Pointers

- Every storage location in the memory of a computer has a number which identifies it uniquely. This identifying number is called its **address**.

- Every variable in C++ is assigned a memory location and thus an address. Sometimes it is desirable to refer to a variable's address.

- A **pointer** is a variable that can hold an address as its value.

- **The Address Operator & :**
  One way to access the address of a variable is to place the **address operator**, an ampersand (**&**), in front of the variable's name.

declaration	**address**
int sum;	&sum
double sales;	&sales
char initial;	&initial

- **Declaring a Pointer Variable:**

```
int *iptr;
char *cptr;
double *dptr;
```

- **Assigning a Value to a Pointer Variable:**

```
iptr = &sum;
cptr = &initial;
cptr = &sales;
```

- In C++, there is a special value for a pointer to indicate that it is currently not pointing at anything, This value is **NULL**.

```
int *myPtr = NULL;          //this is called a NULL Pointer
int *myPtr2 = 0;            //0 is equivalent to NULL
```

- **The Dereferencing Operator * :**
  - The notation ***iptr** means the object that iptr points to (or the contents of the memory location pointed to by iptr).
  - This operator allows us to access memory locations through **indirect addressing.**

- **Using Pointers with * and &:**

```
int num1 = 5, num2 = 3;
int *iptr;

iptr = &num1;
cout << "num1 holds " << num1 << " and num2 holds ",
       << num2 << endl;
cout << "and the location iptr points to holds " << *iptr
       << endl;
iptr = &num2;
cout << "num1 holds " << num1 << " and num2 holds ",
       << num2 << endl;
cout << "and the location iptr points to holds " << *iptr
       << endl;
```

- Output:

```
num1 holds 5 and num2 holds 3
and the location iptr points to holds 5
num1 holds 5 and num2 holds 3
and the location iptr points to holds 3
```

- **Using Pointers to Change a Storage Location's Value:**

```
int num = 5;
int *iptr = &num;          // note initialization of p

*iptr = 10;
*iptr = *iptr + 1;
(*iptr)++;            //not *iptr++ which first increments iptr
```

Using Parameters Which are Pointers

- **Program:**

```
/* program to try to add one to function parameter */
#include <stdio.h>
void trytoadd1(int);

void main()
{
   int k = 5;

   cout << "in main - before call: " << k << endl;
   trytoadd1(k);
   cout << "in main - after call: " << k << endl;
}

/* function that tries to add one to its parameter */
void trytoadd1(int x)
{
   cout << "in function - before adding: " << x << endl;
   x++;
   cout << "in function - after adding: " << x << endl;
   return;
}
```

- Output:

```
in main - before call: 5
in function - before adding: 5
in function - after adding: 6
in main - after call: 5
```

WHAT WENT WRONG???

- **Correct Version of Program - using Pointers:**

```
/* program to add one to function parameter */
#include <stdio.h>
void add1(int *);

void main()
{
   int k = 5;

   cout << "in main - before call: " << k << endl;
   add1(&k);
   cout << "in main - after call: " << k << endl;
}

/* function that adds one to its parameter */
void add1(int *x)
{
   cout << "in function - before adding: " << *x << endl;
   (*x)++;
   cout << "in function - after adding: " << *x << endl;
   return;
}
```

\- Output:

```
in main - before call: 5
in function - before adding: 5
in function - after adding: 6
in main - after call: 6
```

- **Correct Version of Program - using Reference Parameters:**

```
/* program to add one to function parameter */
#include <iostream>
using namespace std;

void add1(int &);                    //function prototype

int main()
{
   int k = 5;

   cout << "in main - before call: "<< k << endl;
   add1(k);
   cout << "in main - after call: "<< k << endl;
   return 0;
}

/* function that adds one to its parameter */
void add1(int &x)        //x is receiving a reference to k
{
   cout << "in function - before adding: " << x << endl;
   x++;
   cout << "in function - after adding: " << x << endl;
   return;
}
```

\- Output:

```
in main - before call: 5
in function - before adding: 5
in function - after adding: 6
in main - after call: 6
```

- Notes:
  - **int &** means **reference to an integer.** This means that the formal parameter and the actual parameter (or argument) are the same, in the sense that any change to the value of the formal parameter will cause a like change to the value of the actual parameter. (The formal parameter becomes an **alias** for the actual parameter.)

Pointers and Arrays

- **In C++, a reference to the name of an array without a subscript, means the address of the array.**

- When an array is sent as a parameter to a function, only the address is sent.

- Because the array name is itself an address, we do not preface it by an & when passing it to a function.

```
int readdata(int numbers[])          // function header

num = readdata(mark);                // function call
```

- **Using * in the Function Header for an Array Parameter:**

```
int readdata(int *numbers)           // equivalent header
```

- Note:
  - a pointer variable holds a value that can change
  - an array name is a constant.

```
int a,b;
int *ptr;
int num[100];

ptr = &a;
ptr = &b;
ptr = num;
ptr = &num[0];
ptr = &num[1];
ptr = &num[99];

num = &a;       // illegal
num = ptr;      // illegal
```

- **Using & to Send an Address to an Array Parameter:**

```
sum = sumarray(&num[0]);        // same as sumarray(num)
```

- **Example:**

```
/* ... */
int sumarray(int numbers[], int n)
{
   int count,sum=0;

   for (count = 0; count < n; count++)
      sum += numbers[count];
   return(sum);
}
```

- To sum the elements 0 to count-1 of the num array:

```
sum = sumarray(num,count);
```

- To sum the elements 5 to 11 of the num array:

```
sum = sumarray(&num[5],7);
```

Pointer Arithmetic

- **Pointer Arithmetic** can be used to address the elements of an array without subscripts.

- We use direct address manipulation to move from one array element to the next.

- The techniques is based on the **displacement** or **offset** of an element, which measures how far an element is from the beginning of the array.

- Example:
  int num[5];

subscript	0	1	2	3	4
num	10	20	45	50	68
offset	+0	+1	+2	+3	+4

Note: num[i] is equivalent to *(num + i)

```
num[0] = 10;
*num = 10;

num[1] = 20;
*(num+1) = 20;

num[2] = 45;
*(num+2) = 45;
```

Note: int num[] is equivalent to int *num

- Example:

```
int i,sum=0;
int num[100];
int *ptr = num;

for (i = 0; i < 100; i++)
   sum += num[i];

for (i = 0; i < 100; i++)        // equivalent loop
   sum += *(num + i);

for (i = 0; i < 100; i++) {      // equivalent loop
   sum += *ptr;
   ptr++;
}
```

- We can use pointer notation to send a function the address of a location offset within an array.

```
sum = sumarray(&num[5],7);

sum = sumarray(num+5,7);        // equivalent call
```

Comparing Pointers

- In C++, relational operators can be used to compare pointers.

```
if (ptr1 == ptr2)          //compares addresses
if (*ptr1 == *ptr2)        //compares contents
```

```
// This program uses a pointer to display the contents
// of an integer array. It illustrates the comparison of
// pointers.
#include <iostream>
using namespace std;

int main()
{
   const int SIZE = 8;
   int set[ ] = {5, 10, 15, 20, 25, 30, 35, 40};
   int *numPtr = set;       // Make numPtr point to set

   cout << "The numbers in set are:\n";
   cout << *numPtr << " ";   // Display first element
   while (numPtr < &set[SIZE-1])
   {
      // Advance numPtr to the next element.
      numPtr++;
      // Display the value pointed to by numPtr.
      cout << *numPtr << " ";
   }

   // Display the numbers in reverse order
   cout << "\nThe numbers in set backwards are:\n";
   cout << *numPtr << " "; // Display last element
   while (numPtr > set)
   {
      // Move backward to the previous element.
      numPtr--;
      // Display the value pointed to by numPtr.
      cout << *numPtr << " ";
   }
   return 0;
}
```

Pointers to Constants

- A pointer to a constant can not be used to change the value it points to.

- Example:

```
const int SIZE = 4;
const double payRates[SIZE] = (18.55,17.45,12.85,14.97};

...

void displayPayRates(const double *rates, int size)
{
   //display all the pay rates
   for (int count = 0; count < size; count++)
      cout << "pay rate for employee" << count+1
            << " is $" << *(rates+count) << endl;

   return;
}
```

- Note:
  The variable pointer **rates** points to a **constant double.** The identifier rates itself is a variable; that is, it can point to different constant doubles.

- Note:
  Use of **const** in the function header, protects the data for being modified within the function.

Constant Pointers

- A **constant pointer** is a pointer that once initialized with an address, can not point to anything else. (While the address can not change, the data at the address can change.)

- Example:

```
int value = 22;
int * const ptr = &value;         //ptr is a constant pointer

*ptr = 100;
*ptr = 200;
```

- Constant pointers can be used without initialization in function headers. The pointer will be initialized by the argument upon a call to the function. The argument value can be different for each call.

- Example:

```
void setToZero(int * const ptr)
{
    *ptr = 0;

    return;
}


//legal calls to the function setToZero()
int x,y,z;
...

setToZero(&x);
setToZero(&y);
setToZero(&z);
```

Dynamic Memory Allocation

- Dynamic memory allocation allows for the allocation of storage for a variable while the program is running ("on the fly").

- Dynamic memory allocation is only possible through the use of pointers.

- Use the **new** operator to allocate memory:

- Example:

```
int *iptr;

iptr = new int;   //requests memory from the OS to store
                  //an integer

*iptr = 50;       //use the newly allocated memory
```

- Example:

```
int *iptr;

iptr = new int[100];   //requests memory from the OS to
                       // create an array of 100 integers

for (int count = 0; count < 100; count++)
   iptr[count] = 1;    //uses the newly allocated array
```

- Note: When memory can not be allocated (e.g., you asked for too much), the **new** operator will by default cause termination of the program with an appropriate error message in a process known as **throwing an exception**. (This will covered at a later date.)

Releasing (or Deleting) Dynamic Memory:

- When a program has finished using dynamicaaly allocated memory, it should release it for future use.

- The **delete** operator is used to free memory that was previously allocated with **new**.

- Example:

```
delete iptr;            //frees a dynamically allocated variable

delete [] iptr;         //frees a dynamically allocated array
```

- Note:
  Memory dynamically allocated in a class constructor should be deleted in the class destructor.

Returning Pointers from Functions

- Functions **must not** return a pointer to a local variable:

```
string * getName()
{
   string myname;
   string *name = &myname;
   cout << "Enter your name: ";
   getline(cin, *name);
   return name;      //the variables are destroyed upon return
}
```

- Functions can return a pointer to an item that was passed as an argument:

```
string * getName( string *name)
{
   cout << "Enter your name: ";
   getline(cin, *name);
   return name;      //the address name points to still exists
                     //upon return
}
```

- Functions can return a pointer to dynamically allocated memory:

```
string * getName()
{
   string *name;

   name = new string;        //dynamic memory allocation
   cout << "Enter your name: ";
   getline(cin, *name);
   return name;             //the string still exists upon return
}
```

Pointers to Structures

- **Example of a Pointer to a Structure:**

```
struct Name {
    string last;
    string first;
};

struct Classmark {
    int test[5];
    double average;
    char lettergrade;
};

struct Student {
    Name name;
    int numclasses;
    Classmark class[5];
    double overallavg;
};

Student student;              //a Student object

Student *studentptr;          //a pointer to a Student object
```

- **Usage:**

```
studentptr  = &student;
(*studentptr).numclasses = 5;
(*studentptr).overallavg = 3.15;
cout << (*studptr).numclasses) << endl;
cout << (*studptr).name.last) << endl;
cin >> &(*studptr).numclasses;
cin >> (*studptr).name.last;
```

- **The -> Operator:**

`ptr -> member` is the same as `(*ptr).member`

```
studentptr -> numclasses = 5;
studentptr -> overallavg = 3.15;
cout << studptr -> numclasses;
cout << studptr -> name.last;
cin >> studptr -> numclasses;
cin >> studptr -> name.last;
```

Pointers to Class Objects

```
class Rectangle {
    int width;
    int height;
  public:
    void setData(int w, int h)
    {
        width = w;
        height = h;
        return;
    }
    Rectangle ()
    {
        width = 0;
        height = 0;
    }
    Rectangle(int h, int h)
    {
        setData(w,h);
    }
```

- **Usage:**

```
Rectangle box;                 //a Rectangle object

Rectangle *boxPtr;             //a pointer to a Rectangle object

boxPtr = &box;                 //boxPtr points to box

boxPtr -> setData(15,12);  //using the method setData()
```

- **Dynamic Allocation of Class Objects:**

```
boxPtr1 = new Rectangle;      //invokes default constructor

boxPtr2 = new Rectangle(10,20)     //invokes constructor

delete boxPtr;                           //invokes desctructor
```


Selecting Members of Objects

- Sample Structure with Pointer Member:

```
struct GradeInfo {
   string name;          //student name
   int *testScores;      //dynamically allocated
   double average;       //test average
};


GradeInfo student, *stPtr = &student;
```

- Example:

```
//display the value pointed to by the testScores member

cout << *student.testscores;

cout << *stPtr -> testScores;       //equivalent

cout << *(*stPtr).testScores;       //equivalent
```

Note: The following are pointers to an int and are equivalent

```
   student.testScotes

   stPtr -> testScores

   (*stPtr).testScores
```

Array of Pointers

- Example:
  A case study to demonstrate how an array of pointers can be used to display the contents of a second array in sorted order, without sorting the second array.

  See sample project