Compound Types

Overview

A compound type is a type that is defined in terms of another type. C++ has several compound types, two of which are References and Pointers. Defining variables of compound type is more complicated than the declarations we've seen so far. A declaration is a **base type** followed by a list of **declarator**. Each declarator names a variable and gives the variable a type that is related to the base type.
The declarations we have seen so far have declarators that are nothing more than variable names. The type of such variable is the base type of the declaration.
More complicated declarators specify variables with compound types that are built from the base type of the declaration.

> **Tldr**
>
> compound type is a type that is defined in terms of another type.

References

A **reference** defines an alternative name for an object. A reference type "refers to" another type. We define a reference type by writing a declarator of the form `&d`, where `d` is the name being declared.

```
int ival = 1024;
int &refVal = ival;
//refVal refers to (is another name for) ival
```

A Reference Is an Alias

> **Note**
>
> A reference is not an object. Instead, a reference is just another name for an already existing object.

After a reference has been defined, *all* operations on that reference are actually operations on the object to which the reference is bound:

```
refVal = 2;
// ival = 2
```

When we assign to a reference, we are assigning to the object to which the reference is bound. When we fetch the value of a reference, we are really fetching the value of the object to which the reference is bound. Similarly, when we use a reference as an initializer, we are really using the object to which the reference is bound.

Because reference are not objects, we may not define a reference to a reference.

Reference Definitions

We can define multiple references in a single definition. Each identifier that is a reference must be preceded by the `&` symbol.

```
int i = 1, i2 = 2; // i and i2 are both int
int &r = i, r2 = i2; // r is a reference bound to i; r2 is an int
int i3 = 3, &ri = i3; // i3 is int; ri is a reference bound to i3
int &r3 = i3, &r4 = i2; // both r3 and r4 are references
```

Pointers

A **pointer** is a compound type that "points to" another type. Like references, pointers are used for indirect access to other objects. Unlike a reference, 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. Unlike a reference, a pointer does not need to be initialized at the time it is defined. Like other built-it 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;
```

Taking The Address of an Object

A point holds the address of another object. We get the address of an object by using the address-of operator `&`.

```
int ival = 42;
int *p = &ival // p holds the address of ival; p is a pointer to ival
```

The second statement defines `p` as a pointer to `int` and initializes `p` to point to the `int` object named `ival`. Because reference are not objects, they don't have addresses. Hence, we may not define a pointer to a reference.

The types must match because the type of the pointer is used to infer the type of the object to which the pointer points. If a pointer addressed an object of another type, operation performed on the underlying object would fail.

Pointer Value

The value (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; value other than the preceding three are invalid.

Using a Pointer to Access an Object

When a pointer point to an object, we can use the dereference operator `*` to access that object.

```
int ival = 42;
int *p = &ival; //p holds the address of ival; p points to ival
cout << *p; // * yield the object to which p points; print 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 << ival; // prints 0
```

Note

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

Important

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 symbols means

```
int i = 42;
int &r = i;     // r is a reference
int *p;                 // p is a pointer
p = &i;                 // address-of operator
*p = i;                 // 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. Because the same symbol is used with very different meanings, it can be helpful to ignore appearances and think of them as if they were different symbols.

Null pointers

A null pointer does not point to any object. Code can check whether a pointer is null before attempting to use it. There are several ways to obtain a null pointer.

```
int *p1 = nullptr; // equiv to int *p1 = 0;
int *p2 = 0;
//must #include <cstldlib>
int *p3 = NULL;
```

The most direct approach is the `nullptr`.

Tip

Modern C++ programs generally should avoid `NULL` and use `nullptr` instead.

⚠ Caution

It is illegal to assign an `int` variable to a pointer, event if the variable's value happens to be 0.

```
int zero = 0;
pi = zero; // error: cannot assign an int to a pointer
```

Tip

Uninitialized pointers are a common source of run-time errors. As with any other uninitialized variable, what happens when we use an uninitial- ized pointer is undefined. Using an uninitialized pointer almost always results in a run-time crash. However, debugging the resulting crashes can be surprisingly hard. Under most compilers, when we use an uninitialized pointer, the bits in the mem- ory 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. Our recommendation to initialize all variables is particularly important for point- ers. 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.

Assignment and Pointers

Both pointers and references give indirect access to other objects. However, there are important differences in how they do so.
There is no such identity between a pointer and the address that it holds. As with any other (nonreference) variable, when we assign to a pointer, we give the pointer itself a new value. Assignment makes the pointer points to a different object.

```
int i = 42;
int *pi = 0;    // pi = nullptr
int *pi2 = &i;  // pi2 holds the address of i
int *pi3;               // uninitialized
pi3 = pi2;              // both address the same object (i)
pi2 = 0;                // pi2 = nullptr
```

It can be hard to keep straight whether an assignment changes the pointer or the object to which the pointer points. The important thing to keep in mind is that assignment changes its left-hand operand.

```
pi = &ival; // value in pi change; pi now points to ival
```

we assign a new value to `pi`, which changes the address that `pi` holds. On the other hand, when we write

```
*pi = 0 // value in ival change; pi is unchanged
```

Other Pointer Operations

As long as the pointer has a valid value, we can use a pointer in a condition. Just as when we use an arithmetic value in a condition, if the pointer is 0, then the condition is `false`

```
int ival = 1024;
int *pi = 0;            //pi = nullptr
int *pi2 = &ival        //holds the address of ival
if (pi)                         // false
        ...
if (pi2)                        // true
        ...
```

Any nonzero pointer evaluates as `true`

Given two valid pointers of the same type, we can compare them using the equality == or inequality (!=) operators. The result of these operators has type bool. Two pointers are equal if they hold the same address and unequal otherwise. Two pointers hold the same address (i.e., are equal) if they are both null, if they address the same object, or if they are both pointers one past the same object. Note that it is possible for a pointer to an object and a pointer one past the end of a different object to hold the same address. Such pointers will compare equal. Because these operations use the value of the pointer, a pointer used in a condition or in a comparison must be a valid pointer. Using an invalid pointer as a condition or in a comparison is undefined.

void Pointers

The type `void*` is a special type that can hold the address of any object. Like any other pointer, a `void*` pointers holds an address, but the type of the object at that address is unknown

```
double obj = 3.14, *pd = &obj;
void *pv = &obj        // obj can be 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
- pass it or return it from a function
- assign it to another `void*` pointer
  We cannot
- use it to operate on the object it address
  - we don't know that object's type

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

Understanding Compound type Declarations

A variable definition consists of a base type and a list of declarators. Each declarator can relate its variable to the base type differently from the other declarators in the same definition. Thus, a single definition might define variables of different types

```
// i is an int; p is a pointer to int; r is a reference to int
int i = 1024, *p = &i, &r = i;
```

Defining Multiple Variables

```
int* p; // legal but might be misleading
```

```
int* p1, p2; //p1 is a pointer; p2 is an int
```

```
int *p1, *p2;   // both are pointers
```

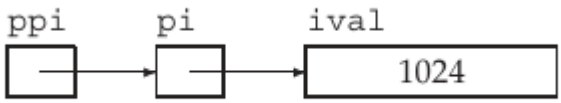
```
int *p1;        // pointer
int *p2;        // pointer
```

> **Tip**
>
> There is no single right way to define pointers or reference. The important thing to choose a style and use it consistently.

Pointers to Pointers

A pointer is an object in memory, so like any object it has an address. Therefore, we can store the address of a pointer in another pointer.
We indicate each pointer level by its own `*`. That is, we write `**` for a pointer to a pointer, `***` for a pointer to a pointer to a pointer, and so on

```
int ival = 1024;
int *pi = &ival;       // pi points to ival
int **ppi = &pi;       // ppi points to pi that points to ival
```



Dereferencing a pointer to a pointer yields a pointer. To access the underlying object, we must dereference the original pointer twice.

```
    cout << ival << endl;
    cout << *pi << endl;
    cout << **ppi << endl;
```

This print

```
1024
1024
1024
```

References to Pointers

A reference is not an object. Hence. we may not have a pointer to a reference. However, because a pointer is an object, we can define a reference to a pointer

```
int i = 42;
int *p;                 // p is a pointer
int *&r = p;    // r refers to the pointer p
r = &i;                 // r refers to a pointer; assigning &i to r makes p point to i
*r = 0;                 // dereferencing r yields i, the object to which p points;
changes i to 0
```

Tip

It can be easier to understand complicated pointer or reference declarations if you read them from right to left.