MECH215
Lecture Notes 8

Pointers

C++ has a data type called a POINTER. The * symbol is used to define a pointer. For example, suppose we have an integer variable called x, we would declare it as follows:

```
int x;
```

To declare a pointer to an integer would use use the following declaration:

```
int* somePointer;
```

Pointer declarations are best understood if we 'read' them backwards as in

1. First read this as "somePointer" is a
2. Read the * symbol as "pointer to an"
3. Finally read the data type to which to pointer is pointing to as integer

If we put the above three ideas together, we would interpret the C++ declaration int* somePointer to mean:

"somePointer is pointer to an integer"

Pointers to other data types are declared in a similar fashion, the following declare pointers to a char, float, double respectively:

```
char*   charPointer;
float*  floatPointer;
double* doublePointer;
```

A variable which is declared to be a pointer simply holds the ADDRESS of some object. Once we have declared a pointer, we can give it a value by using the address-of (&) operator as in:

```
int x;
int* somePointer;
somePointer = &x;   // assign the address where x is stored
                    // to the pointer variable
```

Suppose memory location with address 1000 is used to store the integer variable x and memory location with address 2000 is used to store the pointer variable somePointer. The situation looks like:

```
                contents          address

                ----------
x                  ?                1000
                ----------

                ----------

                ----------
somePointer      1000               2000
                ----------
```

Once we have a pointer to a variable, we can access the variable using the pointer dereferencing operator (*) (C++ is a bit confusing in the sense that the same symbol has more than one meaning depending on where it is used in a program).

Suppose we use the assignment operator to give a value of 4 to the integer variable x:

```
x = 4;
```

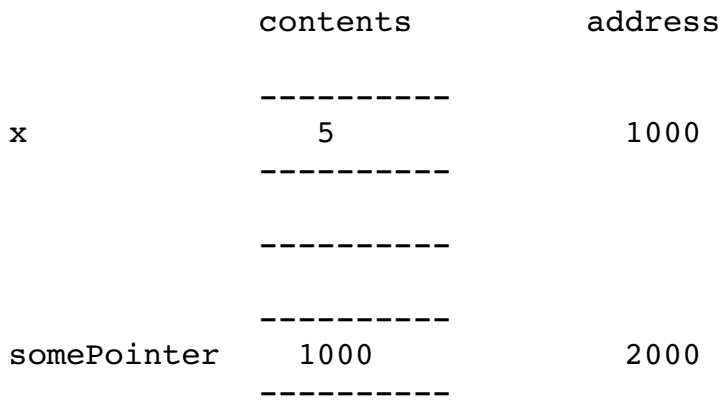
The contents of memory now look like:

```
                contents          address

                ----------
x                  4                1000
                ----------

                ----------

                ----------
somePointer      1000               2000
                ----------
```

We can change the value stored in this x variable by the following:

```
*somePointer = 5;
```

This statement means "go to the memory location at address 1000 and store the value 5 in this location". The situation in memory now looks like:



Here is a small C++ program which illustrate the pointer concepts we have discussed so far:

```
// Author: Ted Obuchowicz and John Peach
// pointer.cpp
// example program illustrating use of pointers and dereferencing

#include <iostream>

using namespace std;

int main() {

    // Define some sample variables
    int    mick    = 4;
    int    keith   = 23;
    float  ron     = 4.567;
    double charlie = 3.333333333;

    // Declare some pointers to our variables
    int*    mick_ptr    = &mick;
    int*    keith_ptr   = & keith; // C++ does not care if you put a space between
                                   // & and variable
    float*  ron_ptr     = &ron;
    double* charlie_ptr = &charlie;

    // print out the addresses of the variables and values of their pointers
    // the pointers will contain the addresses of the respective variables

    cout << "mick is stored at " << &mick
         << ":\tValue stored in mick_ptr = " << mick_ptr  << endl;

    cout << "keith is stored at " << &keith
         << ":\tValue stored in keith_ptr = " << keith_ptr << endl;

    cout << "ron is stored at " << &ron
         << ":\tValue stored in ron_ptr = " << ron_ptr  << endl;

    cout << "charlie is stored at " << &charlie
         << ":\tValue stored in charlie_ptr = " << charlie_ptr  << endl;
```

```
    //  Change the values stored in the variables using pointer dereferencing
    *mick_ptr    = 345;
    *keith_ptr   = 222;
    *ron_ptr     = 2345.678;
    *charlie_ptr = 5555.55555;

    // print out the new values again using pointer dereferencing
    cout << "mick = "    << *mick_ptr    << endl;
    cout << "keith = "   << *keith_ptr   << endl;
    cout << "ron = "     << *ron_ptr     << endl;
    cout << "charlie = " << *charlie_ptr << endl;

    return 0;
}
```

The program output is:

```
mick is stored at 0x7fffe864d0e4:     Value stored in mick_ptr = 0x7fffe864d0e4
keith is stored at 0x7fffe864d0e8:    Value stored in keith_ptr = 0x7fffe864d0e8
ron is stored at 0x7fffe864d0ec:      Value stored in ron_ptr = 0x7fffe864d0ec
charlie is stored at 0x7fffe864d0b8:  Value stored in charlie_ptr = 0x7fffe864d0b8
mick = 345
keith = 222
ron = 2345.68
charlie = 5555.56
```

Some Dos and Don'ts For Pointers

1) Incorrect assignment of pointers

C++ is a very strongly typed language, the language does not allow the assignment of the address of a non-integer data type to an integer pointer variable for example. Basically, if a pointer to a certain data type (int, char, float, double, etc) has been declared, the pointer may be only assigned the address of a variable of the appropriate data type. In other words:

int* may only receive the address of some int variable
char* may only receive the address of some char variable
float* may only receive the address of some float variable
etc.

The following program contains an error since we are attempting to assign the address of a float to something which has been declared to an integer pointer:

```
// Author: Ted Obuchowicz and John Peach
// pointerWrongType.cpp
// example program illustrating typical pointer errors

int main() {

    int   x;
    float f;
```

```
    int* pointer = &f; // bad... the compiler will vomit on this line

    return 0;
}
```

The compiler reports the following error:

```
pointerWrongType.cpp: In function ‘int main()’:
pointerWrongType.cpp:13:21: error: cannot convert ‘float*’ to ‘int*’ in initialization
```

2) Attempting to deference a non-initialized pointer

Attempting to deference a non-initialized pointer variable will give undefined and unpredictable results. Merely declaring a pointer variable does not give it an initial value. This is a common error which has kept many a C++ programmer up at night trying to debug their program. Here is a sample of this hard to trace error:

```
// Author: Ted Obuchowicz and John Peach
// pointerUninitialized.cpp
// example program illustrating use of

#include <iostream>

using namespace std;

int main() {

    int  fender;
    int* les_paul;              // initialized (random garbage in it)

    cout << *les_paul << endl;  // garbage will be produced as output

    return 0;
}
```

This program compiles fine, but will probably produce the dreaded Segmentation Fault run-time error message. Or, it may write garbage depending on what address just happens to be in les_paul:

At run-time we may get the message:

```
Segmentation fault (core dumped)
```

SO ALWAYS REMEMBER TO GIVE AN ADDRESS TO A POINTER VARIABLE USING THE ADDRESS OF OPERATOR (&) APPLIED TO SOME PREVIOUSLY DECLARED VARIABLE.

Void Pointers

There is one exception to the rule about assigning the address of an int to a int pointer, the address of a float to a float pointer, etc. C++ allows the programmer to declare a VOID POINTER. A VOID POINTER may be assigned the address of any type. For example;

```
void* pointerToAnything;
void* anotherPointerToAnything;
int anInteger;
float aFloat;
```

the following assignments of addresses to void pointers are legal:

```
pointerToAnything        = &anInteger;
anotherPointerToAnything = &aFloat;
```

Although void pointers may be assigned the address of any data type, VOID POINTERS MAY NOT BE DEREFERENCED DIRECTLY!

The following program is incorrect since it attempts to deference void pointers directly:

```
// Author: Ted Obuchowicz and John Peach
// voidInvaildDereference.cpp
// example program illustrating INCORRECT  use of void pointers

#include <iostream>

using namespace std;

int main() {

    // declare some void pointers
    void* pointerToAnything;
    void* anotherPointerToAnything;

    // declare some variables of type int and float
    int anInteger = 5;
    float aFloat  = 4.56;

    // assign some addresses to the void pointers..
    // so far this is all legal C++
    pointerToAnything        = &anInteger;
    anotherPointerToAnything = &aFloat;
```

```
    // now we get into trouble by trying to directly dereference
    // the void pointers
    *pointerToAnything = 67;  // compiler complains about this line
    cout << anInteger << endl;

    return 0;
}
```

The compiler output is:

```
voidDereference.cpp: In function 'int main()':
voidDereference.cpp:25:6: error: 'void*' is not a pointer-to-object type
```

The reason why it is forbidden to dereference a void pointer directly is that the compiler has no information as to what the void pointer is pointing to. That is, it does not have a data type (int, float etc) so it does not know how to store the information.

Void pointers may be dereferenced PROVIDED THEY ARE CAST TO A POINTER OF THE APPROPRIATE DATA TYPE BY USING THE CAST OPERATOR:

The following program is the correct version of the above program. It applies the cast operator to the void pointers, then it dereferences the "casted" pointers:

```
// Author: Ted Obuchowicz and John Peach
// voidDereference.cpp
// example program illustrating CORRECT use of
// void pointers by first casting the void pointer to
// an appropriate pointer

#include <iostream>

using namespace std;

int main() {

    // declare some void pointers
    void* pointerToAnything;
    void* anotherPointerToAnything;

    // declare some variables of type int and float
    int anInteger = 5;
    float aFloat  = 4.56;

    // assign some addresses to the void pointers. This is all legal C++
    pointerToAnything        = &anInteger;
    anotherPointerToAnything = &aFloat;

    // get into trouble by trying to directly dereference the void pointers
    * (int*)pointerToAnything = 67;  // cast to an integer pointer
    cout << anInteger << endl;       // then apply the dereference operator
```

```
    * (float*)anotherPointerToAnything = 7778.89;  // cast to a float pointer
    cout << aFloat << endl;

    return 0;
}
```

The program output is:

```
67
7778.89
```

Pointers And Array Names

In C++, the name of an array is the same as the starting address of the first element in the array. We can use this fact to traverse the array using pointer notation:

```
// Author: Ted Obuchowicz and John Peach
// arrayPointer.cpp
// example program illustrating use of arrays as pointers

#include <iostream>

using namespace std;

int main()
{

    const unsigned short SIZE = 5;  // The size of the array
    int array[SIZE] = {0,1,2,3,4};  // some array
    int * ptr;                      // a pointer to the array

    ptr = &array[0];                // same as: ptr = array

    for(int i = 0; i < SIZE; i++) {
        cout <<  (*ptr) << ptr << endl;
        ptr += 1;
    }

    return 0;
}
```

The program output is:

```
00x7fff89ce9e00
10x7fff89ce9e04
20x7fff89ce9e08
30x7fff89ce9e0c
40x7fff89ce9e10
```

In the above program an integer pointer called ptr is assigned the name of the array, and then a for loop is used to successively one 1 to this starting address to print out the array contents and the address at which the contents are stored.

Pointer Arithmetic

The above program illustrates the concept of what is referred to as POINTER ARITHMETIC in C++. If we have an integer pointer (ptr) and we perform the assignment:

```
ptr = ptr + 1;
```

The actual value of ptr will be incremented by 4 and not by 1. Look at the addresses contained in the pointer ptr in the above program at every loop iterations:

00x7fff89ce9e00 for the first time through the loop ptr = ptr + 0;
10x7fff89ce9e04 for the second time through the loop ptr = ptr + 1;
20x7fff89ce9e08 for the third time through the loop 20x7fff89ce9e08 = 10x7fff89ce9e04 + 1
etc.

When we write down something like:

pointer_variable = pointer_variable + 1

the compiler really translates this to

pointer_variable = pointer_variable + 1 * sizeof(thing that pointer_variable is pointing to)

Since an integer occupies 4 bytes of storage (sizeof(int) = 4), when we add "1" to an integer pointer we are really increasing the value of the pointer by 4. If we add "1" to a character pointer, we are increasing the address by 1 since a character occupies only 1 byte of storage. This means that it is incorrect to add "1" to a void pointer since the compiler has no way of determining how much to add to the void pointer since it does not know what the void pointer is pointing to:

```
// Author: Ted Obuchowicz and John Peach
// April 17, 2002
// example program illustrating INCORRECT  use of
// void pointers
```

```
int main() {

    // declare some void pointers
    void* pointerToAnything;

    int anInteger = 5;

    pointerToAnything = &anInteger;

    // this is wrong!
    pointerToAnything = pointerToAnything + 1;

    return 0;
}
```

The compiler output is:

```
pointerVoidWrong.cpp: In function ‘int main()’:
pointerVoidWrong.cpp:18:45: warning: pointer of type ‘void *’ used in
arithmetic [-Wpointer-arith]
```

Here is another method of accessing the elements of the array using only the array name, an offset and pointer dereferencing:

```
// Author: Ted Obuchowicz and John Peach
// pointerDereference.cpp
// example program illustrating use of
// address offset notation

#include <iostream>

using namespace std;

int main() {

    int array[5] = {0,1,2,3,4};

    for(int i = 0; i < 4; i++) {
        cout <<  *(array + i) << "  " <<  array + i << endl;
    }

    return 0;
}
```

The output is (exactly the same as before):

```
0   0x7fff8c8fa190
1   0x7fff8c8fa194
2   0x7fff8c8fa198
3   0x7fff8c8fa19c
```

The concept being exploited in this program is that of pointer arithmetic combined with the fact that an array name is the same as the starting address of the first element of the array. So

array + 0 = address of array[0]
array + 1 = address of array[1]
array + 2 = address of array[2]
array + 3 = address of array[3]

IT IS IMPORTANT TO NOTE THAT WE CANNOT DO SOMETHING LIKE

array = array + 1

if the array has been created with a fixed size. In the example, it was [5]. The name of an array is treated as a constant address, once the compiler has allocated memory to hold an array, this starting address cannot be changed!

The following program does not compile since it attempts to change the address of an array:

```
// Author: Ted Obuchowicz and John Peach
// pointerChange.cpp
// example program illustrating use of
// address offset notation

#include <iostream>

using namespace std;

int main() {

    int array[5] = {0,1,2,3,4};

    for(int i = 0; i < 4; i++) {
        cout <<  *(array) << "  " <<  array << endl;
        array = array + 1; // this is wrong... array is treated as a
                           // constant pointer
    }

    return 0;
}
```

The compiler reports the error:

```
pointerChange.cpp: In function ‘int main()’:
pointerChange.cpp:16:25: error: incompatible types in assignment of ‘int*’ to ‘int [5]’
```

Constant Pointers

C++ allows for the declaration of constant pointers. A constant pointer must be assigned an address when it is declared, and once it has been assigned this address it cannot be subsequently changed to another address:

```
int x;
int* const ptr  = &x;
4  3   2    1   (this is to help you read the pointer declaration)
```

The pointer declaration is to be read as:

1: ptr is a
2: constant
3: pointer
4: to an integer

It would be an error to attempt to assign the value of ptr after its declaration:

```
int x;
int* const ptr;
ptr = &x; // wrong...
```

It would also be an error to attempt to reassign the value of th constant pointer:

```
int x;
int y;
int* const ptr = &x; // this is ok

ptr = &y; // INCORRECT !! assignment of read-only variable `ptr'
```

References And Constant Pointers

The C++ compiler treats reference variables as CONSTANT POINTERS. The compiler performs the dereferencing of these constant pointers for you automatically. For example:

```
int some_int;
int& some_reference = &some_int;

some_int = 5;
cout << some_int << endl; // produces 5 as output
```

```
some_reference = 10;
cout << some_int << endl; // produces 10 as output
```

The compiler actually produces run -time code equivalent to the following:

```
int some_int;
int* const some_reference = &some_int;

some_int = 5;
cout << some_int << endl; // produces 5 as output

*some_reference = 10;     // dereference the reference
                          // variables for you automatically
cout << some_int << endl; // produces 10 as output
```

Pointers to constant data :

Consider the following declaration:

```
const int i = 5;
int const * ptr;
ptr = &i;
```

ptr is a pointer to an integer constant. The compiler allows the integer to be read (directly and indirectly through the pointer) but any attempt to deference the pointer ptr yields a compile time error stating "assignment of read-only location". This is the case in the following example program:

```
// Author: Ted Obuchowicz and John Peach
// pointerConstant.cpp
// example program illustrating use of
// constant pointers

#include <iostream>

using namespace std;

int main() {

    const int i = 5;
    int const * ptr;
    ptr = &i;

    // we may read the value of the integer constant through
```

```
    // its pointer
    cout << *ptr << endl;

    // we are not allowed to update a pointer which is pointing
    // to a constant
    *ptr = 10;

    return 0;
}
```

It is a good thing the compiler enforces this, were it not the case, it would be possible to change the value of a constant indirectly though a pointer.

The address of a non-constant variable may be assigned to a pointer to a constant, but the compiler will still not allow you to dereference the pointer:

```
// Author: Ted Obuchowicz and John Peach
// pointerNonConstant.cpp
// example program illustrating use of
// constant pointers

#include <iostream>

using namespace std;

int main() {

    int i = 5;
    int const * ptr;
    ptr = &i; // assign the address of a non-constant integer
              // i to a pointer which points to an integer constant
              // we may read the value of the integer constant
              // through its pointer
    cout << *ptr << endl;

    // we are not allowed to dereference a pointer which is
    // pointing to a constant
    *ptr = 10;

    return 0;
}
```

Constant Pointers to Constant Data

This is the most restrictive, neither the value of the pointer, nor the contents that it is pointing to can be changed. A declaration of a constant pointer which points to an integer constant is:

```
const int x = 5;
const int * const y = &x;
```

A Generic printmatrix Function Which Uses Pointers

We are now in a position to write a generic printMatrix function which is capable of printing out any sized two-dimensional array. It makes use of the address translation mechanism, pointer arithmetic, and void pointers:

```
// Author: Ted Obuchowicz and John Peach
// printMatrix.cpp
// example program illustrating use of
// pointers to access the elements of a two-dimensional array

#include <iostream>

using namespace std;

void printMatrix(void* data, int rows, int columns, char dataType);

int main() {

    int littleMatrix[3][3] = { 1,2,3,4,5,6,7,8,9 };

    int bigMatrix[5][5]  = { 10, 11, 12, 13, 14,
                             16, 17, 18, 19, 20,
                             22, 23, 24, 25, 26,
                             28, 29, 30, 31, 32,
                             34, 35, 36, 37, 38 };

    char arrayOfChars[2][2] = { 'a', 'b', 'c', 'd'};

    float arrayOfFloats[4][4] = { 1.23, 2.34, 3.33, 4.44,
                                  5.55, 6.66, 7.77, 8.88,
                                  9.99, 10.10, 11.11, 12.12,
                                 13.13, 14.14, 15.15, 16.16};

    cout << "The little matrix is : " << endl;
    printMatrix(&littleMatrix[0][0], 3, 3, 'i');
    cout << endl;

    cout << "The bigMatrix is : " << endl;
    printMatrix(&bigMatrix[0][0], 5, 5, 'i');
    cout << endl;

    cout << "The character array is : " << endl;
```

```
    printMatrix(&arrayOfChars[0][0], 2,2, 'c');
    cout << endl;

    cout << "The float array is : " << endl;
    printMatrix(&arrayOfFloats[0][0], 4,4, 'f');
    cout << endl;

    return 0;
}

void printMatrix(void* data, int rows, int columns, char dataType) {
    for(int i = 0; i < rows; i++){
        for(int j =0; j < columns; j++) {
            switch (dataType) {
                case 'i': cout << *( (int*)data   + (i * columns) + j) << " ";
                          break;
                case 'c': cout << *( (char*)data  + (i * columns) + j) << " ";
                          break;
                case 'f': cout << *( (float*)data + (i * columns) + j) << " ";
                          break;
                default : cout << "Invalid data type" << endl;
                          return;
            }

            if ( j == columns - 1){
                cout << endl;
            }
        }
    }
}
```

The program output is:

```
The little matrix is :
1 2 3
4 5 6
7 8 9

The bigMatrix is :
10 11 12 13 14
16 17 18 19 20
22 23 24 25 26
28 29 30 31 32
34 35 36 37 38

The character array is :
a b
c d
```

```
The float array is :
1.23 2.34 3.33 4.44
5.55 6.66 7.77 8.88
9.99 10.1 11.11 12.12
13.13 14.14 15.15 16.16
```

Strings And Character Arrays

There is a subtle difference between the two following declarations:

```
char * s = "ABC";
```

and

```
char s1[4] = {'A', 'B', 'C', '\0' );
```

The first creates a pointer to a character which is assigned the address of a CHARACTER STRING LITERAL CONSTANT stored in a portion of memory which may only be read from (the CHARACTER STRING LITERAL CONSTANT is terminated by the '\0' character). The second declares and initializes a 4 element of array of characters. The last element in the array is thr null character '\0'. The elements of the array s1 may both be read from and written to at will.

Here is a program which explains the subtleties of the char * s = "ABC";

```
// Author: Ted Obuchowicz and John Peach
// stringLiteral.cpp
// example program illustrating string literals

#include <iostream>

using namespace std;

int main() {

    char * s = "ABC";  // the compiler treats the string "ABC\0"
                       // as constant since string literals are
                       // treated as constants
                       // the pointer s points to the first character
                       // in the string... this means that strings
                       // declared in this manner may be READ from
                       // but not written to

    cout << "s is " << s  << endl; // this is OK, read from the string

    // try to modify s
    *s = 'Z'; //  this will compile fine, but cause a SEGMENTATION FAULT
              //  during run-time
```

```
    cout << "modified s is " << s  << endl;
    cout << "modified s is " << s  << endl;

    return 0;
}
```

When we compile the program with g++ we get a warning but it does compile. The compiler will convert a string constant (a.k.a. a string literal) to a char* but this feature has been removed from the language.

Here is the warning:

```
stringLiteral.cpp: In function ‘int main()’:
stringLiteral.cpp:11:16: warning: deprecated conversion from string
constant to ‘char*’ [-Wwrite-strings]
```

As noted in the program comments, a SEGMENTATION FAULT is generated at run-time by the above program. Here is a different version of the program which uses the array style method of working with character strings:

```
// Author: Ted Obuchowicz and John Peach
// cstring.cpp
// example program illustrating string literals

#include <iostream>

using namespace std;

int main() {

    char  s[] = {'A', 'B', 'C', '\0' };

    cout << "s is " << s  << endl; // this is OK, it will simply start printing
                                   // characters starting at address s until a
                                   // '\0' is encountered

    // try to modify s
    *s = 'Z'; //   this is ok as the array is both read and writable

    cout << "modified s is " << s  << endl;

    return 0;
}
```

This program runs fine:

```
s is ABC
modified s is ZBC
```

A String Reverse Function

Here is a nice function which joins two character arrays but the second array is copied BACKWARDS. It makes use of pointers.

```
// Author: Ted Obuchowicz and John Peach
// reverseString.cpp
// example program illustrating use of char pointer

#include <iostream>
using namespace std;

void reverseString(char s1[], char s2[]);

int main() {

    char string1[50] =  "Keith";
    char string2[50] =  "Richards";

    cout << string1 << endl;
    cout << string2 << endl;

    reverseString(string1, string2);

    cout << string1 << endl;

    return 0;
}

void reverseString(char s1[], char s2[]) {
    // find the end of the first string designated by a '/0'

    int string1End = 0;
    while (s1[string1End]) { string1End++; }

    int string2End = 0;
    while (s2[string2End]) { string2End++; }

    for( int i = --string2End; i > -1; i--) {
        s1[ string1End++] = s2[i];
    }

    // now add the '\0' character to s1
    s1[++string1End] = '\0';
}
```

Array of Pointers

C++ lets a programmer declare and use an array of pointers:

```
/ Author: Ted Obuchowicz and John Peach
// arrayOfPointers.cpp
// example program illustrating use of
// an array of pointers

#include <iostream>

using namespace std;

int main() {

    char* rollingStones[4];  // declare an array of 4 character pointers

    // initialize the 4 pointers to chars to some string literal constants
    // note, we may only READ these string constants and cannot write to
    // them..

    rollingStones[0] = "Keith";
    rollingStones[1] = "Mick";
    rollingStones[2] = "Charlie";
    rollingStones[3] = "Ron";

    /* this will set the values of the 4 pointers to point to the starting
       address of the 4 strings

          rollingStones[0] -------> |'K'|'e'|'i'|'t'|'h'|'\0'|
                                    -------------------------

          rollingStones[1] -------> |'M'|'i'|'c'|'k'|'\0'|
                                    ----------------------

          rollingStones[2] -------> |'C'|'h'|'a'|'r'|'l'|'i'|'e'|'\0'|
                                    ---------------------------------

         rollingStones[3] -------->  |'R'|'o'|'n'|'\0'|
                                     -----------------
    */
    // print out the strings
    for(int i = 0; i < 4; i++) {
        cout << rollingStones[i] << endl;
    }

    return 0;
}
```

The line:

```
char* rollingStones[4];
```

declares the variable named rollingStones to be a 4 element array of pointers to characters (i.e. each element of the array is a pointer to a character)

The lines:

```
rollingStones[0] = "Keith";
rollingStones[1] = "Mick";
rollingStones[2] = "Charlie";
rollingStones[3] = "Ron";
```

initializes the 4 pointers to characters stored in the elements of the array rollingStones to 4 different string literal constants
(which may only be READ FROM).

The 4 strings may be printed out by using the loop:

```
for(int i = 0; i < 4; i++) {
    cout << rollingStones[i] << endl;
}
```

Pointers To Functions

C++ allows one to declare POINTERS TO FUNCTIONS. Consider the following two function prototypes:

```
void*   f1 (void);
void (*f2) (void);
```

The first declares a function which is called f1, this function receives no arguments and returns a void pointer.

The second declares f2 to be a POINTER TO A FUNCTION which receives no arguments and returns nothing.

Here is a program which makes us of a pointer to a function and invokes the function through its pointer:

```
// Author: Ted Obuchowicz and John Peach
// printMessage.cpp
// example program illustrating use of pointers to functions

#include <iostream>
```

```
using namespace std;

void printMessage(void);

int main() {

    // declare f as a pointer to a function which returns a void
    // and takes no argument.  That is, f, is a variable
    void (*f) (void);

    // Assign f the address of the printMessage function
    // You can also use: f = printMessage;
    // Note:  There are no () after printMessage
    f = &printMessage;

    // invoking it indirectly through the pointer f
    // This is going to invoke printMessage()
    (*f)();

    return 0;
}

void printMessage(void) {
    cout << "Hello World" << endl;
}
```

The program output is:

```
Hello World
```

Arrays of Pointers to Functions:

Consider the following declaration:

```
void (*array[4]) (void);
```

This declares the variable called array to be an array of 4 elements. Each element of the array is a POINTER TO A FUNCTION which takes no arguments and returns nothing. We can use this array be assigning its 4 elements the 4 addresses of some functions, then invoke the functions indirectly by dereferencing the 4 pointers stored in the array. The following program does this:

```
// Author: Ted Obuchowicz and John Peach
// arrayOfFunctions.cpp
// example program illustrating use of
// pointers to functions
```

```
#include <iostream>

using namespace std;

const int SIZE = 5;     // Size of the globalArray array
int globalArray[SIZE];

void clear(void);
void addOne(void);
void printOut(void);

int main() {

    void (*  array[4]) (void);  // an array of 4 pointers to FUNCTIONS
                                // which do not return a value

    array[0] = &clear;
    array[1] = &printOut;
    array[2] = &addOne;
    array[3] = &printOut;

    // now call each function one-by-one
    for(int i = 0; i < 4; i++) {
        (*array[i])();
    }

    return 0;
}

void clear(void) {
    for(int i = 0; i < SIZE; i++) {
        globalArray[i] = 0;  // set all the 5 elements to 0
    }
}

void addOne(void) {
    for(int i = 0; i < SIZE; i++) {
        globalArray[i] += 1;
    }
}

void printOut(void){
    for(int i = 0; i < SIZE; i++) {
        cout <<   globalArray[i] << endl;
    }
}
```

The program output is:

```
0
0
0
0
0
1
1
1
1
1
```

Pointers and structs

Consider a struct declared as:

```
struct integerChar {
    int i;
    char t;
};
```

We can declare a variable to be of this type with the line:

```
integerChar struct1;
```

Next, we can declare a pointer to the struct with:

```
integerChar* ptrStruct;
```

Now, we can give the address of the variable struct1 to the pointer:

```
ptrStruct = &struct1;
```

We can access the individual fields of the struct indirectly by pointer dereferencing:

```
(*ptrStruct).i = 5;     // dereference the pointer and access the integer field
(*ptrStruct).t = 'x';   // dereference the pointer and access the char field
```

This method is a bit awkward and clumsy. C++ allows for a 'shorthand notation' when working with pointers to structs:

```
ptrStruct -> i = 10;
ptrStruct -> t = 'z';
```

These ideas are summarized in the following program:

```
// Author: Ted Obuchowicz and John Peach
// structPointer.cpp
// example program illustrating use of pointers to struct

#include <iostream>

using namespace std;

int main() {

    struct integerChar{
        int i;
        char t;
    };

    integerChar struct1;

    integerChar* ptrStruct;

    ptrStruct = &struct1;   // assign the address of the variable struct1 to
                            // the pointer

    (*ptrStruct).i = 5;     // dereference and access the integer field
    (*ptrStruct).t = 'x';   // dereference and access the char field

    cout << "i is " << struct1.i << endl;
    cout << "t is " << struct1.t << endl;

    // alternate method of accessing the elements of the struct using the
    // pointer to the struct

    ptrStruct -> i = 10;
    ptrStruct -> t = 'z';

    cout << "i is " << ptrStruct -> i << endl;
    cout << "t is " << ptrStruct -> t << endl;

    return 0;
}
```

The program output is:

```
i is 5
t is x
i is 10
t is z
```

The Library <cstring>

There is a built-in library called <string> which has some useful string manipulating functions. Some of these functions are:

```
char *strcpy(char *, const char *);
char *strcat(char *, const char *);
int strcmp(const char *, const char *);
```

Your textbook gives numerous examples of the use of these functions. The reader is advised to refer to these examples.

Dynamic Memory Allocation With new And delete

C++ has another method of assigning a value to a pointer variable. This method makes use of the new operator to DYNAMICALLY ALLOCATE MEMORY.

It is often convenient in certain programming situations to be able to request new memory space during program run time. C++ allows this with the 'new' operator. The 'new' operator is used to request a certain amount of memory at program run time, if the call to 'new' is successful, 'new' returns the starting address of this dynamically allocate memory. If the call to 'new' is not successful (if there is no more memory left for example) then 'new' returns the value 0.

Here is a simple program which dynamically allocates enough room for an integer:

```
/ Author: Ted Obuchowicz and John Peach
// newDelete.cpp
// example program illustrating use of
// new and delete operators

#include <iostream>

using namespace std;

int main() {

    int* integerPointer;

    /*  Request enough memory from the heap to hold an integer
        if the request is successful, new returns the address,
        if there is no more memory, new returns the value 0.
    */

    integerPointer = new int;

    // test to make sure memory was allocated
    if ( integerPointer ) {

        // print out some garbage number since we did not initialize
        cout << *integerPointer << endl;
```

```
        // assign the value 10 to this memory location
        *integerPointer = 10;

        // Report address and value
        cout << "Address of newly allocated memory = " << integerPointer << endl;
        cout << "value stored at this address = " << *integerPointer << endl;

        // release the memory back to the heap
        // The pointer still exists, just the int has been released
        delete integerPointer;
    }

    return 0;
}
```

The program output is:

```
261944
Address of newly allocated memory = 0x986010
value stored at this address = 10
```

It is very important to keep track of the memory that has been allocated with new and make sure that it is deleted when it is no longer needed. Failing to do so is called a "memory leak" and the memory that your program uses can rapidly grow with time. This can cause the system to slow down and even run out of resources. While a modern PC has a large amount of resources, you may be running your code on small controller or other imbedded devices where memory is very limited.

There is a variation of the new operator which initializes the memory allocated from the heap with some user-specified value:

```
// Author: Ted Obuchowicz and John Peach
// newDeleteValue.cpp
// example program illustrating use of
// new and delete operators

#include <iostream>

using namespace std;

int main() {

    int* integerPointer;

    /*  Request enough memory from the heap to hold an integer
        if the request is successful, new returns the address,
        if there is no more memory, new returns the value 0.
    */

    integerPointer = new int(55);   // initialize it to 55

    // test to make sure memory was allocated
    if ( integerPointer ) {
```

```
        // print out some garbage number since we did not initialize
        cout << *integerPointer << endl;

        // assign the value 10 to this memory location
        *integerPointer = 10;

        // Report address and value
        cout << "Address of newly allocated memory = " << integerPointer << endl;
        cout << "value stored at this address = " << *integerPointer << endl;

        // release the memory back to the heap
        // The pointer still exists, just the int has been released
        delete integerPointer;
    }

    return 0;
}
```

The program output is:

```
55
Address of newly allocated memory = 0x1737010
value stored at this address = 10
```

Rather than space to hold a single variable, new may be asked to dynamically allocate a contiguous sequence of memory locations (i.e. an array) this is done with

new <data_type> [size]

Here is a program which dynamically allocates an array of integers. The size of the array is specified by the user as input at run-time:

```
// Author: Ted Obuchowicz and John Peach
// dynamicArray.cpp
// example program illustrating use of dynamic allocation of
// memory to create an array of any size

#include <iostream>

using namespace std;

int main() {

    unsigned int size;

    int* pointer;   // this pointer will be set to the starting address
                    // of an array of integers. This array will be dynamically
                    // allocated at run time through a call to the new operator
                    // we will pass the requested size as to new

    cout << "Enter the size of the array you wish to create: ";
    cin >> size;
```

```
    pointer = new int[size];    // request space to hold size integers and assign
                                // starting address of this space to pointer

    for(int i = 0; i < size; i++) {
        *(pointer + i)  = i;    // assign some values to these integers
                                // same as doing: pointer[i] = i;
    }

    // print out the array
    for(int i = 0; i < size; i++) {
        cout << *pointer << " " << pointer <<  endl;
        pointer++;
    }

    cout << "Address of the last element plus one more (danger!): "
         << pointer << endl;

    pointer -= size;            // reset pointer back to start of array

    cout << "Address of the first element: " << pointer << endl;

    // free the memory
    delete [] pointer;

    return 0;
}
```

The program output is:

```
Enter the size of the array you wish to create: 4
0 0x1634010
1 0x1634014
2 0x1634018
3 0x163401c
Address of the last element plus one more (danger!): 0x1634020
Address of the first element: 0x1634010
```

A variation of the above program is to use new to dynamically allocate a two-dimensional array. The program uses the address translation mechanism discussed earlier:

```
// Author: Ted Obuchowicz and John Peach and John Peach
// dynamic2DArray.cpp
// example program illustrating use of

#include <iostream>

using namespace std;

int main() {
```

```
    int row;
    int column;

    /* This pointer will be set to the starting address
       of an array of integers. This array will be dynamically
       allocated at run time through a call to the new operator
       we will pass the requested size as to new
   */
    int* pointer;
    int* pointer_backup;

    cout << "Enter the number of rows: ";
    cin >> row;

    cout << "Enter the number of columns: ";
    cin >> column;

    // request space to hold a 2-d array rows x cols
    pointer = new int [row * column];

    for(int i = 0; i < row; i++) {
        for(int j = 0; j < column; j++) {
            // assign some values to these integers
            *(pointer + (i * column + j))  = i + j;
        }
    }

    // now print out the array
    for(int i = 0; i < row; i++) {
        for(int j = 0; j < column; j++) {

            cout <<   *(pointer + (i * column + j))  << " ";
            if (j == column - 1 ) {
                cout << endl;
            }
        }
    }

    // free the memory
    delete [] pointer;

    return 0;
}
```

Here is the program output:

```
Enter the number of rows: 4
Enter the number of columns: 6
0 1 2 3 4 5
1 2 3 4 5 6
2 3 4 5 6 7
3 4 5 6 7 8
```

Here is another example of the use of the new operator together with pointers. It is a C++ function which receives a char* and will return another char*. The function will dynamically allocate sufficient memory to hold the contents string pointed to by the passed parameter with ALL VOWELS (a,e,i,o,u) removed. The function will return a pointer to the start of the dynamically allocated memory. We will make use of the strlen function found in the library <cstring> within your function.

```
// Author: Ted Obuchowicz and John Peach and John Peach
// removeVowel.cpp
// example program illustrating use of

#include <iostream>
#include <cstring>

using namespace std;

char* removeVowel(char * str);

int main() {

    char* noVowel;          // String with no vowels

    // Test strings
    char s1[] = "xcvbnmhjk";
    char s2[] = "";
    char s3[] = "aeiou";
    char s4[] = "hello";
    char s5[] = "good";

    // Remove the vowels, print the result and free the memory.
    // Test that a string was returned
    noVowel = removeVowel(s1);
    if ( noVowel ) {
        cout << s1 << " : " << noVowel << endl;
        delete noVowel;
    }

    noVowel = removeVowel(s2);
    if ( noVowel ) {
        cout << s2 << " : " << noVowel << endl;
        delete noVowel;
    }

    noVowel = removeVowel(s3);
    if ( noVowel ) {
```

```
        cout << s3 << " : " << noVowel << endl;
        delete noVowel;
    }

    noVowel = removeVowel(s4);
    if ( noVowel ) {
        cout << s4 << " : " << noVowel << endl;
        delete noVowel;
    }

    noVowel = removeVowel(s5);
    if ( noVowel ) {
        cout << s5 << " : " << noVowel << endl;
        delete noVowel;
    }

    return 0;
}

char* removeVowel(char* str) {

    int length = strlen(str);
    length++;                           // account for end-of-string character
    char* noVowel = new char[length];   // dynamically allocate memory
    char* currentPosition = noVowel;    // current position in noVowel

    // Test to make sure that memory was allocated
    if (noVowel) {


        // Copy all the non-vowel characters
        while(*str) {
            if ( *str != 'a' && *str != 'e' && *str != 'i' &&
                 *str != 'o' && *str != 'u' ) {

                // copy non-vowel
                *currentPosition = *str;

                // move the pointer to the next memory location
                currentPosition++;

            }

            // advance the pointer to the next character
            str++;
        }

        // add the end of string character to the end of the new string
        *currentPosition =  '\0';
    }
    return noVowel;
}
```

The program output is:

```
xcvbnmhjk : xcvbnmhjk
 :
aeiou :
hello : hll
good : gd
```