C++ Programmer's Guide to Pointers (What You Need to Know!)

Introduction

This guide aims at providing a practical summary of c++ pointers and pointer use. The first thing a student needs to know about the concept of pointers is the fact that it is an extremely important and useful tool. Many students have an initial idea that pointers are difficult. They are not if you really understand them. Combine information found in this summary with the class notes and the textbook for more comprehensive understanding of pointers. Finally, remember that practice makes perfect. The more you solve, the easier pointers will be.

Pointer vs. Variables

1. Pointer Declaration

Knowing the difference between pointers and variables is the first step toward understanding pointers. You have used many variables in your programming so far:

```
int x ;
double y;
bool z;
char c;
string d
```

Pointers are themselves variables that have special characteristics. For instance, they have their own special way of declaring them.
The general rule of declaring any pointer is as follows:

```
type * pointer_name ;
```

Compare it to the general rule of declaring a normal variable:

```
type  variable_name ;
```

'type' can be any valid data type. Including:
a. Primitive Data Types (int, double, float, char, bool, ..)
b. User-defined Data Types (Structs or Classes) (Circle, Rect, Record, Student ..)
c. Type can also be a pointer. In other words, pointers are data types too.

Here are some examples:

Normal Variables	Pointer Variables

int x;
double y;
Circle mycircle; // Circle is struct
Car mycar; // Car is a class
float z
	*int * xptr;*
	*double * yptr;*
	*Circle * mycircleptr;*
	*Car * mycarptr;*
	*float * zptr // pointer to float*
	*float ** zptrptr; // pointer to float pointer*

2. Pointer Initialization

Initialization means assigning an appropriate value to a defined variable. Let's compare between how we initialized normal variables and how we initialize pointers. The general rule of initializing any pointer is as follows:

```
variable_name = valid_address;
```

Compare it to the general rule of declaring a normal variable:

```
variable_name = valid_value;
```

By valid address we mean either zero (0), null (NULL) or the address of variable that meets the pointer's expectation. To elaborate, suppose xptr is a pointer to float. Then only an address to a float variable can match the expectation of xptr. Here are some examples

```
float x = 5;
float * xptr;
xptr = &x;

float ** xptrptr ;    // xptrptr is a pointer to "float pointer"
xptrptr = &xptr ;   // notice that &xptr will give the address of the
                    //float pointer xptr. Therefore, it is a valid initialization.
```

The & is an operator. Remember the ! operator (negation operator) that we used to negate the value of a Boolean expression or variable. The & (address) operator tells the c++ compiler to look up the address of what comes after the operator and return it. So &x will give the address of x and &y will give me the address of y and so on.

3. Pointer Dereferencing

To dereference a pointer is to access (read or write) the content of the address that the pointer is pointing to.

The general rule of dereferencing any pointer is as follows:

```
type variable = value ;
type * pointer = & variable;
*pointer = value ; // write
cout<<*pointer ; // read
```

Compare it to the general rule of referencing a normal variable:

```
type variable = value; // write
cout<<variable; // read
```

To access the content of the address pointed to by the pointer we simply use the * operator followed by the pointer name. We do that for read or write. Here are some examples.

```
float x = 5;
float * xptr;
xptr = &x;
cout<<x<<endl;
*xptr = 10;
cout<<x<<endl;
cout<<*xptr<<endl;
```

4. Pointers & Arrays (Pointer Arithmetic)

Arrays are nothing but a set of variables of a certain type that hold consecutive memory locations. The name of the array is a pointer to the first element of the array. If I know where to find the first element of the array and I know the size of that array, I can pretty much reach any point in the array easily.

The general rule of creating an array is as follows:

```
type arrayname [size];
```

Because arrays are nothing but pointers I can give the array another name as follows:

```
type arrayname[size];
type * ptr ;
ptr = arrayname;
```

To access the content of the array we use the following notation

```
for ( int I = 0 ; I < size ; I++)
{
  cout<<arrayname[I]<<endl;
}
```

To access the array using an equivalent pointer we use the following

```
for ( int I = 0 ; I < size ; I++)
{
   cout<<*(ptr+I)<<endl;
}
```

Remember that ptr is the address of the first element. C++ allows us to use ptr+1 to mean the address of the second element of the array and so on. The following notations are equivalent:

Normal Array Access	Pointer Array Access
arrayname[I]	*(ptr+I)
arrayname[0]	*(ptr+0)
arrayname[2]	*(ptr+2)
arrayname[x-4]	*(ptr+x-4)

Adding or subtracting from pointers helps moving forward or backward inside arrays. This is referred to pointer arithmetic.

5. Pointers & Functions

Since we know by now that pointers and arrays are closely related, we can pass arrays to functions using pointers.

The following is a declaration of a function that receives an array.

```
return_type  function_name ( type  array [], int size);
```

The following is a declaration of a function that receives pointer:

```
return_type  function_name ( type * ptr,  int size);
```

To pass the array to the function we do the following

```
return_type  function_name ( type  array [], int size);
 int main ( )
{  type arrayname[size];
   function_name(arrayname,size);
}
```

To pass the array using pointers we do the following:

```
return_type  function_name ( type *  ptr, int size);
 int main ( )
{  type arrayname[size];
   type * ptr = arrayname;
   type *anotherptr = &arrayname[0];
   function_name(ptr,size);
   function_name(anotherptr,size);
}
```

6. Pointers & User-defined Types

User defined types are structs and classes. The use of pointers with structs and classes is similar. Assuming we have created a class *class_name* and struct *struct_name* The following is how we access the data of a struct with pointers.

```
struct_name  mystruct;
struct_name * structptr = &mystruct;
structptr->datamember; // method 1
(*structptr).datamember; // method 2
```

Both method 1 and method 2 are equivalent. However, method 1 uses fewer parentheses and is easier to read.

Compare it to the normal way of accessing struct data:

```
struct_name  mystruct;
mystruct.datamember1;
```

The following is how we can access class functions and public data using pointers:

```
class_name  myclass;
class_name * classptr = & myclass;
classptr ->member_function1(); // method 1
classptr->public_data_member1;// method 1
(*classptr).memberfunction();  // method 2
(*classptr).public_data_member1; //method 2
```

Remember using public data members is not a suggested OOP approach. The better approach is to define accessors (get functions) and modifiers (set functions) and use them to access class data.