* **Pointers**

A **pointer** is a variable whose value is the address of another variable.

**Example:**

**int a=10;**

**int \*ptr;**

**ptr=&a; /\* The pointer holds the address of a \*/**

**\*ptr=20; /\* dereference the pointer and change the value stored in the memory location pointed by ptr \*/**

**ptr=new int; /\* a new memory of 4 bytes is reserved and its starting address is stored in ptr \*/**

**\*ptr=10;**

**delete ptr; /\* delete memory \*/**

* **Inaccessible objects**

The inaccessible object bug: changing the value of the only pointer to an object, so you can’t access the object anymore

**thing\* p = new thing();**

**thing\* q = new thing();**

**p = q; //The previous memory that pointer p was pointing to is now inaccessible.**

* **Memory leaks**

Memory leaks: forgetting to delete dynamic data (often related to inaccessible objects).

**void foo() {**

**double\* q = new double(3.0);**

**/\*no delete q in the function\*/**

**/\* Access to memory pointed by q will be lost but it**

**won’t be deleted \*/**

**}**

**...**

**for(i=0;i<1000000;i++) foo(); // massive memory leak!**

* **Dangling pointers**

A "dangling pointer" is a pointer variable that contains a non-null address that is no longer valid... the pointer isn’t null, but it isn’t pointing to a valid object either

**int\* bar() {**

**int i; //local variable, allocated on stack**

**return &i; // return pointer to local variable...**

**// bad news! Stack frame is popped upon return**

**}**

**Another Example of a dangling pointer:**

**int\* p;**

**int\* q;**

**p = new int(99);**

**q = p;**

**delete p; // q and p are now dangling pointers (they are pointing to deleted memory location)**

**p = nullptr; // q is still dangling**

* **Reading C++ type declarations**

C++ type declarations can be more easily understood when read ‘backwards’, that isright to left, keeping in mind that **\*** means ‘pointer to’ and **&** means ‘reference to’:

**Examples:**

**int \* p // p is a pointer an int**

**int const \* p // p is a pointer to a const int**

**// which means the same thing as...**

**const int \* p // p is a pointer to an int that is const**

**int \* const p // p is a const pointer to an int**

**int const \* const p // p is a const pointer to a const int**

**// which means the same thing as...**

**const int \* const p // p is a const pointer to an int that is const**

**int & p // p is a reference an int**

**int const & p // p is a reference to a const int**

**// which means the same thing as...**

**const int & p // p is a reference to an int that is const**