Computer Programming

Array and Pointer

Hung-Yun Hsieh
September 27, 2022

Array of Data

Static arrays are blocks of static memory whose size must be determined at compile time, before the program runs

- Array
  - An array is a group of like-type data
    - Often used to maintain a group of variables of same data type

```
int a1, a2, a3, a4, a5, a6, a7, a8, a9, a10;
```

☞ An array is a group of elements of the same data type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier

  - Static array: a C++ data structure
    - Variable name with the number of elements and their common data type

```
aType aName[aNumber];
```

    - Example

```
int a[10];
```

Declaring the identifier a for use with an array of 10 integers

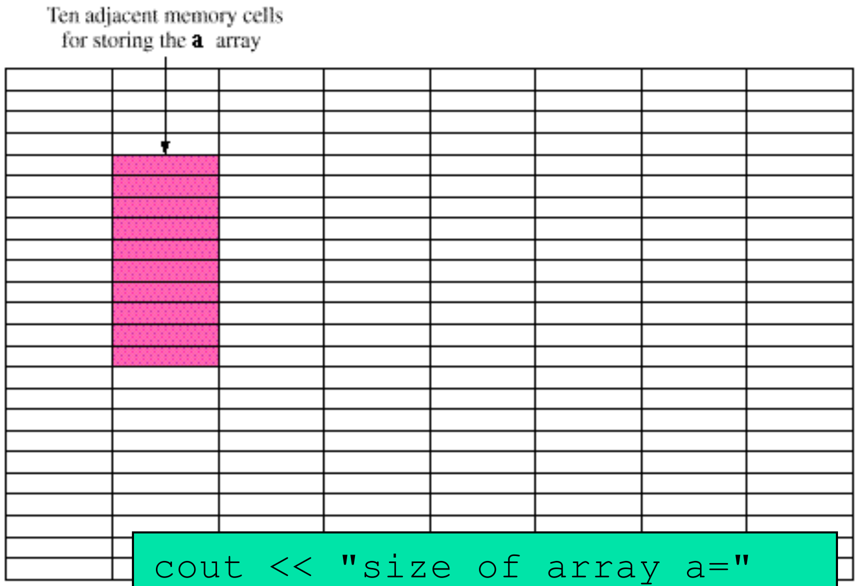
Array Index

Size of the array allocated in this way cannot be changed throughout the entire program execution (called static array)

- Memory allocation

```
int a[10];
```

Memory space that can store 10 integers is allocated (reserved) for use with array a



```
cout << "size of array a="
     << sizeof(a);
```

- Array index
  - Array index starts from 0
  - Use `a[i]` to access the (`i+1`)th element in array `a`

```
for (int i=0;i<=9;i++) a[i] = i;
```

☞ Be careful about using the array index; C++ *does not* do the index checking for you through []

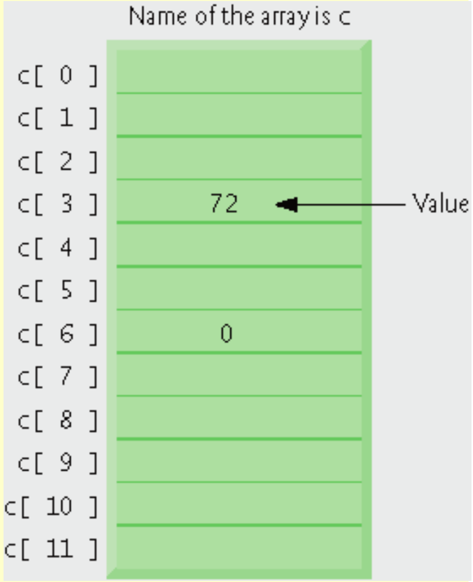
Using Arrays

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

int main( )
{
    double a[2];
    int c[12];

    a[0] = 11.1;
    a[1] = 22.5;

    c[3] = 72;
    c[6] = 0;

    cout<< "a[0]=" << a[0] << ", a[1]=" << a[1] << endl;
    cout<< "c[0]=" << c[0] << ", c[3]=" << c[3] << endl;
}
```

Values of `a[0]` and `a[1]` are un-initialized ("garbage" value)

Available elements are: `c[0]`, `c[1]`, `c[2]`, … `c[11]`

What is the output of the following statement? `cout << a;`



☞ Array is often handled on an *element-by-element* basis using [] – the array name itself has a special purpose

Initializing Arrays

```
double x[2]={0};
double x[2]={};
```

to initialize the values of elements in x to 0's in a handy way

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


int main( )
{
    double x[2] = {0, 0};
    int a[3] = {11, 22}, b[] = {44, 55, 66};

    cout<<"a[0]="<<a[0]<<", a[1]="<<a[1]<<", a[2]="<<a[2]<<endl;
    cout<<"b[0]="<<b[0]<<", b[1]="<<b[1]<<", b[2]="<<b[2]<<endl;

    cout<<"Please enter two real numbers: "<<endl;
    cin>>x[0]>>x[1];

    cout<<"x[0]="<< x[0] <<", x[1]="<< x[1] <<endl;

}
```

Any unassigned element is automatically set to 0 using {}

Compiler will determine the size of array b automatically using {} (i.e., 3)

Character Array

`cout` handles character array *differently* from other types of array, as it does for `char` and other integral types

- An array of characters

```
char x[3]={'c', 'a', 't'};
```

Note that `'\0'` (ASCII code = 0) is different from `'0'` (ASCII code = 48)

☞ An important functionality of a character array is to store and represent a "string" (specified via double quotes)

☞ If a character array

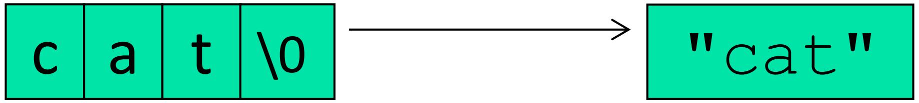
① stores each character of a string one by one in its elements

② stores a null character '\0' in the *last element* of the array

then the array name can be properly manipulated as a string by existing C string functions

```
char a[4]={'c', 'a', 't', '\0'};
```

To show the string:
`cout << a;`



`cout` prints until a null character is encountered

Character Array Example

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

int main( )
{
    char aa;
    char bb[4], cc[15];

    aa='g';
    bb[0]='c';
    bb[1]='a';
    bb[2]='t';
    bb[3]='\0';

    cout << "aa=" << aa << ", bb=" << bb << endl;
    cout << "Enter a string of less than 14 characters:\n";
    cin >> cc;
    cout << "String=" << cc << endl;
}
```

```
char bb[4]= {'c', 'a', 't', '\0'};
char bb[4]= {'c', 'a', 't'};
char bb[] = {'c', 'a', 't', '\0'};
char bb[] = "cat";
```

`cin >>` reads until the white-space character is encountered

`cout` prints until a null character is encountered

```
cin >> setw(sizeof(cc)) >> cc;
```

At most `sizeof(cc)-1` characters are read through `cin`

One for '\0'

```
cin.getline(cc, sizeof(cc));
```

Multi-Dimensional Array

- Consider a matrix or table

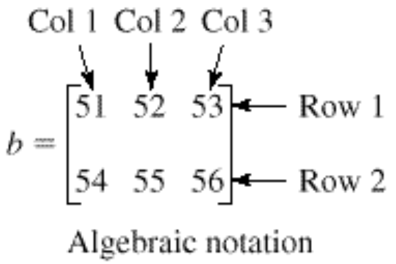
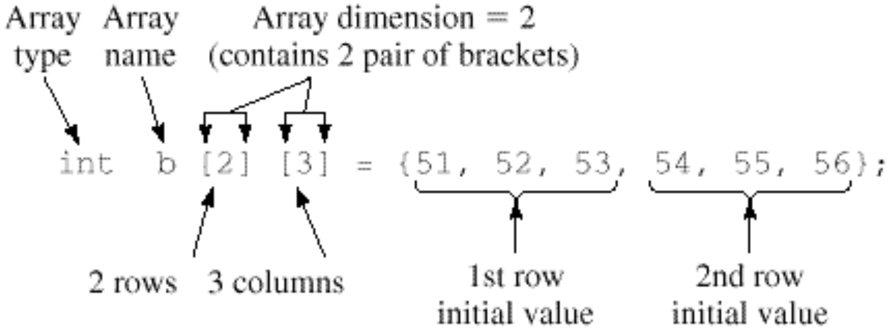


Table or matrix form



```
int  b [2] [3] = {51, 52, 53, 54, 55, 56};
```

C++ notation

- Declaring a multi-dimensional array
  - Name of array, dimension, number of elements for each dimension, and the common data type

```
aType aName[aNumber1][aNumber2][aNumber3];
```

☞Total number: aNumber1*aNumber2*aNumber3

Order of Elements

`a[i][j]` is located at position `i*3+j`

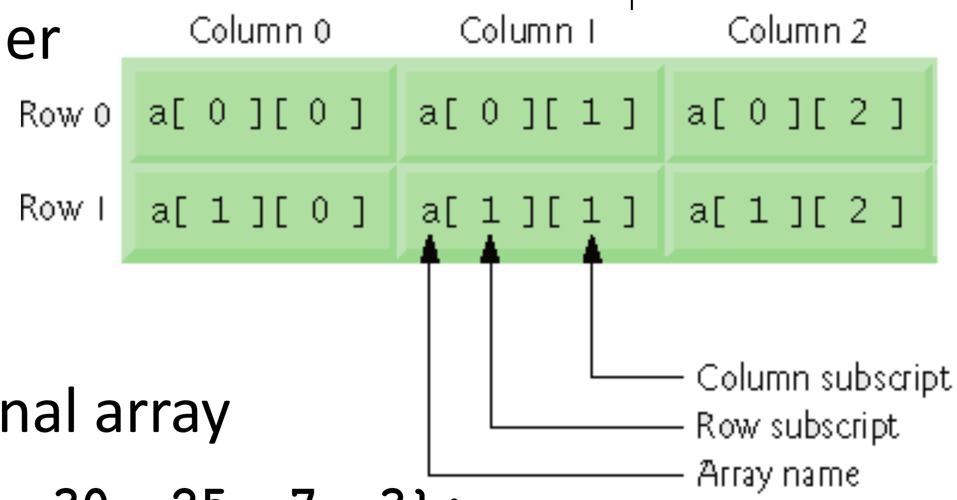
a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

actual layout

- Array indices
  - Index starts from 0 for each dimension
  - Stored in *row-major* order

```
int a[2][3];
```

    - 6 elements



- Array initialization
  - Similar to one-dimensional array

```
int a[2][3] = {50, 11, 30, 25, 7, 3};
```

☞ Multi-dimensional array is just an abstraction for programmers

- Array elements are still stored *linearly* in the memory

☞ The order progresses by incrementing the rightmost index

More on Array Initialization

- Plain listing

```
int a[4][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

- Hierarchical listing

```
int a[4][3] = {{1, 2, 3},
               {4, 5, 6},
               {7, 8, 9},
               {10, 11, 12}};

int a[][3] = {{1, 2, 3},
              {4, 5},
              {7, 8, 9},
              {10}};
```

Elements that are left out without specification are implicitly assigned to 0

Determining Array Dimension

```
#define WIDTH  5
#define HEIGHT 3

int main ()
{
  int jimmy [HEIGHT][WIDTH];
  int n,m;

  for (n=0;n<HEIGHT;n++)
    for (m=0;m<WIDTH;m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
  return 0;
}
```

```
#define WIDTH  5
#define HEIGHT 3

int main()
{
  int jimmy [HEIGHT * WIDTH];
  int n,m;

  for (n=0;n<HEIGHT;n++)
    for (m=0;m<WIDTH;m++)
    {
      jimmy[n*WIDTH+m]=(n+1)*(m+1);
    }
  return 0;
}
```

☞ The compiler knows the structure of a multi-dimensional array for easier access by programmers

Another Example

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

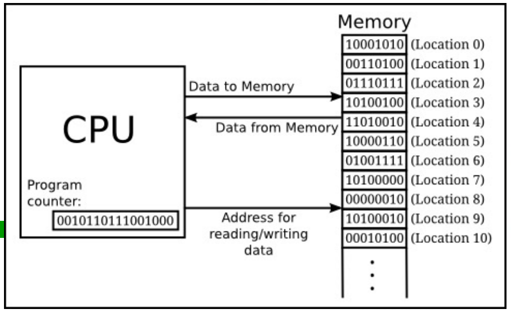
```
#define HEIGHT 20
#define WIDTH  60
#define RADIUS 5
int main()
{
    char point[HEIGHT][WIDTH];
    int  i, j;
    for (i=0; i<HEIGHT; i++)
    for (j=0; j<WIDTH ; j++) point[i][j] = ' ';

    for (i=0; i<HEIGHT; i++)
    for (j=0; j<WIDTH ; j++){
        double d = sqrt(pow(WIDTH/2-j,2)+pow(HEIGHT/2-i,2));
        if (abs(d-RADIUS)<0.3) point[i][j] = '*';}
    for (i=0; i<HEIGHT; i++)
    for (j=0; j<WIDTH ; j++) {
        cout << point[i][j];
        if (j==WIDTH-1) cout << '\n';}
}
```

$$\left|\sqrt{\left(j-\frac{width}{2}\right)^2+\left(i-\frac{height}{2}\right)^2}-radius\right|\le 0.3$$

Computer Programming

Pointer

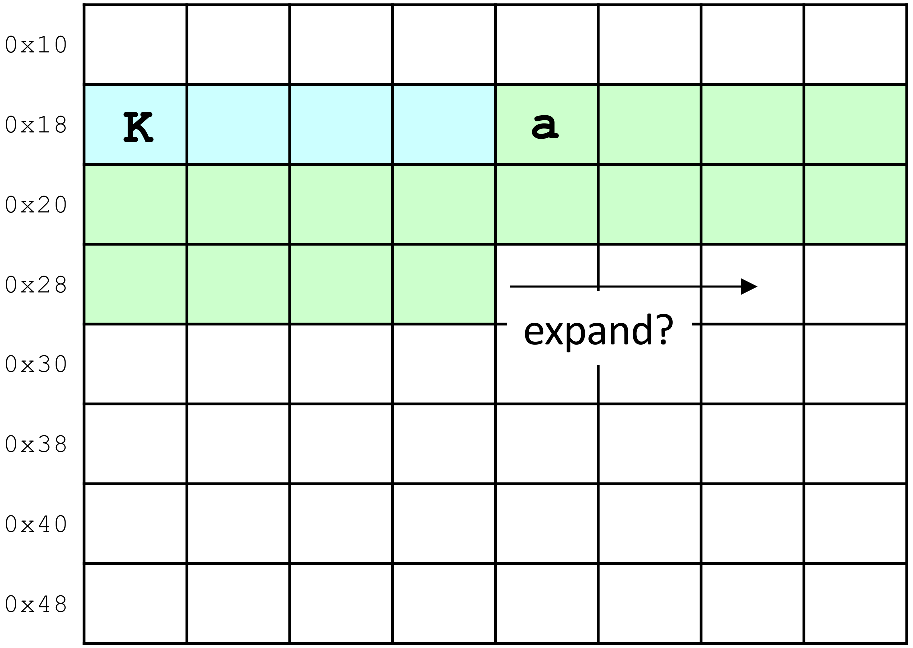
Variable and Memory



- Variable
  - A variable occupies a location in memory where value can be stored for use by the program

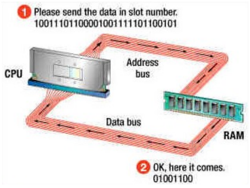
```
int K;

int a[4], b;
```

☞ The use of the variable name avoids the need to worry where the value is stored



- Sometimes we want to move the variable to a different location (e.g. for getting more space)

CPU Datapath



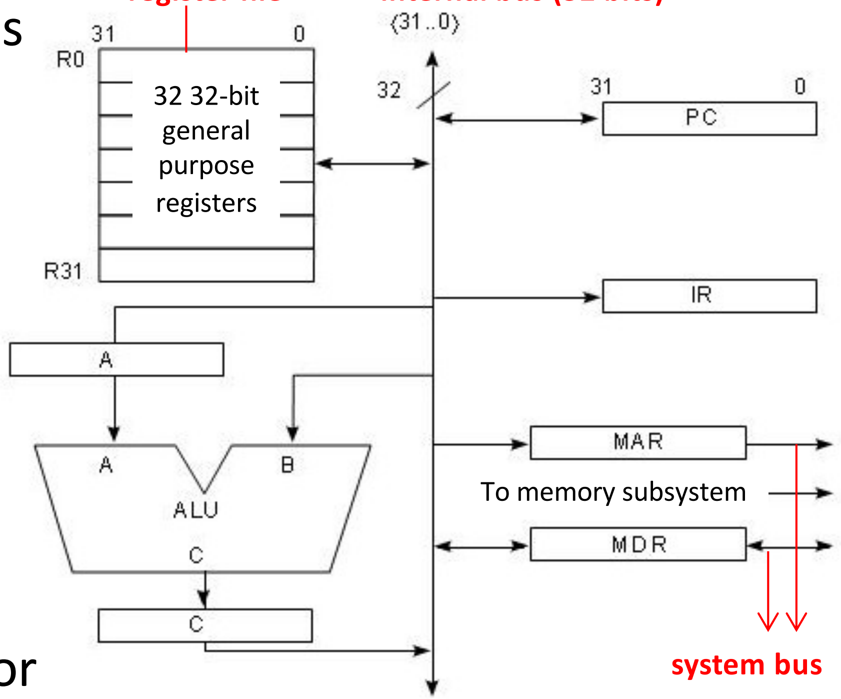
The bus can be considered as a group of parallel wires, where each wire can transfer 1 bit of data at a time

- Datapath
  - A datapath is a collection of functional units that perform data processing operations, including the ALU and several registers:
    - IR (instruction register) stores the current instruction to be executed
    - PC (program counter) stores the address of the next instruction to fetch
    - MAR (memory address register) stores the memory address of the target data
    - MDR (memory data register) stores the data to get from (or put into) the memory

arithmetic logic unit

register file

internal bus (32 bits)

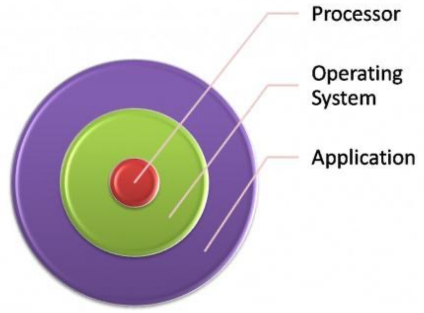


system bus

CPU "Size"

A word is a fixed-sized group of bits that are handled as a unit by the instruction set -- it is 2 bytes for a 16-bit CPU and 4 bytes for a 32-bit CPU

- 64-bit CPU (microprocessor architecture)
  - A CPU with direct support for 64-bit data and address
    - A CPU that treats 64-bit data as a unit (a *word* size of 64 bits)
    - 64-bit integer and address registers
    - ☞ *Integer* size, *datapath* width, and *memory address* width are all 64 bits
- ☞ *cf.* 64-bit software (operating system or application)
  - 64-bit memory address (addressing space)
  - ☞ It is possible to run a 32-bit OS on a 64-bit CPU (but not vice versa)
  - ☞ It is possible to run a 32-bit application on a 64-bit OS (but not vice versa)
  - ☞ Size of data (e.g. integer) could matter



Memory Address

- Memory address
  - Locations ("cells") in the memory are indexed by the memory address
    - Address 102, address 31503, …
    - ☞ Typically represented in hex format: 0x66, 0x7b0f, …
  - ☞ How to find out where a variable is stored (its address)?
- Address operator **&**
  - The address operator is a unary operator that returns the memory address of its operand

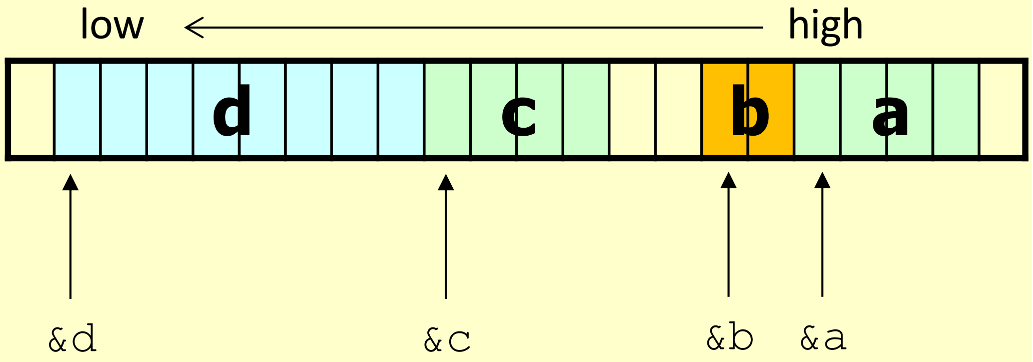
```
int y = 5;
cout << "Memory address of y = " << &y;
```

    - ☞ **&y** returns the *starting address (byte)* of the location that integer variable **y** is stored

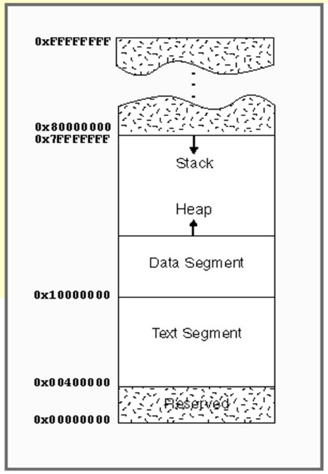
Addresses of Variables

When a CPU reads from or writes to the memory, it will do this in *word-sized* chunks (e.g. 4 bytes for 32-bit)

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

int main()
{
    int   a;
    short b;
    int   c;
    double d;

    cout << "Memory address of int    a = " << &a << endl;
    cout << "Memory address of short  b = " << &b << endl;
    cout << "Memory address of int    c = " << &c << endl;
    cout << "Memory address of double d = " << &d << endl;
 }
```



Data alignment

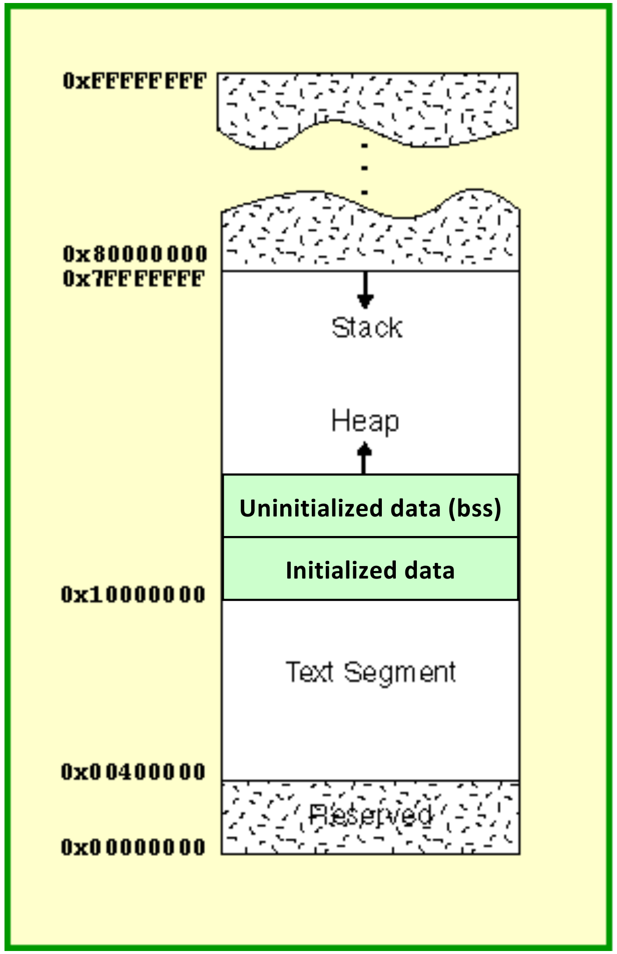


- ☞ Variables (auto) in the stack are occupied from high memory locations downwards
- ☞ Try global variables (declared *outside* `main()`)

Memory Layout of a Program

Check the error *"segmentation fault"*

- Partitioned into segments
  - A segment is a single memory block of variable size
- Text segment
  - Executable code & constants
- Data segment
  - Static and global variables
- Shared memory segment
  - Stack: auto variables & function parameters
  - Heap: dynamically allocated variables (more on this later)



Access for the Variable

Size of a pointer variable in a 64-bit program is 8 bytes

- Retrieving the value
  - `&a` returns the byte address of the starting location that variable `a` is stored
  - An address by itself is not sufficient for retrieving the value of the variable unless the type of variable is known
- Pointer
  - A special type of variable that contains the *memory address* as its value
  - A pointer contains the (starting) memory address of a variable allocated in the memory

☞ A pointer is declared according to the type of variable whose memory address it stores

Pointer

A pointer has two memory addresses associated with it: the memory it occupies and the memory that it points to

- Declaration of a pointer variable

```
pType *pName;
```

pType is the data type of the variable of which the memory address is contained in pName

  - Example

```
int *countPtr, count;
double *xPtr, *yPtr, zValue;
```

Pre-defined constant

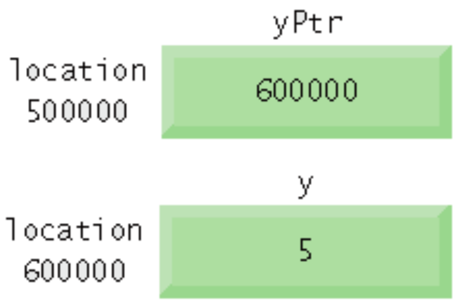
  - A pointer can be initialized to the value of 0 (NULL) or a proper address in memory

```
int y = 5;
```

```
int *yPtr;
yPtr = &y;
```

The variable yPtr now contains the memory address of y

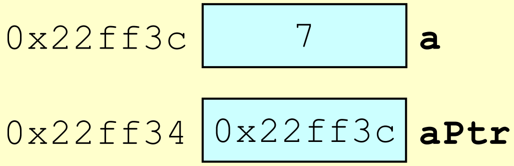
yPtr points to y



Using Pointers

cf. variable reference

```
int a;
int &aRef = a;
```



0x22ff3c | 7 | a

0x22ff34 | 0x22ff3c | aPtr

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

int main( )
{
    int a;
    int *aPtr = NULL;

    a = 7;
    aPtr = &a;

    cout << "The address of a is "  << &a   << "\n"
         << "The value of aPtr is " << aPtr << endl;
    cout << "The value of a is " << a << "\n"
         << "The value of *aPtr is "  << *aPtr << endl;
    cout << "The address of aPtr is " << &aPtr << endl;
    cout << "The value of *&aPtr is " << *&aPtr << endl;
    cout << "The value of &*aPtr is " << &*aPtr << endl;
}
```

Indirection (dereferencing) operator * returns synonym (alias) for the variable its operand points to

```
*aPtr = 9;
cout << "The value of a is "
<< a << endl;
```

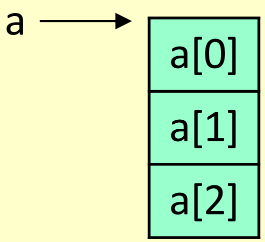
* and & are inverses of each other

Pointer and Array

A value context is when you are retrieving and using the value of an object, such as the right hand side of `x = y;`

An object context is when you are modifying or querying the object itself, such as the left hand side of `x = y;`

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

int main( )
{
    int a[3] = {10, 20, 30};
    int *aPtr, *bPtr;

    aPtr = a;
    bPtr = &a[0];

    cout << "The address of array a is " << a << "\n"
         << "The address of element a[0] is " << &a[0] << endl;
    cout << "The value of aPtr is " << aPtr << "\n"
         << "The value of bPtr is " << bPtr << endl;
    cout << "The value of aPtr[1] is " << aPtr[1] << endl;
    cout << "The value of bPtr[2] is " << bPtr[2] << endl;
}
```



An array name is converted to a pointer to the first element when used in a value context (as opposed to an object context)

Object context: `sizeof(a)` vs. `sizeof(aPtr)`

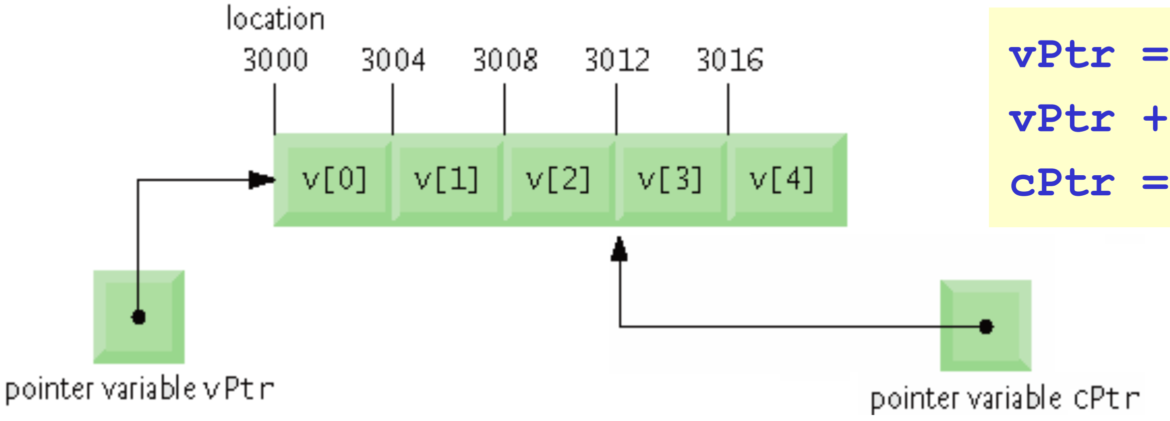
Pointer Arithmetic

```
int a[10];
a[i]  == *(a+i)
&a[i] == a+i
```

- Pointer arithmetic
  - ☞ Useful when operated on a pointer to an array
  - Increment or decrement pointer (++ or --)
  - Add/subtract an integer to/from a pointer
  - Pointers may be subtracted from each other

```
int *vPtr, *cPtr;
int v[5];
```

```
vPtr = &v[0];
vPtr ++;
cPtr = vPtr + 2;
```



Precedence Revisited

Note the precedence in
`&a[i]` (`a`: array)
`++*p,` `*++p,` `*p++` (`p`: pointer)

Operators	Associativity	Type
`()` `[]`	left to right	parentheses
`++` `--` `static_cast<`*type*`>()`	left to right	unary (postfix)
`++` `--` `-` `~` `!` `&` `*`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Character Pointer

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

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

int main( )
{
    char s1[10] = {'H','E','L','L','O'};
    char *s2 = "Hello";

    for (int i=0; (s1[i]=s2[i])!='\0'; i++);

    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;

    s1[0] = 'h';
    cout << "s1 = " << s1 << endl;
    // s2[0] = 'h';        // wrong
}
```

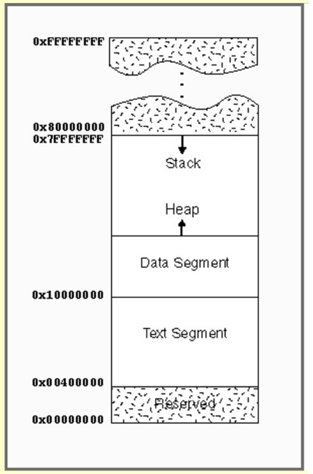
`sizeof(s1)=10`
`sizeof(s2)=8`

String literals have static storage and the space is read only

Initialize the array with characters in `"HELLO"` (`char s1[10]="HELLO";`)

Point to the starting address of the string literal `"Hello"`

Add `const` before `char` to avoid warnings for some compilers



☞ String literals are allocated in the (initialized) DATA or TEXT segment that is read-only (no modification)