**Pointers**

1. **Pointers and Dereference**

The pointer variable stores a memory address.

While references are a newer mechanism that originated in C++, pointers are an older mechanism that was inherited from C. It is advised to avoid using pointers as much as possible, since a reference will usually do the trick.

Similar to C, we declare a pointer with the pointer type, by adding a \*. We can then set the address to this pointer to initialize the pointer.

int \*ptr;

ptr = &gum;

🡪 To get the value at memory address kept by the pointer, we use the dereference operator \*

* When \* is used in a declaration, it is creating a pointer.
* When \* is not used in a declaration, it is a dereference operator.

*NOTE* We can change the value of the variable **being pointed** by changing the value of **\*pointer** (deref)

v1 = 0;

p1 = &v1;

\*p1 = 42;

cout << v1 << endl; // 42

cout << \*p1 << endl; // 42

Also note the use of ptr1 = ptr2, and \*p1 = \*p2

Diagram

Description automatically generated

1. **Dynamic Variables**

Since a pointer can be used to refer to a variable, we can even manipulate variables that have **no identifiers**. To create such variable, we use the **new** keyword , followed by **variable** **type\_name**

Data\_Type \*p1;

p1 = new Data\_Type;

cin >> \*p1;

cout << \*p1 + 7;

The **new** keyword produces a new, nameless variable, and **returns a pointer** to this new variable.

1. **Basic Memory Management**

A speical area of memory, called the heap, is used to store dynamic variables. The call to the **new** operator will create the new dynamic variable in the heap, and return a pointer to this variable.

The **delete** operator can be used to eliminate the dynamic variable and frees the memory.

Syntax: delete ptr\_to\_variable\_to\_free

delete ptr1;

After a call to delete, the value of the **pointer variable**, like ptr1 above, is **undefined**.

1. **Null Pointer**

We can declare a pointer without initializing it with any memory address:

int \*ptr;

OR, if we freed a dynamic variable, the value of the pointer is now **undefined**.

However, this is dangerous, because we would not know where it’s pointing (**dangling pointers**), and a dereference call would be disastrous. Therefore, we can assign it a value of nullptr (keyword)

int \*ptr = nullptr;

1. **Defining Pointer Types**

Note that to avoid using int\*, we can typedef this to become our own-defined type (e.g., IntPtr)

typedef int\* IntPtr;

IntPtr p; // equivalent to int \*p;

🡪 Avoids errors and confusing, for example with int\* p1, p2; (only p1 becomes a pointer)

or void sample\_function(int\*& pointer\_variable);

*(call-by-reference to pointer variable – changing where pointer points)*

1. **Pointer Arithmetic**

Say we have a DoublePtr d = new double [10];

Now we know that d contains the address of the indexed variable d[0]

🡪 d+1 is the address of d[1], d+2 is the address of d[2]

*NOTE* that even though the address gap is not 1 (due to data types, e.g., double can take up to 8 bytes), the compiler can still evaluate to the next element for us.

Therefore, the following codes are equivalent:

|  |  |
| --- | --- |
| for (int i = 0; i < arraySize; i++)  cout << \*(d + i)<< " "; | for (int i = 0; i < arraySize; i++)  cout << d[i] << " "; |

**Dynamic Arrays**

A **dynamic array** is an array whose size is not specified when you write the program, but is determined while the program is running. In C++ an array variable is actually a pointer variable that points to the first indexed variable of the array.

int a[10];

typedef int\* IntPtr;

IntPtr p;

🡪 You can resize the dynamic array

*Note* that you *cannot change the pointer value in an array variable* ~~a = ptr2;~~

Diagram

Description automatically generated with medium confidence

**Using dynamic arrays**

Dynamic arrays are created using the new operator. Since array variables are pointer variables, you can use the new operator to create dynamic variables that are arrays and treat these dynamic array variables as if they were ordinary arrays.

* Define a pointer type: Define a type for pointers to variables of the same type as the elements of the array. For example, if the dynamic array is an array of double, you might use the following:

typedef double\* DoubleArrayPtr;

* Declare a pointer variable: Declare a pointer variable of this defined type. The pointer variable will point to the dynamic array in memory and will serve as the name of the dynamic array.

DoubleArrayPtr a;

* Call new: Create a dynamic array using the new operator:

a = new double[array\_size];

* Use like an ordinary array: The pointer variable, such as a, is used just like an ordinary array (index, declare and pass to function like ordinary arrays)
* Call delete[ ]: When your program is finished with the dynamic variable, use delete and empty square brackets along with the pointer variable to free the memory. If we don’t use [], the compiler might only delete the memory of the first element.

delete [] a;

\*\*The major benefit of dynamic array is that the array size need not be specified at declaration. However, we still cannot shrink the array size during runtime. We can only allocate memory for a new array, then copy over the previous values.

This is harder than using <vector>.

**Multidimensional Dynamic Arrays**

Just remember that a multidimensional array is just an array of arrays, or array of arrays of arrays.

🡪 If we want to create a 2 dimensional array:

* A type definition may help to keep things straight.

typedef int\* IntArrayPtr;

* To obtain a 3-by-4 array of ints, you want an array whose base type is IntArrayPtr.

IntArrayPtr \*m = new IntArrayPtr[rows]; (pointer m to array of int pointers)

* This is an array of three pointers, each of which can name a dynamic array of ints, as follows:

for (int i = 0; i < rows; i++) {

m[i] = new int[cols];

}

Now we have a *rows x cols* 2D array.

🡪 *Note* the call to delete [], as we have to first delete the sub-arrays before deleting the main array.

for (int i = 0; i < rows; i++) {

m[i] = new int[cols];

}