Pointer Things

Matt Warner

1 Using ptrdiff_t for pointer subtraction

By definition, `ptrdiff_t` can represent any possible result of subtracting two pointers that point to the same array. Or, if you perfer, the distance between two elements of the same array in either direction.

Note:-

`ptrdiff_t` is always a signed value.

It's nearly always equivalent to `make_signed_t<size_t>`

Although `ptrdiff_t` is signed and `size_t`, they're closely related. However, mixing them can easily lead to accidental signed-to-unsigned comparisions.

When comparing signed and unsigned values of the same size, the compiler first converts the signed value to unsigned. Therefore, if the signed value is negative, this can produce suprising results.

2 Pointer types

When creating a pointer, we can do so in the following ways:

```
int* x = nullptr; // Pointer to an int
const int* x = nullptr; // Pointer to a const int
int* const x = nullptr // Const pointer to an int
const int* const x = nullptr; // Const pointer to const int
```

The following are the operations allowed/forbidden for these types

```
int x = 2;
int y =3;

// Pointer to an int
int* ptr = new int(4);
delete ptr;
ptr = &x; // OK
*ptr = 3; // OK

// Pointer to const int (address pointed to can be changed (value stored at that address
↪  cannot) )
const int* ptr2 = &x;
ptr2 = &y; // OK
 *ptr2 = 3; // Error!

// const pointer to int (address cannot be changed but the value stored at address can)
int* const ptr3 = &x;
*ptr3 = 3; // OK
 ptr3 = &y; // Error!

// const pointer to a const int (address cannot be changed nor can the value stored at
↪  that address)
const int* const ptr4 = &x;
*ptr3 = 3; // Error!
 ptr3 = &y // Error;
```

3 Function pointers

Functions have memory addresses, so its only natural that we can create pointers to functions. The syntax looks like this:

```
return_type (*Funcptr) (parameter type, .....);
```

Now we can give it an address to a function. Note: you cannot declare and initalize function pointers at the same time. You must first declare and then initalize. Consider the following code:

```
void foo() { std::cout << "Hello from foo. "; }

int main() {

    // Declaring a function pointer
    void (*funcptr)();

    // funcptr is pointing to foo
    funcptr = foo;

    // calling the function.
    funcptr();

    return 0;
}
```

3.1 Passing a function pointer as a parameter

```
void print(void (*funcptr)()) {
    std::cout << funcptr();
}
int main () {
    void (*funcptr)();
    funcptr = foo;
    print(funcptr);
}
```