Dynamic Memory Management

- Control allocation and de-allocation of memory
  - Use `new` and `delete` to dynamically (*run-time*) allocate and de-allocate the required memory space (from *heap*)

☞ *Operator* `new` allocates a variable of the data type, and then returns the pointer to the allocated memory

```
int *p;
p = new int;
```

`p` is `NULL` if the allocation is not successful (e.g. out of memory)

☞ *Operator* `delete` frees the previously allocated memory space specified by the pointer (one element)

```
delete p;
```

☞ Don't forget to free the memory space that will not be used anymore by the function (to avoid memory leak)

Dynamic Allocation of Arrays

```
sizeof(a)=8
sizeof(b)=400
```

- Dynamic allocation of arrays
  - Operator `new[]` dynamically allocates an array with size that can be determined at *run* (*execution*) *time*

```
int N;
int *a;
```

```
int N, *a;
cin >> N;
a = new int[N];
for (int i=0;i<N;i++) a[i] = i*i;
```

Be careful to check the validity of `N` (e.g. positive integer) before giving it to the new operator

☞ *cf.* The size of a static array specified at compile time

```
const int N=100;
int b[N];
```

  - Operator `delete []` de-allocates an array of elements that is dynamically created

```
delete [] a;
```

De-allocate the entire array starting from `a`, not just the first element

Dynamic Allocation Example

```
#include <iostream>
using namespace std;

int main( )
{
    int i,num=0, *p=NULL;
    cout << "Please enter the number of elements: ";
    cin >> num;

    p = new int[num];
    for (i=0;i<num;i++)
    {
      cout << "Element " << i << ": ";
      cin >> p[i];
    }

    for (i=0;i<num;i++) cout <<"\nSquare of "<<p[i]<<" is " <<p[i]*p[i];
    delete [] p;
}
```

Proper error checking (for `num` and `p`) is recommended

By default an exception is thrown if `new` fails (e.g. insufficient memory)

Use `new (nothrow) int[…]` to avoid the exception (**but** check for yourself if pointer `p` is NULL)

Resizing the Array

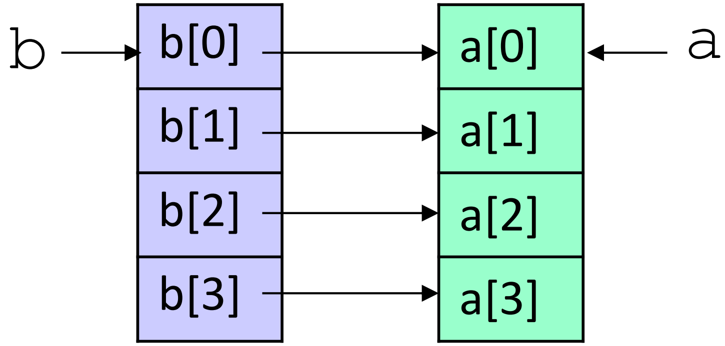
```
#include <iostream>
using namespace std;

int main( )
{
    int osize=10, nsize=20;
    int *p=NULL, *temp=NULL;

    p = new int[osize];            // the original block
    for (int i=0;i<osize;i++) {p[i] = i*i; cout << p[i] << " ";}

    temp = new int[nsize];         // new block for the expanded array
    for (int i=0;i<osize;i++) temp[i] = p[i];

    delete [] p;                   // deallocate the old block
    p = temp;                      // point to the new block

    for (int i=osize;i<nsize;i++) {p[i] = i*i; cout << p[i] << " ";}
    delete p;
}
```

The original array needs to be freed after the content has been copied to the new array to avoid memory leak

Array of Pointers

- Array of pointers
  - Elements of an array can be pointers

```
int a[4] = {5, 6, 7, 8};
int *b[4] = {a, a+1, a+2, a+3};
```

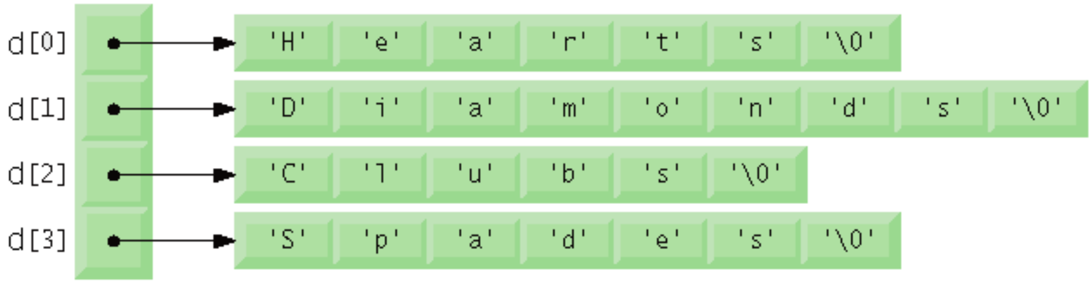


  - Array of pointers to characters

```
char *c = "Bridge";
char *d[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

Add const before char to avoid warnings for some compilers

☞ What if the first dimension is to be dynamically determined?



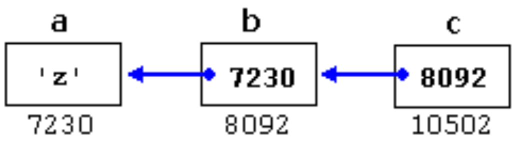
Pointing to Pointers

- "Pointer" pointer
  - A pointer to a memory address of a particular data type
  - A pointer can point to a *variable* that is itself a pointer
  - For *each level of dereference* needed, add an asterisk (`*`) to the declaration

```
char a = 'z';
char *b = &a;
char **c = &b;
```

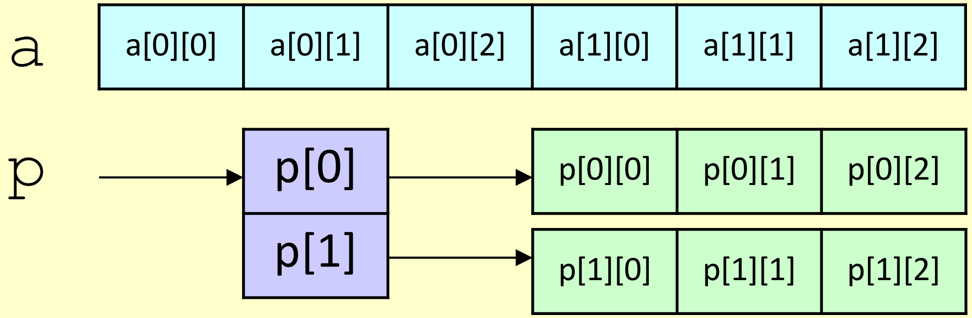
`c` points to the memory address of a variable (i.e. `b`) that is a pointer of type `char *`

  - Multiple levels of dereferencing
    - `c` has type `char**` and a value of 8092
    - `*c` has type `char*` and a value of 7230
    - `**c` has type `char` and a value of 'z'



Using Multi-Dimensional Pointers

```
#include <iostream>
using namespace std;

int main( )
{
    int i,j;
    int a[2][3]={{1,2,3}, {4,5,6}};
    int **p=NULL;

    p = new int*[2];
    for (i=0;i<2;i++) p[i] = new int[3];

    for (i=0;i<2;i++)
    for (j=0;j<3;j++) p[i][j] = a[i][j]*a[i][j];

    for (i=0;i<2;i++) delete [] p[i];

    delete [] p;
}
```



`p` is a pointer to integer pointers

`a` is of the type `int (*)[3]`

`p[i][j] == *(p[i]+j) == *(*(p+i)+j)`

Generic Pointer

- Generic pointer
  - Dereferencing a memory address makes sense only when the *data type of the variable* is known beforehand
  - A pointer typically needs to be declared to point to a particular data type (e.g. `int *`)
  - It is possible to declare a *generic* pointer and then convert it to point to data of different types at different times

```
void *p;
```

☞ The *void pointer* can be used to point to any data type ("store" the memory address for any type of data)

☞ Proper casting still needs to be performed to dereference the memory address stored in the void pointer

Using the Void Pointer

Recall that `cout` handles character pointer *differently* from other types of pointer

```
#include <iostream>
using namespace std;

int main()
{
    int i = 6;
    char c = 'a';
    void *p;

    p = &i;
    cout << "Value of i is " << *(int*)p << endl;

    p = &c;
    cout << "Value of c is " << *(char*)p << endl;
}
```

```
cout << p;
cout << &c;
```

```
cout << (void*)&c;
```

☞ A single pointer variable `p` of type `void *` can be used for handling data of *different types*

0x12345678

Little Endian

78	56	34	12

Big Endian

12	34	56	78

low address →

A Play with the Pointer

$-1313.3125 = 11000100101001000010101000000000_2$
$= C4A42A00_{16}$

```
#include <iostream>
#include <iomanip>
using namespace std;

int main( )
{
    unsigned char a[4] = {0x00, 0x2A, 0xA4, 0xC4};
    void *p = a;

    cout << "floating value=" << setprecision(8) << *(float*)p << endl;

    float x = -1313.3125;
    p = &x;

    cout << "byte sequence=";
    for (int i=3;i>=0;i--)
    cout << hex << uppercase << (int)( (unsigned char*)p )[i] << ' ';
}
```

Use of `const` with Pointers

- The `const` qualifier
  - `const` means that the value of a particular variable should not be modified
- Pointer and data
  - ☞ For a pointer, the pointer itself can be constant (not pointing to different data once initialized), or the data it points to can be constant (the data cannot be modified)
  - ☞ *Constantness* avoids *unintended change* of the value

① Non-constant pointer to non-constant data

② Non-constant pointer to constant data

③ Constant pointer to non-constant data

④ Constant pointer to constant data

Non-constant Pointer

A string pointer is a pointer to a constant array of characters

```
const char *string = "Hello";
```

① Non-constant pointer to non-constant data

- The most flexible case

```
int a = 0, b = 0;
int * p = &a;
*p = 3;                    // okay
p  = &b;                   // okay
```

② Non-constant pointer to constant data

- The data the pointer points to cannot be modified

```
int a = 0, b = 0;
const int * p = &a;
*p = 3;                    // wrong
p  = &b;                   // okay
```

```
int const * p == const int * p
```

```
const int a = 0, b = 0;
```

☞ It is okay to change the value stored in the memory location `p` points to through `a`

Constant Pointer

An array name can be considered as a constant pointer to the beginning address of the array

③ Constant pointer to non-constant data

- The pointer always points to the same location

```
int a = 0, b = 0;
int * const p = &a;
*p = 3;                  // okay
p  = &b;                 // wrong
```

Note that p needs to be initialized at declaration

④ Constant pointer to constant data

- Both the pointer and the data it points to cannot be modified

```
int a = 0, b = 0;
const int * const p = &a;
*p = 3;                  // wrong
p  = &b;                 // wrong
```

Review

- Array
  - 1-D and multi-dimensional arrays
  - Array size and index
  - Array initialization
  - Array as argument
- Pointer
  - Memory address and address operator
  - Pointer and array
  - Pointer as argument
  - Memory management
  - Pointer with const