da allocare lo spazio realmente necessario.
- Creazione di strutture dinamiche (es. array a dimensione variabile, liste, alberi, grafi, …)