# Pointer

February 13, 2017

A pointer is a compound type that "points to" another type. Like references, pointers are used for indirect access to other objects. A pointer is an object in its own right. Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime. A pointer need not be initialized at the time it is defined. Like other built-in types, pointers defined at block scope have undefined value if they are not initialized.

We define a pointer type by writing a declarator of the form *d, where d is the name being defined. The * must be repeated for each pointer variable.

int *ip1, *ip2; // both ip1 and ip2 are pointers to int

double dp, *dp2; // dp2 is a pointer to double; dp is a double

Taking the Address of an Object

A pointer holds the address of another object. We get the address of an object by usin the address-of operator (the & operator).

int ival = 42;

int *p = &ival; // p holds the address of ival; p is a pointer to ival

We may not define a pointer to a reference. With two exceptions, the types of the pointer

and the object to which it points must match.

double dval;

double *pd = &dval; // ok: initializer is the address of a double

double *pd2 = pd; // ok: initializer is a pointer to double

int *pi = pd; // error: types of pi and pd differ

pi = &dval; // error: assigning the address of a double to a pointer to int

The value (i.e., the address) stored in a pointer can be in one of four states:

1. It can point to an object.

2. It can point to the location just immediately past the end of an object.

3. It can be a null pointer, indicating that it is not bound to any object.

4. It can be invalid; values other than the preceding three are invalid.

When a pointer points to an object, we can use the dereference operator (the * operator) to access that object.

int ival = 42;

int *p = &ival; // p holds the address of ival; p is a pointer to ival

cout << *p; // * yields the object to which p points; prints 42

Dereferencing a pointer yields the object to which the pointer points. We can assign to that object by assigning to the result of the dereference:

*p = 0; // * yields the object; we assign a new value to ival through p

cout << *p; // prints 0

When we assign to *p, we are assigning to the object to which p points.

We may dereference only a valid pointer that points to an object.

Some symbols, such as & and *, are used as both an operator in an expression and as

part of a declaration. The context in which a symbol is used determines what the symbol means.

```
int i = 42;
int &r = i; // & follows a type and is part of a declaration; r is a reference
int *p; // * follows a type and is part of a declaration; p is a pointer
p = &i; // & is used in an expression as the address-of operator
*p = i; // * is used in an expression as the dereference operator
int &r2 = *p; // & is part of the declaration; * is the dereference operator
```

In declarations, & and * are used to form compound types. In expressions, these same symbols are used to denote an operator.

A null pointer does not point to any object. Code can check whether a pointer is null before attempting to use it. To obtain a null pointer:

```
int *p1 = nullptr; // equivalent to int *p1 = 0;
int *p2 = 0; // directly initializes p2 from the literal constant 0
// must #include cstdlib
int *p3 = NULL; // equivalent to int *p3 = 0;
```

nullptr is a literal that has a special type that can be converted to any other pointer type. Older programs sometimes use a preprocessor variable named NULL, which the cstdlib header defines as 0. The preprocessor is a program that runs before the compiler. Preprocessor variables are managed by the preprocessor, and are not part of the std namespace. As a result, we refer to them directly without the std:: prefix. When we use a preprocessor variable, the preprocessor automatically replaces the variable by its value. Hence, initializing a pointer to NULL is equivalent to initializing it to 0. Modern C++

programs generally should avoid using NULL and use nullptr instead.

It is illegal to assign an int variable to a pointer, even if the variables value happens to be 0.

int zero = 0;

pi = zero; // error: cannot assign an int to a pointer

Uninitialized pointers are a common source of run-time errors. Under most compilers, when we use an uninitialized pointer, the bits in the memory in which the pointer resides are used as an address. Using an uninitialized pointer is a request to access a supposed object at that supposed location. There is no way to distinguish a valid address from an invalid one formed from the bits that happen to be in the memory in which the pointer was allocated. If possible, define a pointer only after the object to which it should point has been defined. If there is no object to bind to a pointer, then initialize the pointer to nullptr or zero. That way, the program can detect that the pointer does not point to an object.

The type void* is a special pointer type that can hold the address of any object.

double obj = 3.14, *pd = &obj;

// ok: void* can hold the address value of any data pointer type

void *pv = &obj; // obj can be an object of any type

pv = pd; // pv can hold a pointer to any type

There are only a limited number of things we can do with a void* pointer: We can compare it to another pointer, we can pass it to or return it from a function, and we can assign it to another void* pointer. We can not use a void* to operate on the object it addresses—we dont know that objects type, and the type determines what operations we can perform

on the object.

Generally, we use a void* pointer to deal with memory as memory, rather than using the pointer to access the object stored in that memory.