Pointers and Dynamic Memory Allocation

1. The Pointer

In the work on sorting, we avoided problems with variations in the types being sorted by using (integer) indexes to indirectly reference the data. C++ provides a standard method of this indirection. A **pointer** data type stores the address of a data item. We declare a pointer variable using

```
type* var;
```

where *type* is any declared data type, either basic or user-defined; and *var* is a valid variable name, including a possible array specification. Although the * is usually written next to the type, it can also be written adjacent to the variable name, or even separated from both. Thus

```
int* ptr;          // ptr is a pointer to an integer
int *ptr;          // *ptr is an integer (see below)
```

and

```
int * ptr;
```

are all equivalent. The * actually attaches to the variable name. Each of the above declare a variable called ptr which can be used to store a pointer to an int. If we want to declare two pointer variables, we write

```
int *ptr1, *ptr2;
```

and not

```
int* ptr1, var2;  // declares an int called var2
```

because of the above-mentioned attachment of the * to the variable name. It is in fact preferable to declare pointer types and non-pointer types in separate statements to avoid the easily-created mis-declaration.

What sort of value do we assign to a pointer variable? The address of a data value.

```
int* ptr;
int value;

ptr = &value;
```

The operator & means 'address of'. Thus the last assignment says 'store the address of the variable value in the variable ptr'.

Thus, the value stored in a pointer variable is where a value can be found. Initially a pointer variable has an undefined value – it points at no particular value. If we wish to initialise a pointer variable to a value to indicate that it does not yet point to anything, we can set it to 0. Often the constant value NULL (declared as 0 in the header cstdlib) is used instead of 0 to indicate the address 0. This is not common in C++ (but very common in C).

How do we refer to the value pointed to?

We use the operator *, so that *ptr means the value stored at the address stored in ptr.
For example,

```
int x=1, y=2, z;
int *p;

p = &x;      // p now points at x
z = *p;      // set z to the value located where p points to ...  1
p = &y;      // p now points at y
z += *p;     // add to z the value where p now points to
cout << z;
```

would output the value 3.

Referring to the value pointed at by a pointer is called **dereferencing** the pointer. Note that we cannot dereference a 0 pointer as it does not point at anywhere (legal).

We can now replace our array of indexes in our sort by an array of pointers.
So

```
DataType X[SIZE];
DataType* Index[SIZE];

for (i=0;i<n-1;i++)
      Index[i] = &X[i];
```

and we must now write the two functions so that

```
int CompareData(int IV1, int IV2)
{
      // compares *Index[IV1] to *Index[IV2]
}
```

And

```
void ExchangeData(int IV1, int IV2)
{
      // exchanges Index[IV1] and Index[IV2]
}
```

Note that we still refer to the pointers by their position in the array of pointers. This ensures that the functions are generic – involve integers only, not the datatype being sorted. This is because, even though a pointer is just an address, pointers to different types are different pointer types.

The advantage of the pointer over the index, however, is that the dataset to be sorted need not be in an array. For example

```
double SalesSpr, SalesSum, SalesAut, SalesWin;
double *SalesIndex[4] = {&SalesSpr,&SalesSum,&SalesAut,&SalesWin};
```

allows us to sort the four values without placing them in an array.

1.1 Declaring a pointer type

As with any type, either basic or user-defined, we can alias such a type to a new type. This is especially desirable for pointers. For example we could declare

```
typedef DataType* DataPtr;
```

and then use the type `DataPtr` to declare further pointers, as in

```
DataPtr Index[10];
```

This is vital for passing a pointer type to a function, especially as pass-by-reference, which we will see later.

2. Pointers and Argument Passing

The concept of pass-by-reference is C++'s method of transferring information from a function. The mechanics of this procedure is that the address of the variable is placed on the stack. The compiler ensures that all references to a formal argument passed by reference are indirect. That is, the formal argument is an alias of the actual argument.

C does not have this construct. Instead, when the address of a variable is needed by a function, so that its value could be modified, the address has to be explicitly transferred.

Consider the `Swap` function for exchanging two integer values.

In C++, we have

```
void Swap(int& X1, int& X2)
{
      int temp;

      temp = X1;
      X1 = X2;
      X2 = temp;
}
```

and would be called, for example, as

```
      Swap(A,B);
```

This means the addresses of the two actual arguments `A` and `B` are placed on the stack and used as the addresses of the two formal arguments `X1` and `X2`.

In C, both the call and the function itself must explicitly specify the pointers and the values located where they point. So C requires the `&` operator to specify that the address is to be passed, as in

```
      Swap(&A,&B);
```

And the function must use the `*` form of referring to the value at the place pointed to.

```
void Swap(int *X1, int *X2)
{
      int temp;

      temp = *X1;
      *X1 = *X2;
      *X2 = temp;
}
```

Thus we have explicit loading of the addresses onto the stack (`&A`), the referencing of the formal arguments as pointers in the function header (`int *`), and the reference to the value stored indirectly by those pointers (`*X1`). Apart from the syntax, the actual mechanics are (almost) identical to the C++ pass by reference. The two differing notations can both be used in C++, but the prototype for one form cannot appear with the use of the other.

We could also have declared

```
      typedef int* intPtr;
```

and then used

```
      void Swap(intPtr,IntPtr);
```

3 Arrays and Pointer Equivalence

If a function argument is an array type, passing it an address indicates the location of element 0 of the array formal argument. For example, the function

```
int SumArray(int arr[], int n)
{
      int i, sum=0;

      for (i=0;i<n;i++)
            sum += arr[i];
      return sum;
}
```

can sum the array

```
      int A[10] = {1,2,3,4,5,6,7,8,9,10};
```

as

```
SumArray(A,10)
```
or
```
SumArray(&A[0],10)
```

That is, the array name `A` is equivalent to the address of `A[0]`. The same function can sum the last 9 elements as

```
SumArray(&A[1],9)
```

That is, an array name and an address *seem* equivalent.

In fact a pointer type can also be referenced just like an array. For example, consider

```
int A[10];
int *B = A;        // note how we are giving B a value, not *B
```

This specifies that the pointer `B` is given the value of the address of the array `A`. If we refer to `*B`, we are referring to the contents of `A[0]`. But we can also say `B[0]` to refer to the same value!! And `B[1]`, `B[2]`, and so on correspond to `A[1]`, `A[2]`, ...
Thus, all the problems inherent in overrunning array dimensions apply to dangerous usage of pointer references.

4. Pointer Arithmetic

When we specify a subscripted array element such as `A[1]`, we are in fact asking C++ to carry out pointer arithmetic. So

`A[1]` has the same meaning as `*(A+1)`

That is, `A[1]` refers to the contents of the location found by adding `1` to the address `A`. Well, not quite `1`, but one memory location capable of holding one item of the type of the array `A`. In the case of a C++ int that would probably be 4 (bytes). But we do not say we want to add 4. We add 1. The compiler determines how many bytes one of that particular type occupies.

C++ provides an operator called `sizeof` which yields the information needed to understand the concept of pointer arithmetic. This operator usually appears as if it were a function, as

```
sizeof(int)
```

which provides the size (in bytes) of an `int`, although the parentheses are not needed in this form. Similarly `sizeof(float)`, `sizeof(char)`, in fact

`sizeof(`*type*`)` or `sizeof` *type*

for any *type*, including user-defined types, has the value of the storage required. It can also be used to tell us the memory usage of arrays. For example,

```
const char message[] = "How long is this?";
```

would result in

```
sizeof(message)
```

yielding a value of 18. Thus

`sizeof(`*var*`)`

results in the memory usage of the variable *var*. In this form the parentheses are required.

Note, however, that the result of asking for the size of a pointer variable is the memory required by the pointer itself, not what it points at.

Thus, for the two variables

```
char C[10];
char*D = C;

sizeof(C)    yields  10
```
while
```
sizeof(D)    yields  4       (the size of a char*)
```

So

`A[1]` is located `sizeof(int)` bytes after `A[0]`

and

`A[i]` is the same as `*(A+i)` which is `i*sizeof(int)` bytes after `A[0]`.

The only valid arithmetic operations involving pointer types are increment (`++`), decrement (`--`), addition of an integer (`+`, `+=`) and subtraction of an integer (`-`, `-=`), or one pointer may be subtracted from another.

5. Differences between Pointers and Arrays

Recall that one form of array – the character array – sometimes acts differently in C++. If we had

```
int A[10] = {0,1,2,3,4,5,6,7,8,9};
char B[20] ="0123456789";

cout << A;         // outputs the address of the array
cout << B;         // outputs the string contents
```

That is, because a character array is regularly used to store C-strings, C++ `iostream` library treats character array names differently to other type array names. (How do we output the address of B?)

One of the common constructs seen in C programs is the use of `char` pointers where in C++ we normally use `char` arrays as in

```
char *msg = "An error has occurred.\n";
```

and then use `msg` just like a character array.

This can also be used in C++ programs, but it should be remembered that a character pointer and a character array are not quite the same. The above construct allocates a memory location (capable of storing an address) to the variable `msg`, and then sets its value to the address of the C-string `"An error has occurred.\n"`.

The declaration

```
char msg2[] = "A booboo has occurred.\n";
```

allocates 24 bytes for the variable `msg2`, storing the given string in those 24 bytes. Thus the pointer version requires more memory.

Note also that `sizeof(msg2)` would yield 24, while `sizeof(msg)` would return 4 – the (usual) memory required for a character pointer.

As mentioned earlier, pointer variables can be treated like arrays. Thus, the letter `'h'` in the first C-string can be referred to by

```
*(msg+9)          or          msg[9]
```

as also can

```
msg2[9]      be referred to as      *(msg2+9)
```

The critical differences concerns what we can do with the two C-strings involved.

We can write

```
msg = msg2;
```

and even

```
msg = "A different message";
```

so that the pointer `msg` now points at a different C-string. Only a change to the value of the pointer is produced.

Many a novice programmer is surprised that we cannot say

```
msg2 = "A different message";
```

but we may pass `msg2` to a function such as

```
void func(char m[])
{
      m = "def";
}
```

where it seems to be valid. This is because the formal argument type `char[]` is equivalent (to the compiler) to `char*` for which such assignment is valid.

Neither the value of the C-string, nor the value of the pointer variable declared as

```
char* msg3;
```

are altered upon return from the call

```
func(msg3);
```

as we are changing the value of a local copy of the pointer variable. Assigning a value to, say `m[2]` in the function would alter that character in both a `char` array or a C-string pointed to by a `char` pointer. If we wanted to change the value of the pointer (that is, where it points), we would have to pass the pointer by reference, either using `char**`, `char*&`, or by defining a pointer type and using the `&` notation.

Note that we should not change any individual character of the pointer version, as we set the pointer to point to a constant C-string.

In fact, we should write

```
char* msg = "abc";
```

as

```
const char* msg = "abc";
```

if we really want to indicate that 'msg is a pointer to a constant C-string'. Again the `*` goes with the variable name. This means that the C++ compiler will watch out for any attempt to change the contents of the C-string pointed at by `msg`, but will allow us to change where `msg` points. If we want to be able to change the C-string contents, be careful if it is pointing at a C-string constant.

This is different to declaring that the value of the pointer itself is constant. For that we place the keyword `const` after the `*` as in

```
char* const msg = "abc";
```

where we cannot change the value of `msg` itself – `msg` is now a `const` pointer to a `char`. This is in fact how a `char[]` declaration is treated – like a pointer which cannot change its value.

This variation in the effect of the `const` keyword on pointers can be very confusing. Compilers often give error messages regarding the passing of pointers to `const`s or `const` pointers where the function expects the reverse. The use of `typedef` to create pointer types can reduce this problem.

6. Pointers and `structs`

Suppose we have a user-defined type

```
struct Stype
{
      int member1;
      float member2;
      char member3[5];
};
```

and then declare a pointer to such a type

```
Stype* SS;
```

or even better, declare a pointer type

```
typedef Stype* Sptr;
```

and

```
Sptr SS;
```

Note that, until we specify a value for `SS`, the pointer points at nothing in particular. So let's declare

```
Stype S;
```

and set

```
SS = &S;
```

To reference the members of the variable `S`, we use the form

```
S.member1
S.member2
```

and `S.member3`

down to

```
S.member3[3]
```

To reference these indirectly via the pointer `SS`, we can use

```
(*SS).member1
```

and similar. However, another more useful notation is

```
SS->member1
```

where the operator `->` is interpreted as the following component of what `Sptr` points at.

7. Pointers as Return Values

We stated in the previous section that the use of a pointer requires the variable to be given the value of another variable's address. This address may be for a global variable, an automatic variable or a local variable. It can be passed to functions, including by reference, and also be the return value of a function.

We have already seen library functions which return pointer values. For example, many functions in the `cstring` library return such values.

The function

```
char* strchr(const char*,int);
```

returns a pointer to the character, passed as the second argument, as it appears in the first argument. Thus

```
strchr("abcdef",'d')
```

would have the value of the address of the `'d'` in the first C-string.

So

```
char* ans;
ans = strchr("abcdef",'d');
cout << ans << endl;
```

results in the output

```
def
```

Thus the function returns an address located within the memory locations passed to it. But what if we want to point at memory not passed by argument? Consider the following function which aims to return the name of a month, given the month number.

```
char* MonthName(int m)
{
      static char monthname[12][4] ={"Jan","Feb","Mar","Apr","May",
                        "Jun","Jul","Aug","Sep","Oct","Nov","Dec"};

      return monthname[m];
}
```

An alternative is

```
void MonthName(int m, char** n)
{
      static char monthname....

      *n = monthname[m];
}
```

Why is the `static` keyword used? Obviously, the names of the months will be the same each time the function is called, so having them set once is efficient. But the best reason is that, without the `static`, there are problems. Then, the array `monthname` is an automatic variable, created when the function is entered and destroyed when leaving. That is, the memory allocated for the array is then available to other future automatic variables. Thus the pointer variable which is allocated this (address) value will point at data which may be altered by later actions. There are two ways of remedying such a situation.

The first is to ensure that addresses returned as the value of a function point at memory which is fixed at compilation time – by referencing either global variables or static variables. Such variables are placed in the static data area at compilation time. In the above case, the keyword `static` inserted before the declaration of `monthname` serves this purpose.

The second approach – giving a pointer the value of memory not currently allocated to a known variable – leads to the solution of a problem referred to at the beginning of this subject. Can we get more space for variables after a program has been compiled? Yes, as we shall see soon.

7.1 References as return values

Earlier we indicated the similarity between pointers and references. Both are implemented by addresses, with pointers needing dereferencing before use. We can also return such addresses as references, without the need for dereferencing. Consider the following example involving the `struct` called `Stype` appearing on the previous page.

Suppose we have a function which is to return an `Stype` value. There are four ways of doing this.

Firstly, we could have

```
void func1(Stype& v)
{
      // code setting members of v
}
```

with a call such as

```
func1(s);
```

to set `s` to the resulting value. The function directly refers to the variable `s` as the formal argument `v` is merely an alias to `s`.

Or we could have

```
Stype func2()
{
      Stype v;
      // code setting members of v
      return v;
}
and
      s = func2();
```

This involves a local automatic variable `v` which is set, then the whole structure is copied to some location for the return from the function, and that value is then copied to `s`. If the structure is large, there is a lot of copying, which was avoided by the alias.

Or we could use pointers.

```
Stype* func3()
{
      static Stype v;
      // code setting members of v
      return &v;
}
with
      s = *func3();
```

This time the local variable is not automatic, and only the address of that variable is returned. The copy of its data to `s` is still performed once the address is dereferenced.

Or, we could return a reference – an alias – to the static local variable.

```
Stype& func4()
{
      static Stype v;
      // code setting members of v
      return v;
}
and
      s = func4();
```

Note that the last two are similar, as would be expected.

There are several differences between returning a reference value as opposed to a pointer. One very important difference is that, as the reference is an alias, it must refer to an actual memory location (a variable) and hence 0 cannot be returned in this case. If you want to return 0, you must return a pointer value.

8. Dynamic Memory Allocation

C++ provides a means of requesting memory from an area called the **heap**, usually located above the static area. To request memory for a particular datatype, say an `int`, we can specify

```
      var = new int;
```

where the operator gets the required number of bytes and returns the **address** of the memory so allocated. This is then stored in the pointer variable *var*. If the memory is not available, there are two possible outcomes.

First, the standard for C++ specifies that an **exception** occurs, causing a termination of the program (unless an exception handler is provided by the programmer – not covered in this course). Alternatively, the address returned could be set to 0. In this case, the returned value should be checked before the pointer is dereferenced. Although different compilers provide either or both of the above, we will assume that the latter is the result. For example,

```
float* f;
f = new float;
if (f == 0)
      // don't go on using f
```

Once the memory is allocated, the pointer can be used just like any other pointer. That is, this new memory location can be used to store data – albeit only by indirection using the pointer.

The stronger effect of this new operator is its ability to provide access to successive memory locations, that is, arrays. For example, to request 20 bytes to store C-strings in we could say

```
char* str;
str = new char[20];
```

and `str` would now point at 20 memory locations. We could now use those 20 locations, say by

```
strcpy(str,"A new message");
```

or

```
str[0] = 'a';
```

That is, `str` can now be used as if it were a normally-declared array, with a reminder about the differences between an array and a pointer listed above.

For example

```
str = 0;
```

would set `str` back to pointing at nothing, and we would lose the whereabouts of the 20 bytes previously allocated.

Our month name function can now be rewritten as

```
char* MonthName(int m)
{
      char monthname[12][4] ={"Jan","Feb","Mar","Apr","May","Jun",
                              "Jul","Aug","Sep","Oct","Nov","Dec"};
      char* n;

      n = new char[4];
      strcpy(n,monthname[m]);
      return n;
}
```

Note how it is now more complicated in that we have to use `strcpy` to actually change the contents of where `m` points at rather than changing where `m` points.

That is,

```
n = monthname[m];
```

changes the value of the pointer `n` to the address of the requisite month name, while

```
strcpy(n,monthname[m]);
```

changes the contents of the four bytes, newly allocated, and whose location is stored in `n`, to the same bytes as is contained at `monthname[m]`.

Note that it would still be preferable to make the array of names in the function a `static const` array just to save execution time in creating the automatic variable.

Way back we indicated a desire to be able to set up a database of a size controlled by input values. Here's how we can now do it.

```
struct Video
{
      char Title[MAXTITLE+1];
      long BarCode;
      short RunTime;     // in minutes
      char Category[MAXCATEGORY+1];  // comedy, adventure,
                                                  // action, drama, etc.
      short NumCopies;
};

Video* DBase;
int NumRecords;

cout << "How big is existing database? ";
cin >> NumRecords;
MAXVIDEO = NumRecords + 10;          // allows for 10 additions
DBase = new Video[MAXVIDEO];
if (DBase == 0)
{
      cout << "No heap space for DBase.\n";
      return 1;
}
. . .
. . .
```

Note, of course, that `MAXVIDEO` can no longer be a `const`.

8.1 Can we get rid of this dynamic memory?

One of the nice features of the memory allocated by the operator `new` is that, once its use is complete, it can be returned for future use by other `new` actions.

The operator `delete` returns the piece of dynamic memory referred to by the value of the pointer `ptr` by

```
delete ptr;
```

The value of `ptr` may or may not changed by this operation. It may still point at the memory previously allocated. But that memory is now free to be used by other new operations. Such a pointer (referring to release memory) is called a **dangling pointer**. It is advisable to set the pointer's value back to 0 once memory is returned.

The operator `delete` also can return dynamic arrays back to the useable memory using the same notation.

Thus

```
delete [] DBase;
```

will free up the memory allocated in the last section. We will see later how important it is to use this form when returning memory allocated as an array.

You should get used to deleting dynamic allocations once their usage is complete. This is especially true of memory allocated within functions. Consider

```
void func()
{
      int* ptr;

      ptr = new int[100];
      . . .
}
```

Unless the location of the memory allocated via the `new` in the function is transmitted to other functions, the 100 pieces of memory are no longer accessible once the function is exited, and are wasted if they are not deleted. This can lead to a problem called a **memory leak**, in which we can ultimately run out of heap space.

8.2 Dynamic memory allocation overhead

How does `delete` know how much memory is to be freed up for future usage, when all it receives is the address of that memory? Because each new operation does not just allocate the memory for the datatype required. There are several bytes allocated to store the number of bytes associated with the dynamic memory. That is why it is impossible, for example, to delete part of a dynamically allocated array. Thus, asking for one byte to store a character actually allocates more than that.

There is also waste memory whenever a datatype occupies an odd number of bytes, in which case bytes are often unused so that the addresses of any data item has an even address.

So

```
char *c1, *c2;

c1 = new char;
c2 = new char;
```

would yield addresses more than one byte apart. How much overhead is used is, of course, implementation-dependent.

9. Another Use for Pointers and Dynamic Memory Allocation

Although the above uses of pointers and dynamic memory allocation seem very powerful, their strengths lie in allowing for new data structures other than the traditional scalars and arrays. Consider the following problem.

A pizza restaurant allocates tables on a first-come, first-served basis. Customers come to the counter, give their names, how many in their party, and take a seat. The receptionist keeps a list of customers in the order they arrive. When a table becomes free the person at the top of the list is allocated a table and then crossed off the list. (We'll see later that this list is in fact called a queue.) How can we computerise this process?

Suppose we create a `struct` with the following contents:

```
struct Cust
{
      char name[30];
      int partysize;
      Cust *next;
};

typedef Cust* CustPtr;
```

Note that an instance of the struct `Cust` contains the customer's name, number in the party and a pointer to the next customer in the list. We then create one single pointer to the first person on the list

```
CustPtr Head;
```

Initially the list is empty and hence the pointer's value is (set to be) 0. What processes are needed for this list? We need to be able to add a customer to the end of the list, and get the first one off the top of the list. We thus could use three functions: `Initialise`, `Add` and `Remove`. Here are algorithms for each.

`Initialise`

zero the head of the list

`Add`

create a new customer record
link it to the tail of the list

Remove
	if there is a customer
		get the customer's name and party size
		make the head of the list point at the next customer
		cleanup the memory used by the customer
		return the name and party size

So

```
void Initialise(CustPtr& Head)
{
	Head = 0;
}

void Add(char name[], int psize, CustPtr& Head)
{
	CustPtr tmp, curr;

//	create a new customer and store the information
	tmp = new Cust;
	strcpy(tmp->name,name);
	tmp->partysize = psize;
	tmp->next = 0;

//	add to the end of the list if the list is not empty
	if (Head != 0)
	{
		curr = Head;
		while (curr->next)		// is this the end?
			curr = curr->next;
		curr->next = tmp;
	}
	else
		Head = tmp;
}

int Remove(char name[], int& psize, CustPtr& Head)
{
	CustPtr tmp;

	if (Head == 0)
		return 0;
	strcpy(name,Head->name);
	psize = Head->partysize;
	tmp = Head; // make a copy of the pointer to current head
	Head = Head->next;		// change the head
	delete tmp; // delete the old head's memory
	return 1;
}
```

With these three functions, we can create an arbitrarily long list. That is, we have the ability to maintain a database without a size limit set at compile time. This was one of the concerns in our first case study.

Note that our access to the list is via the pointer to the head of the list. Entries in the list are accessed by following the links from one entry to the next. (That's why the data structure is called a linked list.) One of the nice concepts in he earlier case study was that the way the data was stored was hidden from the user of the database. In the above situation we could satisfy that same need by making `Head` local to the three functions and not available to the calling function. We could then remove `Head` from the argument list of the functions. This ensures that the user cannot access the structure of the data, protecting the links from accidentally being interfered with.

But what if we want two lists of customers? We then need differing `Head`s, each passed to the functions when needed. But the protection is gone. Wouldn't it be nice if we could have any number of lists, but with the protection?