EEE146

POINTERS AND REFERENCES

Variables and memory addresses

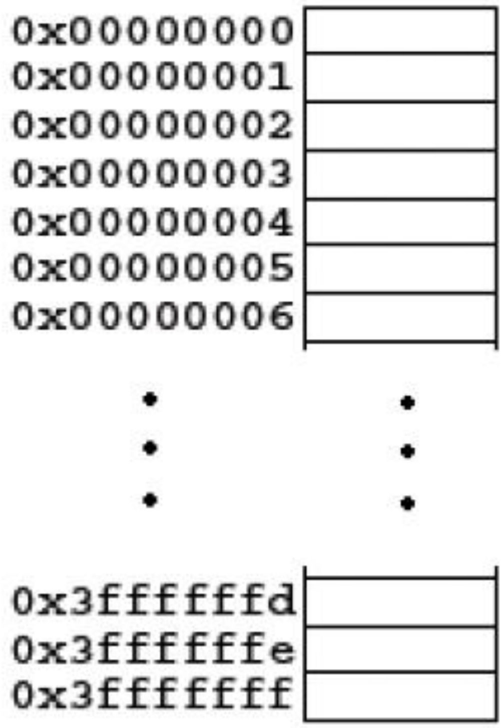
Computer memory can be considered as a very large array of bytes.

For example a computer with 1GB of RAM actually contains an array of

1024 × 1024 × 1024= 1,073,741,824 Bytes

0=0x00000000

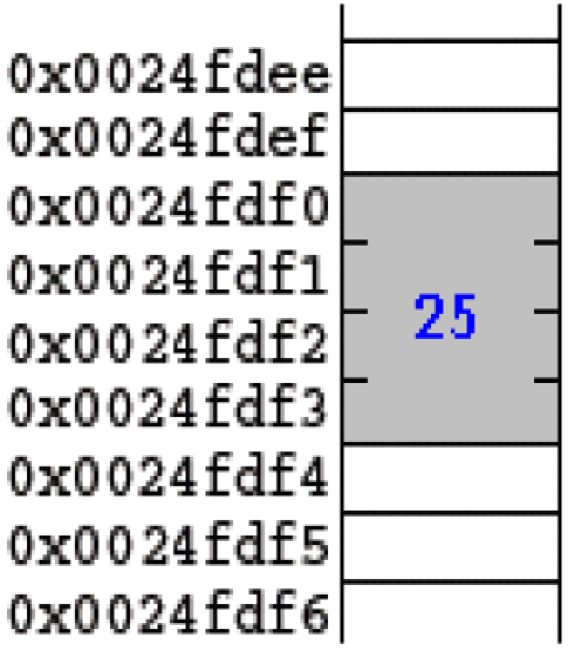
1,073,741,824=0x3fffffff



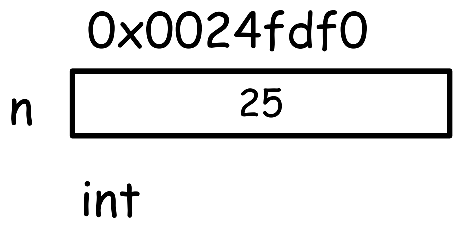
Variables and memory addresses

When a variable is declared and assigned to a value four fundamental attributes associated with it:

- its name
- its type
- its value (content)
- its address



e.g. int n=25;



Variables and memory addresses

In C/C++ the address operator & returns the memory address of a variable.

```
#include <iostream>
using namespace std;
int main(){
    int n=32;
    cout<< "n= " << n << endl;
    cout<< "&n= " << &n << endl;
    return 0;
}
```

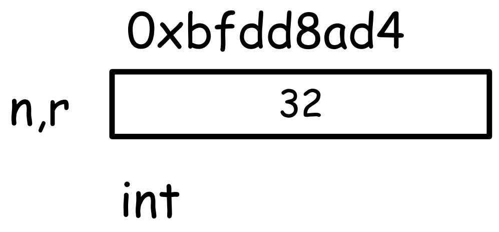
```
Output:

n= 32
&n= 0x0024fdf0
```

References

The reference is an alias, a synonym for a variable.

It is decelerated by using the reference operator &.

```
#include <iostream>
using namespace std;
int main(){
 int n=32;
 int &r=n; //r is a reference for n
 cout << n << " " << r <<endl;
 --n;
 cout << n << " " << r <<endl;
 r*=2;
 cout << n << " " << r <<endl;
 cout << &n << " " << &r <<endl;
 return 0;
}
```



```
Output:
32 32
31 31
62 62
0xbfdd8ad4     0xbfdd8ad4
```

C++ References vs Pointers

- References are often confused with pointers but three major differences between references and pointers are:
- You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

References

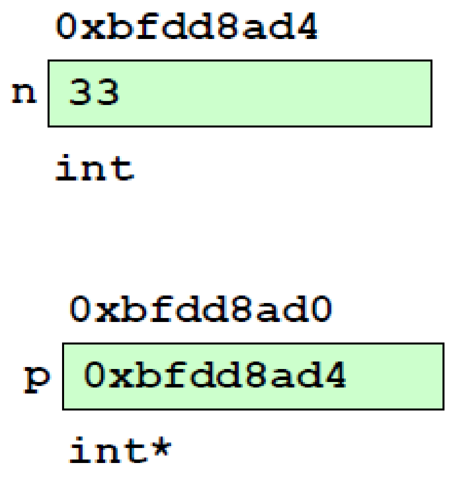
```cpp
 #include <iostream>
using namespace std;
void swap(double &x,double &y){
  double z;
  z=x; x=y; y=z;
}
int main(){
 double a =11.1, b=22.2;
 cout<<a <<" " << b << endl;
 swap(a,b);
 cout<<a <<" " << b << endl;
 return 0;
}
```

```
Output:
11.1   22.2
22.2   11.1
```

Pointers

- The address operator returns the memory adress of a variable.
- We can store the address in another variable, called ***pointer***.

```cpp
#include <iostream>
using namespace std;
int main()
{
  int n = 33;
  int* p = &n; // p holds the address of n
  cout << " n = " <<  n << endl;
  cout << "&n = " << &n << endl;
  cout << " p = " <<  p << endl;
  cout << "&p = " << &p << endl;
  cout << "*p = " << *p << endl;
}
```



```
 n = 33
&n = 0xbfdd8ad4
 p = 0xbfdd8ad4
&p = 0xbffafad0
*p = 33
```

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

void takas(double *x, double *y){
  double z;
  z  = *x;
  *x = *y;
  *y = z;
}

int main(){
    double a = 11.1,  b = 22.2;

    cout << "a b : " << a << " " << b << endl;

    takas(&a, &b);

    cout << "a b : " << a << " " << b << endl;
}
```

```
a b: 11.1  22.2
a b: 22.2  11.1
```

Pointers and Arrays

- The name of an array is the address of its first element.
- The array name is a constant pointer.

```
    float numbers[20];
    float *ptr = &numbers[0];   // valid
```

The following assignments are equivalent:

```
    numbers[4] = 25.8;
    *(ptr+4) = 25.8;
```

Dynamic Memory Management

The declaration:

```
double mass[10];
```

Array size define at ***compile-time***

Alternatively we can use a *named constant*;

```
const int n = 10;
double mass[n];
```

Array size define at ***compile-time***

Note that "*Standard* C++" Array size defined at ***run-time FORBIDDEN!***

~~`int n;`~~
~~`cin >> n;`~~
~~`double mass[n];`~~

or

~~`int n = 10;`~~
~~`double mass[n];`~~

* * * This type of arrays are called **Static Arrays** * * *

- C++ provides run-time or **dynamic arrays** for which memory is allocated during execution.
- To allocate memory dynamically at run-time we use `new` operator.

General form:

```
pointer = new type;  // for single element

pointer = new type [number_of_elements];
```

For example, to request a 10 element block of type `int` dynamically, we can use

```
      int * mass;
      mass = new int [10];
```

or

```
      int * mass = new int [10];
```

The delete operator reverses the action of the new operator, that is it frees the memory allocated by the new operator.

Its form is:

```
delete pointer;      // single element
delete [] pointer;   // a block of elements
```

e.g.

```
delete [] mass;
```

```
int main (){
     double *x, mean, s;
     int i, n;

     while(true){
          cout << "How many elements: ";  cin >> n;
          if(n<=0) break;

          x = new double[n];
          s = 0.0;
          cout << "Input elements: ";
          for(i = 0; i<n; i++){
               cin >> x[i];
               s += x[i];
          }

          mean = s/n;
          cout << "Mean = " << mean << endl;
          delete [] x;
     }
} // main
```

Sample output of the previous program:

```
How many elements: 3

Input elements: 1 2 3

Mean = 2.0

How many elements: 6

Input elements: 2 4 5 9 1 0

Mean = 3.5

How many elements: 0
```

Null pointers

It is always a good practice to assign the pointer NULL to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries, including iostream.

Null pointers

Consider the following program:

```
#include <iostream>
using namespace std;
int main ()
{ int *ptr = NULL;
  cout << "The value of ptr is " << ptr ;
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
The value of ptr is 0
```

Null pointers

To check for a null pointer you can use an if statement as follows:

```
if(ptr)        // succeeds if p is not null

if(!ptr)       // succeeds if p is null
```

Pointer arithmetic

As you understood pointer is an address which is a numeric value; therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers:

++, --, +, and -

Incrementing a Pointer

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main (){
 int var[MAX] = {10, 100, 200};
 int *ptr;
 ptr = var;
 for (int i = 0; i < MAX; i++){
 cout << "Address of var[" << i << "] = ";
 cout << ptr << endl;
 cout << "Value of var[" << i << "] = ";
 cout << *ptr << endl;
 ptr++;
 }
 return 0;
}
```

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

Decrementing a Pointer

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
  int var[MAX] = {10, 100, 200};
  int *ptr;
  ptr = &var[MAX-1];
  for (int i = MAX; i > 0; i--)
  {
   cout << "Address of var[" << i << "] = ";
   cout << ptr << endl;
   cout << "Value of var[" << i << "] = ";
   cout << *ptr << endl;
   ptr--;
  }
  return 0;
}
```

```
Address of var[3] = 0xbfdb70f8
Value of var[3] = 200
Address of var[2] = 0xbfdb70f4
Value of var[2] = 100
Address of var[1] = 0xbfdb70f0
Value of var[1] = 10
```

Pointer comparisons

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main (){
  int var[MAX] = {10, 100, 200};
  int *ptr;
  ptr = var;
  int i = 0;
  while ( ptr <= &var[MAX - 1] ){
   cout << "Address of var[" << i << "] = ";
   cout << ptr << endl;
   cout << "Value of var[" << i << "] = ";
   cout << *ptr << endl;
   ptr++;
   i++;}
   return 0;
}
```

```
Address of var[0] = 0xbfce42d0
Value of var[0] = 10
Address of var[1] = 0xbfce42d4
Value of var[1] = 100
Address of var[2] = 0xbfce42d8
Value of var[2] = 200
```

Pointers vs. arrays

Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing .

Consider the following program:

Pointers vs. arrays

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
  int var[MAX] = {10, 100, 200};
  int *ptr;
  ptr = var;
  for (int i = 0; i < MAX; i++){
   cout << "Address of var[" << i << "] = ";
   cout << ptr << endl;
   cout << "Value of var[" << i << "] = ";
   cout << *ptr << endl;
   ptr++;
  }
  return 0;
}
```

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

Pointers vs. arrays

However, pointers and arrays are not completely interchangeable. For example, consider the following program:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main (){
 int var[MAX] = {10, 100, 200};
 for (int i = 0; i < MAX; i++){
   *var = i; // This is a correct syntax
   var++;    // This is incorrect.
 }
 return 0;
}
```

It is perfectly acceptable to apply the pointer operator * to var but it is illegal to modify var value.

Pointers vs. arrays

Because an array name generates a pointer constant, it can still be used in pointer-style expressions, as long as it is not modified. For example, the following is a valid statement that assigns var[2] the value 500:

```
*(var + 2) = 500;
```

Above statement is valid and will compile successfully because var is not changed.

Passing pointers to functions in C++

C++ allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

Passing pointers to functions in C++

```cpp
#include <iostream>
#include <ctime>
using namespace std;
void getSeconds(unsigned long *par);
int main (){
 unsigned long sec;
 getSeconds( &sec );
 cout << "Number of seconds :" << sec << endl;
 return 0;
}
void getSeconds(unsigned long *par)
{
 // get the current number of seconds
 *par = time( NULL );
 return;
}
```

```
Number of seconds :1294450468
```

Passing pointers to functions in C++

```
#include <iostream>
using namespace std;
double getAverage(int *arr, int size);
int main (){
  int balance[5] = {1000, 2, 3, 17, 50};
  double avg;
  avg = getAverage( balance, 5 ) ;
  cout << "Average value is: " << avg << endl;
  return 0;
}
double getAverage(int *arr, int size){
  int i, sum = 0;
  double avg;
  for (i = 0; i < size; ++i){
  sum += arr[i];
  }
  avg = double(sum) / size;
  return avg;
}
```

```
Average value is: 214.4
```