Lecture on pointers, references, and arrays and vectors

pointers

for example, check out: http://www.programiz.com/cpp-programming/pointers [the following text is an excerpt of this website]

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

int main()
{
    int var1 = 3;
    int var2 = 24;
    int var3 = 17;
    cout << &var1 << endl;
    cout << &var2 << endl;
    cout << &var3 << endl;
}
```

Output

```
0x7fff5fbff8ac
0x7fff5fbff8a8
0x7fff5fbff8a4
```

Example if Pointer values and pointer dereferencing:

```
#include <iostream>
using namespace std;
int main() {
    int *pc
    int c;

    c = 5;
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;

    pc = &c;    // Pointer pc holds the memory address of variable c
    cout << "Address that pointer pc holds (pc): "<< pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

    c = 11;    // The content inside memory address &c is changed from 5 to 11.
    cout << "Address pointer pc holds (pc): " << pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

    *pc = 2;
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;

    return 0;
}
```

Output

```
Address of c (&c): 0x7fff5fbff80c
Value of c (c): 5

Address that pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 5

Address pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 11

Address of c (&c): 0x7fff5fbff80c
Value of c (c): 2
```

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.

We declare a variable

```
int    i = 17;
```

We can now declare reference variables for i as follows.

```
int&    r = i;
```

Read the & in these declarations as reference. Thus, read the first declaration as “r is an integer reference initialized to i” and read the second declaration as “s is a double reference initialized to d.”. Following example makes use of references on int and double:

```
#include <iostream>

using namespace std;

int main () {
   // declare simple variables
   int    i;
   double d;

   // declare reference variables
   int&    r = i;
   double& s = d;

   i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r  << endl;

   d = 11.7;
   cout << "Value of d : " << d << endl;
   cout << "Value of d reference : " << s  << endl;

   return 0;
}
```

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

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

References are usually used for function argument lists and function return values. So following are two important subjects related to C++ references which should be clear to a C++ programmer:

Concept	Description
References as parameters	C++ supports passing references as function parameter more safely than parameters.
Reference as return value	You can return reference from a C++ function like a any other data type can be returned.

References and Pointers comparison:

```
#include <iostream>
// using references
void swapRef(int &a, int &b) {
    int temp = 0;
    temp = a;
    a = b;
    b = temp;
    return;
}
// using Pointers
void swapPoint(int *a, int *b){
    int temp = 0;
    temp = *a;
    *a = *b;
    *b = temp;
    return;
}
// example of using pointers or references
// result is the same
int main(void){
   int a = 10;
   int b = 20;
   std::cout<< "a = "<< a <<"\n"<<"b = "<<b<<"\n";
   swapRef(a,b);
   std::cout<< "After swapRef()";
   std::cout<< "a = "<< a <<"\n"<<"b = "<<b<<"\n";
   a = 10; b = 20;
   swapPoint(&a,&b);
   std::cout<< "After swapPoint()";
   std::cout<< "a = "<< a <<"\n"<<"b = "<<b<<"\n";
   return 0;
}
```

To const or not to const

[see http://duramecho.com/ComputerInformation/WhyHowCppConst.html]

There are many uses and usages of **const** [it means stay constant do not change]

An obvious example:

```
const int a= 5;
cout << a << endl;
a = 6; // <== this will not compile! ERROR! You shalt not change a
```

You can use **const** for pointers, but there are a few difficulties:

```
const int * p;
int const * q;
int * const r;
int const * const s;
```

Confusing, right? Here the explanation.

```
const int * p;
```

Declares that *p* is a variable pointer to a **const** integer. For example

```
const int a=5;
const int *p;
p = NULL; // p points to nothing
p = &a; // p points to a
```

in fact on a mac this seems to be equivalent to **int * p** and is also equivalent to **int const * q** and also **int * const r** sugggests to have a constant pointer to a variable integer, ti seems that on mac all these lead to the same answers and also compile. The only difference is

```
int const * const s;
```

it is a const pointer to a const value; this only compiles using something like this:

```
const int a=5;
int const *  const p = &a;
```

You can protect return values of function against change using

```
const char *function() { return "Some text"; }
```

You can use **const** in arguments, here it becomes difficult: For example:

```
void addOne(int a) { a =  a + 1;}
int a = 5;
addOne(a); // a does not change
```

makes a copy of *a* and then adds one, but the results gets lost. We can use a reference:

```
void addOne(int &a) { a =  a + 1;}
int a = 5;
addOne(a)
cout << a << endl; //prints 6
```

Now *a* gets changed, if we do not want that *a* gets changed then we can use **const**

```
void addOne(const int &a) { a =  a + 1;} /COMPILER ERROR
```

This forces us to make sure that we do not manipulate objects that should be left alone.

Within classes we can add **const** after method definition to make sure that the method cannot change the class object it self.

```
class Class2 {
    void Method1() const;
    int Membervariable1;
}
```

For example we have encountered this in the **operator** methods: Look at our definition of **operator+**, correctly we would want to make sure that an operation **a+b** does not change or **b**

```
const Rectangle Rectangle::operator+(const Rectangle &other) const {
  Rectangle newr = *this;
  newr.side = side + rhs.side;
  newr.height = height + rhs.height;
  return newr;
}
```

Arrays

we know about

```
double a;
a = 5;
```

if we want to get a list of values and use them more like we know from math then we can define an array

```
double a[10];
for (int i=0;i<10;i++)
{
    a[i] = 5+i;
}
```

or easier:

```
int a[3] = {4,5,6};
```

or even easier:

```
int a[] = {4,5,6,7};
```

We can also define 2D-arrays:

```
int b[2][3] = {{1,2,3},{4,5,6}};
```

But what about dynamic arrays, the ones above are fixed length.

Allocation of memory on the fly

Use the keywords **new** and **delete**.

```
int * myArray = new int[Size] //gives you a contigous memory block
```

once your done with it you MUST delete it:

```
delete [] myArray;
```

if you use **new** to allocate for a single memory cell

```
int * a = new double;
//then you need this to delete:
delete a;
```

if you use **new** to allocate multiple memory cells

```
int * a = new double[SIZE];
//then you need this to delete:
delete [] a;
```

Now this is all very combersome, and you should probably not use your own arrays or Array classes.

Standard Template Library (STL)

Vector class is an example of the STL

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

int main()
{
   // create a vector to store int
   vector<int> vec;
   int i;

   // display the original size of vec
   cout << "vector size = " << vec.size() << endl;

   // push 5 values into the vector
   for(i = 0; i < 5; i++){
      vec.push_back(i);
   }

   // display extended size of vec
   cout << "extended vector size = " << vec.size() << endl;

   // access 5 values from the vector
   for(i = 0; i < 5; i++){
      cout << "value of vec [" << i << "] = " << vec[i] << endl;
   }

   // use iterator to access the values
   vector<int>::iterator v = vec.begin();
   while( v != vec.end()) {
      cout << "value of v = " << *v << endl;
      v++;
   }

   return 0;
}
```