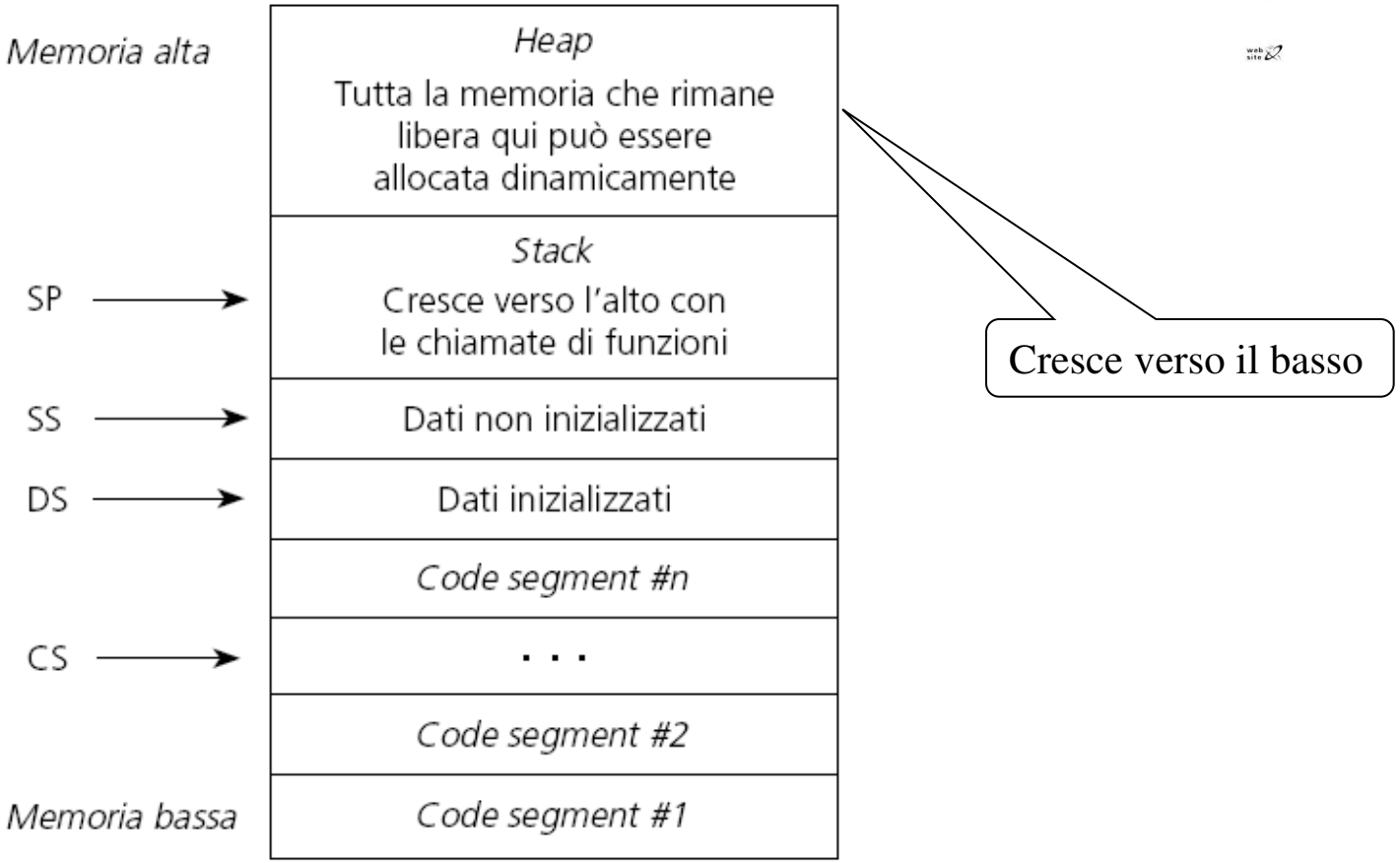
Allocazione Dinamica della Memoria

- Le variabili globali occupano posizioni fisse all'interno del segmento dati assegnato al programma (dal OS)
- Le variabili locali sono memorizzate nello stack e "vivono" solo mentre è in esecuzione la funzione che le usa.
- Entrambe devono essere definite in fase di compilazione
  - Non sempre è possibile sapere quanta memoria sarà necessaria al programma
- Tramite `new()` e `delete()` è possibile gestire la memoria in maniera dinamica
  - Le variabile dinamiche usano una zona della memoria chiamata heap.

mappa della memoria di un processo



operatore `new`

- genera dinamicamente una variabile di un certo tipo assegnandole un blocco di memoria della dimensione opportuna
- restituisce un puntatore che contiene l'indirizzo del blocco di memoria allocato, cioè della variabile, la quale sarà quindi accessibile dereferenziando il puntatore
- la sintassi è:

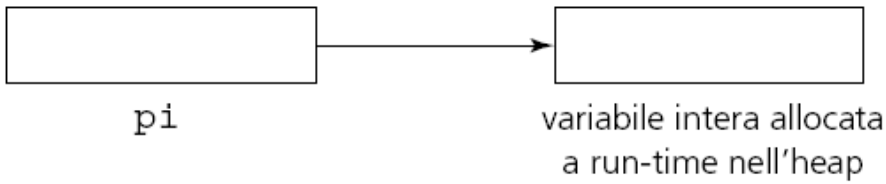
Il controllo sui tipi è fatto dal compilatore

```
tipo* puntatore = new tipo                  // non array
tipo* puntatore = new tipo[dimensione]      // array
```

```
char* p = new char[100]; // genera dinamicamente un vettore
                         //  di cento char
```

- Se il puntatore è già stato definito si può semplicemente usare `new` per assegnargli la variabile dinamica

```
int* pi;
pi = new int;
```



operatore `new` - Esempio


```
int main()
{
   char* str= "Corso di Programmazione";
   int lungh = strlen(str);
   char* puntatore= new char[lungh+1];
   strcpy(puntatore,str);
   cout << endl << "puntatore=" << puntatore;
   delete [] puntatore;
   return 0
}
```

Attenzione: l'heap non è infinito. In mancanza di spazio new restituisce NULL (non errore)

operatore `delete`

- libera la memoria allocata dinamicamente perché possa eventualmente essere riallocata mediante successive chiamate all'operatore `new`
- la sintassi dell'operatore `delete` è:

```
delete puntatore // non array
delete [] puntatore // per array
```

lo spazio assegnato per le variabili dinamiche:

```
int* ad = new int;
char* adc = new char[100];
```

si può liberare con le istruzioni:

```
delete ad;
delete [] adc;
```

- rende riutilizzabile la memoria puntata ma non cancella il puntatore che può quindi essere riutilizzato, ad esempio per puntare un'altra variabile successivamente allocata con `new`

new, delete vs malloc(), free

- Gli operatori new e delete sono molto più efficienti
- Allocano (e deallocano) memoria in funzione del tipo di dato da memorizzare
  - Effettuando ogni volta i relativi controlli
- Sono operatori (non funzioni)
  - Non necessitano di alcun header di file
- Non richiedono casting di tipo

esempi di `new` e `delete`


```
#include <iostream>
#include <string>
int main()
{
    char* p = new char[80];
    strcpy(p, ”Catania si trova in Sicilia")
    cout << p << endl;
    delete p;
    cout << "Prema Invio per continuare";
    cin.get();
}
```

In realtà new non è costretto a predeterminare la dimensione del vettore

esempi di `new` e `delete`


```
#include <iostream>
#include <cstring>      //contiene la funzione strlen
using namespace std;
int main()
{
   char* str = ”Io abito a Catania";
   int lung = strlen(str) // ottiene la lunghezza di str
   char* p = new char[lung+1];   // memoria a str + '\0'
   strcpy (p, str);              // copia str a p
   cout << "p = " << p << endl; // p sta ora in str
   delete [] p;
   return 0;
}
```


esempi di `new` e `delete`


```
#include <iostream>
using namespace std;
struct scheda_p {
   int numero;
   char nome[30];
};

int main ()
{
   struct scheda_p* una_scheda = new struct scheda_p;
   cout << "Introduca il numero del cliente: " ;
   cin >> una_scheda->numero;
   cout << "Introduca il nome:";
   cin >> una_scheda->nome;
   cout << "\nNumero: " << una_scheda -> numero;
   cout << "\nNome; " << una_scheda -> nome;
   delete una_scheda;
}
```


esempi di new e delete


```
#include <iostream>
using namespace std;
int main()
{
    int* Giorni = new int[3];
    Giorni[0] = 15;
    Giorni[1] = 8;
    Giorni[2] = 1999;
    cout  << "Le feste del paese sono in Agosto "
          << Giorni[0] << "/ "
          << Giorni[1] << "/ "
          << Giorni[2];
    delete [] Giorni;          // Libera memoria
    return 0;
}
```

L'output del programma è:

```
Le feste del paese sono in Agosto 15/8/1999
```


esempi di `new` e `delete`


```
#include <iostream>
#include <stdlib>
using namespace std;
int main()
{
   cout << "Quanti elementi ha il vettore?";
   int lun;
   cin >> lun;
   int* mioArray = new int[lun];
   for (int n = 0; n < lun; n++)
      mioArray[n] = n + 1;
   for (int n = 0; n < lun; n++)
      cout << '\n' << mioArray[n];
   delete [] mioArray;
   return 0;
}
```


gestione dell'overflow dello heap

- Per far fronte all'eventuale mancanza di spazio nell'heap si può utilizzare la funzione `set_new_handler()` che serve per gestire gli errori
- Definita nell'header file `new.h`, ed ha come argomento un puntatore a funzione
- Quando la si invoca le si passa il nome di una funzione scritta per gestire l'errore.

gestione dell'overflow dello heap


```
#include <iostream>
using namespace std;
void overflow()
{
   cout << "Memoria insufficiente - fermare esecuzione << endl;
   exit(1);
}
int main()
{
   set_new_handler (overflow);
   long dimensione;
   int* pi;
   int nblocco;
   cout << "Dimensione desiderata?";
   cin >> dimensione;
   for (nblocco = 1; ; nblocco++)
   {
      pi = new int[dimensione];
      cout << "Allocazione blocco numero: " << nblocco <<
      endl;
   }
}
```

Il programma produrrà un output del tipo:

```
Dimensione desiderata?
Allocazione blocco numero: 1
Allocazione blocco numero: 2
Allocazione blocco numero: 3
Allocazione blocco numero: 4
Allocazione blocco numero: 5
Memoria insufficiente - fermare esecuzione
```


tipi di memoria in C++

Memoria automatica

- variabili definite all'interno di funzioni si dicono *automatiche*.
- si allocano automaticamente nello *stack* quando viene invocata la funzione in cui sono definite e si cancellano quando essa termina;
- sono ovviamente locali alla funzione in cui sono definite,
  - sono locali al blocco che le contiene, cioè alla più piccola sezione di codice rinchiusa tra parentesi graffe in cui sono definite

Memoria statica

L'allocazione statica avviene nel *data segment*. Ci sono due modi per allocare staticamente una variabile:

1. definirla al di fuori da qualsiasi funzione, `main()` compresa
2. anteporre alla sua definizione la parola riservata `static`

Memoria dinamica

- Viene allocata a *run-time* nello *heap* dagli operatori `new` e `delete`.
- In assenza di un *garbage collector,* (es. Java e Visual Basic) la memoria rimane allocata fino a che viene affrancata dall'esecuzione di un'opportuna istruzione di `delete`.

