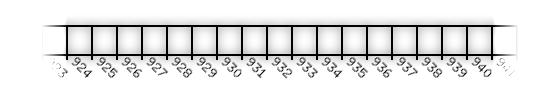
**Pointers & Dynamic Arrays**

**Preamble**

Consider what we know so far about variables. The computer’s memory is made up of lots of bytes – each byte has a number, or an address, associated to it. This picture might represent memory addresses 924 – 940.

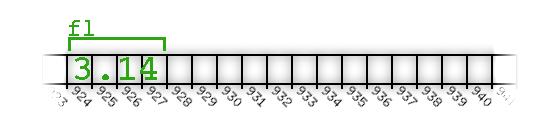
[](http://cis.payap.ac.th/wp-content/uploads/2009/11/pt1.jpg)

These memory addresses can then be used to store some data needed in our program;

#include <iostream>

int main()  
{  
 float fl=3.14;  
 std::cout << fl << std::endl;  
 return 0;  
}

In this case we declare a memory location to store a floating point number, assign its value to be 3.14, and then return it to the screen. Floats normally take up 4 bytes of memory, so 4 memory locations are reserved and the number is stored there;

[](http://cis.payap.ac.th/wp-content/uploads/2009/11/pt2.jpg)

(Note that 924 isn’t 3, 925 isn’t ., 926 isn’t 1 and 927 isn’t 4 – the number is stored in binary form taking up all 4 bytes space.)

Now let’s look closer at line 5 of the code;

std::cout << fl << std::endl;

When we use fl in a line like this, 2 distinct things occur;

1) The program finds and grabs the address reserved for fl (i.e. 924). The program grabs the address of the first byte of data.

2) The contents stored at that address are retrieved. The program knowing a float takes 4 bytes space will return 4 bytes worth of data.

So far, normally (but not always) we have performed both tasks together, now consider if we want to separate these two tasks. C++ gives us a couple of operators which enable us to only perform one of the tasks;

|  |  |  |
| --- | --- | --- |
| **operator** | **Meaning** | **example** |
| & | do only step 1 on a variable | &fl |
| \* | do step 2 on a number(address) | \*some\_num |

Given this if we change line 5 of the code to;

std::cout << “fl’s address=” << (unsigned int) &fl << std::endl;

The output of this will be somewhat different, as we are only grabbing the address of the variable, not the contents of the variable. Consider for a moment how we have used the & symbol before with Call by Reference variables. When I tested this program, this was my output;

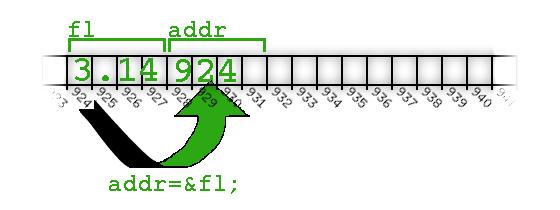
**fl’s address=1244884**

Interestingly the address of f1 is actually an integer, so we can actually store the address of f1 in another variable, for instance;

unsigned int addr=(unsigned int) &fl;

std::cout << “fl’s address=” << addr << std::endl;

Here we declare a new variable called addr to store the address of variable f1.

[](http://cis.payap.ac.th/wp-content/uploads/2009/11/pt3.jpg)

The second operator is the \* operator, this performs step 2, i.e. returns the contents stored at an address. So, we could run some code like this;

#include <iostream>

int main()

{  
 float fl=3.14;  
 unsigned int addr=(unsigned int) &fl;  
 std::cout << “fl’s address=” << addr << std::endl;  
 std::cout << “addr’s contents=” << \* (float\*) addr << std::endl;  
 return 0;  
}

Note that we have had to add a little ‘(float\*) part to get this to work. Don’t worry too much about this, as now we are going to use a completely different, yet related ‘**\***‘ operator as we start to look at pointers.

**Pointers**

A pointer is the memory address of a variable, it’s called a pointer because it ‘points’ towards the variable, identifying it by where it is, rather than by its name. In the above case we could either talk about variable fl, or we could talk about the variable stored in location 1244884. We have already used pointers when dealing with call-by-reference parameters – where the call by reference parameter says we want to deal with the variable stored in location X, rather than the value stored in the variable stored in location X. The system automatically sorts this out for us, without us even needing to know what a pointer is, but to take more control over the computer’s memory, there are a number of cases where we will want to use pointers. We will look at some of these cases, but first let’s look further into what a pointer is, and how we can manipulate them.

We can store a pointer in a variable, but not of type int or double, it has to be pointer type. A pointer is an address, *an address is an integer, but a pointer is not an integer!* – to declare a variable of pointer type we use the ‘\*’ operator (a slightly different one to the one used in the preamble). So, a pointer to point towards a variable of double type can be declared using;

double \*p;

This can only store pointers to variables of double type, as we may need to know the size of the memory location used by the variable. Unsurprisingly you use the int keyword to declare pointers to variables of  
type int. Note that we can declare both variables and pointers to variable in the same declaration;

int \*p1, \*p2, v1, v2;

Having declared these pointers, we can then set them to point towards a particular variable;

p1 = &v1;

Having done this, we now have two ways of referring to the data stored in that memory location, either by ‘***v1***’ or ‘***\*p1***‘ (the variable pointed to by p1). The asterisk here is called the ‘***dereferencing operator***’ and the pointer is said to be ‘***dereferenced***‘. Consider this;

v1 = 0;

p1 = &v1;  
\*p1 = 42;  
cout << v1 << endl;  
cout << \*p1 << endl;

This will output;

**42**

**42**

While p1 contains a pointer to v1, they refer to the same memory location, so any changes are made to both. If we set our second pointer to point to v1, it will also output 42.

p2=p1;

cout << \*p2;

Note the difference which happens if you add the dereferencing operator;

\*p2=\*p1;

When adding the askerisk we are no longer dealing with the pointers, but the memory locations they point to – here the value stored in the memory location pointed to by pointer p2, will be changed to the value stored in  
the memory location pointed to by p1.

So, now we can manipulate data stored in a variable without ever mentioning the name of the variable – the same as we can when using call by reference parameters. Exciting stuff! It’s a small step on to creating  
‘***dynamic variables***‘, variables which are created and destroyed by the program. The new operator can be used to create variables with no identifier for their name – nameless variables which we refer to via pointers;

int \*p1;

p1 = new int;  
cin >> \*p1;  
\*p1 = \*p1 + 7;  
cout << \*p1;

This creates a new nameless dynamic variable, which takes in an input from the keyboard, adds 7 to it and returns it.

**The Freestore**

The freestore is also called the heap – an area of memory used by dynamic variables. If we have too many dynamic variables the heap will be used up – and any additional calls to ‘new’ will cause errors. The size  
of the heap varies, but normally it is ‘big enough’. Pretty much all of the programs you write will not run into this problem, however, it’s a good idea to get into the habit of recycling the heap, when you’ve finished with dynamic variables. The keyword delete can be used to remove dynamic variables, making the memory available again.

delete p1;

This call makes the pointer p1 undefined, removing the data stored in the computer’s memory, but not the pointer – we could reuse the pointer again if we choose.

This pointer is known as a ‘***dangling pointer***’ – a pointer which doesn’t point anywhere – trying to use the ‘***\*p***‘  
will result in unpredictable and disastrous memory access.

**Type Definitions**

The typedef command can be used to define an alias for any other type name or definition. So we could use typedef to create an alias name for the double type;

typedef double Ken;

We can then declare variables of type double (or rather type Ken), using;

Ken v1;

This can be useful when declaring pointers, as we can declare a new type for pointers such as;

typedef int\* IntPtr;

Now we can declare new pointers to variables of int type, by using the IntPtr type name;

IntPtr p;

Why? Well, it means there will be no confusion between declaring pointers and integers as now we have a new type with which to declare pointers. Looking at this you should notice that the following two declarations are  
the same;

int \*p1;

int\* p1;

But, this can be confusing in cases such as;

int\* p1, p2;

where only p1 is a pointer, p2 is a normal integer variable.

The second benefit of creating an alias type definition is if we need to send a pointer as a parameter to a function, particularly if we need to send it as a call by reference parameter – in which case we would need to attach both a dereferencing operator (\*) and an ampersand (&) – which can cause problems. Therefore, by using alias type definitions we could send it more easily;

void sample\_function(IntPtr& pointer\_variable);

So if we need to create a type definition for pointers, why didn’t C++ do it for us in the first place? Well, C++ prides itself on the minimal number of keywords it uses, adding pointer types would mean adding more keywords, when the ‘\*’ would do just fine.

Now we should be reasonably happy with how pointers can be manipulated….  
But that leads us on the big question… Why? Well to begin with we’ve been using pointers quite a bit already! :P

**Arrays – Dynamic Arrays**

Array variables are actually pointer variables. Arrays are a series of addresses in the computers memory – i.e. a series of pointers. In the following declarations a and p are the same kind of variable;

int a[10];

typedef int\* IntPtr;

IntPtr p;

\They are so much the same kind of variable that we can assign the pointer p to the array a;

p=a;

After this assignment, p[0] points to the same memory location as a[0] and so on. The difference between the two is just that you can’t change the pointer value of an array variable as you can a pointer variable. So while  
it is legal to point p towards another memory location, we can’t change where a points;

IntPtr p2;

a=p2; // NOT LEGAL

This leads us nicely on to another big use for pointers – we can create dynamic arrays using pointers. One problem with arrays that we have faced so far is that we are forced to specify the size of the array when we declare it – and we are forced to choose a constant size (18 holes or 9 holes). ‘***Dynamic Arrays***‘ are arrays whose size we can change or allow the user to input the expected size.

typedef double\* DoublePtr;

DoublePtr d;  
d = new double[10];

In this example we can replace double with any type (including structs or classes), and we can replace 10 with any number including a user inputted value. If we are using our own user specified types (objects) with arrays, then the delete statement becomes more important. To delete the whole array from the heap we use the following command;

delete [] d;

The square brackets inform the computer that ‘d’ is an array, so it checks the size and frees up the whole array from the heap. Suppose we created a pointer to an array of a user specified type, such as an object. We shouldn’t reassign this pointer to point to another area of the computers memory, or it could confuse the system when the delete call is made, due to the changing size of each indexed location.

**Multidimensional Dynamic Arrays**

Multidimensional dynamic arrays are ‘arrays of arrays’ or ‘arrays of arrays of arrays’ etc. To create a two dimensional dynamic array, first create a one dimensional array of whatever type you require, then for each  
of those elements create further dynamic arrays;

typedef int\* IntArrayPtr;

IntArrayPtr \*m = new IntArrayPtr[3];  
for (int i = 0; i<3; i++)  
 m[i] = new int[4];

This creates a 3 by 4 dynamic array. For each new array created, the array needs to be deleted from the heap after it has been finished being used;

for(i=0; i<3; i++)

delete [] m[i];  
delete [] m;