Pointers and Dynamic Variables

Motivation

Sometimes when you write a program you do not know at compile-time how large an array needs to be, or how many instances of a class you might need because it depends what happens when the program is running. For example, imagine a program that asks a user interactively to enter some data that must be stored in an array, but the program does not know how many items will be entered. If the array is declared of fixed size at compile time, either it can be too small or too large, depending on how much data is entered. C and C++ have (different) instructions for solving this problem. We look at C++'s solutions here.

Pointers

A pointer variable contains a memory address as its value. Normally, a variable contains a specific value; a pointer variable, in contrast, contains the memory address of another object. We say that a pointer variable, or pointer for short, ***points to*** the object whose address it stores.

Pointers are declared to point to a specific type of object and can point to objects of that type only. The asterisk `*` is used to declare pointer types.

```
int*    intptr;    // intptr can store the address of an integer variable
double* doubleptr; // doubleptr can store the address of a double variable
char*   charptr;   // charptr can store the address of a character
```

We say that `intptr` is "a pointer to int" and that `doubleptr` is "a pointer to double." Although we could also say that `charptr` is a pointer to char, we say instead that `charptr` is a string. Pointers to `char` are special.

The * is a *pointer declarator operator*. In a declaration, it turns the name to its right into a pointer to the type to the left. It is a *unary, right associative* operator. It does not matter whether you leave space to the right or left of it, as demonstrated below. These three are equivalent:

```
int* intptr;
int * intptr;
int *intptr;
```

In general, if `T` is some existing type, `T*` is the type of a variable that can store the address of objects of type `T`, whether it is a simple scalar variable such as `int`, `double`, or `char`, or a more structured variable such as a structure or a class:

```
struct Point { int x, int y };
Point* Pointptr;
```

You can even write a declaration like this:

```
struct Spline
{
    int x;
    int y;
    Spline *nextpoint;
};
```

which we will not even attempt to explain here (since it is a topic for the sequel to this class). What do you think it defines?

Pointers are even more general than this. The following are all valid declarations that declare pointers to many different kinds of things.

```
int  *intp;        // pointer to an integer memory cell
char *pc;          // pointer to char, also known as a C string
int * N[10];       // N is array of pointers to int (NOT POINTER TO ARRAY OF INTS)
int (*B)[10];      // B is a pointer to an array of 10 ints
int* f(int x);     // f is a function returning a pointer to an int
int  (*pf)(int x); // pf is a pointer to a function returning an int
int* (*pg)(int x); // pg is a pointer to a function returning a pointer to an int
```

Notice the difference in how an array of `int*` is declared versus a pointer to an array of `int`s, and how a pointer to a function is declared versus a function that returns pointers to other things.

The `typedef` statement is useful for declaring types that are pointers to things:

```
typedef int* intptr;
intptr  intp, intq;       // p and q are both pointers to int
intptr  func( intptr np); // func returns a pointer to an int
```

This is even more interesting:

```
typedef bool (*MessageFunc) (char* mssge);
// This defines a MessageFunc to be a pointer to a function with a
// null-terminated string as an argument, returning a bool.
```

A `MessageFunc` is a pointer to a function. This makes it possible to pass one function to another, as in the following `handle_error()` function, which calls `f()` with a message and either exits the program or not depending on the boolean value returned by `f()`.

```
void handle_error( MessageFunc f, char* err_mssge)
{
    if ( f(err_mssge) )
        exit(1);
}
```

The address-of operator, `&`, can be used to get the memory address of an object. It is the same symbol as the reference operator, unfortunately, and might be a source of confusion. However, if you remember this rule, you will not be confused:

If the symbol "`&`" appears ***in a declaration***, it is declaring that the variable to its right is a reference variable, whereas if it appears ***in an expression that is not a declaration***, it is the address-of operator and its value is the address of the variable to its right.

```
int x = 200;
int y = 100;
int &ref = x;  // a declaration, so & is the reference operator
int *p   = &x; // appears on right of =, so it is in an
               // expression and is address-of operator; p stores address of x
p = &y;        // address-of operator; now p stores the address of y
int &z = y;    // reference operator; z is an alias for y
```

Dereferencing Pointers

Accessing the object to which a pointer points is called ***dereferencing the pointer***. The "*" operator, when it appears in an expression, dereferences the pointer to its right. Again we have a symbol that appears to have two meanings, and I will shortly give you a way to remain clear about it.

Once a pointer is dereferenced, it can be used as a normal variable. Using the declarations and instructions above,

```
p = &y;
cout << *p;         // p contains address of y, which stores 100,
                    // so 100 is printed. p has been dereferenced.
cin >> *p;          // p is dereferenced before the input operation.
                    // *p is the variable y, so this stores the input into y.
*p = *p + *p;       // just like y = y + y
*p = *p * *p;       // just like y = y * y
(*p)++;          // * has lower  precedence than the post-increment ++
                    // so this increments *p which is y so y =  101
int  a[100] = {0}; // a is an array of 100 ints, initially all 0
int *q = &a[0]; // q is the address of a[0]
q++;                // q is address of a[1]
*q  = 2;         // now a[1] contains  2
cout << *q++;       // increment q (so q is address of a[2]) and print a[2]
```

The dereference operator and the address-of operator have higher precedence than any arithmetic operators, except the increment and decrement operators, `++` and `--`.

If you think about a declaration such as

```
int * p;
```

and treat the `*` as a dereference operator, then it says that `*p` is of type `int`, or in other words, `p` is a variable that, when dereferenced, produces an `int`. If you remember this, then you should not be confused about this operator.

Relationship Between Pointers And Arrays

An array name is a constant pointer; i.e., it is pointer that cannot be assigned a new value. It always contains the address of its first element.

```
int list[5] = {2,4,6,8,10};  // list is the address of list[0]
cout << *list << endl;       // prints 2 because *list is the contents of list[0]

int * const aPtr = list;     // aPtr has the same address as list
                             // &list[0] is stored in aPtr.
aPtr[2] += 10;               // adds 10 to  list[2]
```

The fact that array names are essentially constant pointers will play an important role in being able to create arrays of arbitrary size while the program is running, rather than by declaring them statically (at compile time).

This is all we need to know for now in order to be able to talk more about C strings and the C string library.

Classes, Structures, and Pointers

There is a special notation that is used when a dereference operator is combined with a member access operator. Suppose we have the following declarations:

```
struct Point
{
    double x;
    double y;
};

Point * pptr;
```

and we dynamically allocate a `Point` variable using `pptr` :

```
pptr = new Point;
```

Then to access the `x` member of the `Point` pointed to by `pptr`, we could write either of these:

```
(*pptr).x
pptr->x
```

They are equivalent. The ***arrow operator*** `->` dereferences the pointer to its left and accesses the member of the structure or class to which it points to its right, so `pptr->x` is a shortcut for `(*pptr).x`.

The same thing is true for classes. Given

```
class CPoint
{
public:
    void print(ostream & out) const;
private:
    double x;
    double y;
};

CPoint * cpptr;
```

we would write

```
cpptr->print(fout);
```

to call the `print()` member function of the class on the dynamic object pointed to by `cpptr`.

The `this` Pointer

C++ defines a special pointer variable that is part of every class, called `this`. The `this` pointer points to the object on which a function is being called. In essence it points to the object itself. A few examples will make this clearer.

```
class Simple
{
    public:
        Simple()
        Simple * addr()
        void set(double x )
        double val()
    private:
        double x;
};

Simple*  Simple::addr()
{
    return this;  // return a pointer to the object itself
}

void Simple::set( double x )
{
    this->x = x;
}
```

In this first example, the `addr()` member function returns a pointer to the object on which it is called, so we can use it as follows:

```
Simple s1;
Simple s2;

cout << "The address of s1 is " << s1.addr() << endl;
cout << "The address of s2 is " << s2.addr() << endl;
```

and the output might look like

```
The address of s1 is 0xbfc03718
The address of s2 is 0xbfc03710
```

The `set()` member function is poorly designed because the parameter has the same name as the private variable. Without a `this` pointer we could not write its definition. But with it, we can write the assignment

```
this->x = x;
```

so that we can tell that the left-hand side is the private member of the object and the right-hand side is the parameter.

Dynamic Memory Allocation

Variables declared with block scope are created on what we call the ***runtime stack***. They get created when the program's execution enters the block and they get destroyed when it leaves that block. Global variables, i.e., variables with file scope, get created when the program first starts up and they get destroyed when the program terminates. You have no control over the lifetimes of either of these types of variables.

C++ provides an operator named `new` that lets you create a new, nameless variable while the program is running. The simplest use of it looks like this:

```
int * p;
p = new int;
```

The `new` operator has a single argument which is the name of an already defined type. When it is executed, it creates an object of this type. The value of the expression

```
new int
```

is the address of the new object of type `int`. This can be assigned to a pointer of type `int`. We can say informally that `new` "returns" the address of an `int`, even though this is not an accurate statement, because `new` is not a function and it has no return value. A variable that si created by the new operator is called a ***dynamically allocated variable***, or just a ***dynamic variable***, because it is created at runtime by the program.

More generally

```
T * p = new T;
```

creates a new dynamic variable of type `T` and assigns its address to the `T` pointer `p`.

If the type is the name of a class, the constructor of that class is invoked. For example,

```
Class MyClass
{
public:
    MyClass();
    MyClass( int x, int y);
    // other stuff here
};
MyClass *cptr = new MyClass;
```

creates an instance of `MyClass`, causing the default constructor to be invoked, and assigns its address to `cptr` and

```
MyClass *c2ptr = newMyClass(4,5);
```

creates a second instance of `MyClass`, invoking the non-default constructor.

You can also use the initializer notation with elementary types, like this:

```
int * p = new int (100);
```

which creates a new variable of type `int` with initial value 100.

More About Memory Management

Block scope variables "live on the stack" and file scope variables are allocated in a special part of your program's memory when the program starts up. Variables created with the `new` operator come from a part of the memory called the ***heap*** or ***freestore***. The heap is of finite size. If you allocate too much of it the program will terminate. It is pretty big and most programs that are written properly do not run out of heap space. Programs that have memory-related errors may run out of heap space. This program will use up its heap store on a 32-bit machine:

```
int main()
{
    for ( int i = 0; i < 100000000000; i++ )
        int *p = new int;
    return 0;
}
```

because it asks for 400 billion bytes of heap, and a program's heap on a 32-bit machine is not that large. When `new` cannot allocate memory, it causes the program to exit[1]. With older C++ compilers, `new` returns a `NULL` pointer instead of making the program terminate.

It is very important that you assign the address of the new variable to a pointer. If you do not, your program will have created a chunk of memory that is completely inaccessible. When it does this, it is just wasting the machine's memory. That memory cannot be reused until the program terminates. If you reassign the value of a pointer to a new dynamic variable you also create unusable memory.

```
int *p;
p = new int (10);
p = new int (20);
```

The first dynamic variable with value 10 cannot be accessed after `p` is assigned the address of the second one. It is an example of a ***memory leak***.

To enable you to return memory that you no longer need, C++ has a `delete` operator. The `delete` operator has a single argument – the pointer whose memory is to be deleted. It deletes the dynamic variable pointed to by that pointer:

```
delete p;
```

deletes the dynamic variable pointed to by `p`.

Since `p` contains just an address, how does the runtime system know how many bytes of memory should be deleted? Remember that pointers are declared to point to specific types of objects. If `p` points to a 4 byte `int`, 4 bytes starting at `p` will be deleted. If it points to an 8 byte `double`, 8 bytes will be deleted.

If you do something like this:

```
int x;
int *p = &x;
delete p;
```

your program will crash, because `x` is a block scope variable, not a dynamic variable, and the machine will be very unhappy about your trying to delete what `p` points to.

This type of code is also a major problem:

```
int *p1, *p2;
p1 = new int(10);
p2 = p1;
// do stuff ...
delete p1;
cout << *p2 ; // get junk!
```

When the memory pointed to by `p1` is deleted, since `p2` also points to it, it now points to junk. The address it contains is meaningless, and trying to dereference it is equally meaningless. We say that p2 is a ***dangling pointer***. Dangling pointers are a common programming error.

[1] It throws an exception that is unhandled unless you specifically handled it. Exception handling is covered in CSci 235.

Dynamic Arrays

A ***dynamic array*** is an array that is created on the heap using the `new` operator. The advantage of a dynamic array over a static array is that its size can be determined at runtime. The syntax is slightly different when creating dynamic arrays:

```
int * q;
q = new int[100]; // 100 ints are allocated and q points to the start of the array
```

The general form of a statement to dynamically allocate an `N` element array of type `T` is

```
T * p = new T[N];
```

Notice that what comes after the new token is the type name followed by the square brackets with the expression in square brackets. If we allocated q dynamically as above we can use it as if it were an ordinary array, as follows:

```
for (int i = 0; i < 100; i++)
    q[i] = i*i;
```

Now `q` can be treated like the name of an array, but it is different, because unlike an array name, `q` is not a `const` pointer. It can be assigned to a different memory location. Note that `*q` is the first element of the array, i.e., `q[0]`, and `*(q+1)` is the same as `q[1]`. More generally, `*(q+k)` is the same as `q[k]`.

To delete the memory associated with a dynamic array, you have to use a slightly different form of the `delete` operator:

```
delete[] q;  // or delete [] q;
```

The square brackets go between the `delete` and the pointer name. You do not put anything in between the square brackets.

```
delete   q[];   // WRONG
delete [100] q; // WRONG
delete    q;    // WRONG
```

A few examples will illustrate. The first asks the user to enter a string and stores it in an array of fixed size. It then allocates an array of twice the size and puts two copies of the string into it.

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

int main( )
{
    char input[100];

    cout << "Enter a string:";
    cin >> input;
    int len = strlen(input);
    char * ditto = new char[2*len];

    strcpy(ditto, input);
```

```
    strcat(ditto, input);
    cout << ditto << endl;
    delete [] ditto ;

    return 0;
}
```

The next program shows how a function can create a dynamic array and return a pointer to it.

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

int * doublesize ( int a[], int size )
{
    int * p = new int[size*2];
    for ( int i = 0; i < size; i++ )
        p[i] = a[i];

    return p;
}


int main( )
{
    int n;
    int *squares, *moresquares;

    cout << "How many squares to create?";
    cin >> n;
    if ( n <= 0 || n > 10000 )
        return 1;

    squares = new int[n];
    for ( int i = 0; i < n; i++ )
        squares[i] = i*i;

    p2 = doublesize(squares, n);

    for ( int i = n+1; i < 2*n; i++ )
        p2[i] = i*i;

    for ( int i = 0; i < 2*n; i++ )
        cout << p2[i] << " ";
    cout << endl;

    delete [] squares;
    delete [] p2;

    return 0;
}
```