12.10 — Pass by address

In prior lessons, we've covered two different ways to pass an argument to a function: pass by value (2.4 -- Introduction to function parameters and arguments) and pass by reference (12.5 -- Pass by lvalue reference).

Here's a sample program that shows a `std::string` object being passed by value and by reference:

```
#include <iostream>
#include <string>

void printByValue(std::string val) // The function parameter is a copy of str
{
    std::cout << val << '\n'; // print the value via the copy
}

void printByReference(const std::string& ref) // The function parameter is a
reference that binds to str
{
    std::cout << ref << '\n'; // print the value via the reference
}

int main()
{
    std::string str{ "Hello, world!" };

    printByValue(str); // pass str by value, makes a copy of str
    printByReference(str); // pass str by reference, does not make a copy of str

    return 0;
}
```

When we pass argument `str` by value, the function parameter `val` receives a copy of the argument. Because the parameter is a copy of the argument, any changes to the `val` are made to the copy, not the original argument.

When we pass argument `str` by reference, the reference parameter `ref` is bound to the actual argument. This avoids making a copy of the argument. Because our reference parameter is const, we are not allowed to change `ref`. But if `ref` were non-const, any changes we made to `ref` would change `str`.

In both cases, the caller is providing the actual object (`str`) to be passed as an argument to the function call.

Pass by address

C++ provides a third way to pass values to a function, called pass by address. With **pass by address**, instead of providing an object as an argument, the caller provides an object's *address* (via a pointer). This pointer (holding the address of the object) is copied into a pointer parameter of the called function (which now also holds the address of the object). The function can then dereference that pointer to access the object whose address was passed.

Here's a version of the above program that adds a pass by address variant:

```
#include <iostream>
#include <string>

void printByValue(std::string val) // The function parameter is a copy of str
{
    std::cout << val << '\n'; // print the value via the copy
}

void printByReference(const std::string& ref) // The function parameter is a
reference that binds to str
{
    std::cout << ref << '\n'; // print the value via the reference
}

void printByAddress(const std::string* ptr) // The function parameter is a pointer
that holds the address of str
{
    std::cout << *ptr << '\n'; // print the value via the dereferenced pointer
}

int main()
{
    std::string str{ "Hello, world!" };

    printByValue(str); // pass str by value, makes a copy of str
    printByReference(str); // pass str by reference, does not make a copy of str
    printByAddress(&str); // pass str by address, does not make a copy of str

    return 0;
}
```

Note how similar all three of these versions are. Let's explore the pass by address version in more detail.

First, because we want our `printByAddress()` function to use pass by address, we've made our function parameter a pointer named `ptr`. Since `printByAddress()` will use `ptr` in a read-only manner, `ptr` is a pointer to a const value.

```
void printByAddress(const std::string* ptr)
{
    std::cout << *ptr << '\n'; // print the value via the dereferenced pointer
}
```

Inside the `printByAddress()` function, we dereference `ptr` parameter to access the value of the object being pointed to.

Second, when the function is called, we can't just pass in the `str` object -- we need to pass in the address of `str`. The easiest way to do that is to use the address-of operator (&) to get a pointer holding the address of `str`:

```
printByAddress(&str); // use address-of operator (&) to get pointer holding address
of str
```

When this call is executed, `&str` will create a pointer holding the address of `str`. This address is then copied into function parameter `ptr` as part of the function call. Because `ptr` now holds the address of `str`, when the function dereferences `ptr`, it will get the value of `str`, which the function prints to the console.

That's it.

Although we use the address-of operator in the above example to get the address of `str`, if we already had a pointer variable holding the address of `str`, we could use that instead:

```
int main()
{
    std::string str{ "Hello, world!" };

    printByValue(str); // pass str by value, makes a copy of str
    printByReference(str); // pass str by reference, does not make a copy of str
    printByAddress(&str); // pass str by address, does not make a copy of str

    std::string* ptr { &str }; // define a pointer variable holding the address of
str
    printByAddress(ptr); // pass str by address, does not make a copy of str

    return 0;
}
```

Pass by address does not make a copy of the object being pointed to

Consider the following statements:

```
std::string str{ "Hello, world!" };
printByAddress(&str); // use address-of operator (&) to get pointer holding address
of str
```

As we noted in 12.5 -- Pass by lvalue reference, copying a `std::string` is expensive, so that's something we want to avoid. When we pass a `std::string` by address, we're not copying the actual `std::string` object -- we're just copying the pointer (holding the address of the object) from the caller to the called function. Since an address is typically only 4 or 8 bytes, a pointer is only 4 or 8 bytes, so copying a pointer is always fast.

Thus, just like pass by reference, pass by address is fast, and avoids making a copy of the argument object.

Pass by address allows the function to modify the argument's value

When we pass an object by address, the function receives the address of the passed object, which it can access via dereferencing. Because this is the address of the actual argument object being passed (not a copy of the object), if the function parameter is a pointer to non-const, the function can modify the argument via the pointer parameter:

```
#include <iostream>

void changeValue(int* ptr) // note: ptr is a pointer to non-const in this example
{
    *ptr = 6; // change the value to 6
}

int main()
{
    int x{ 5 };

    std::cout << "x = " << x << '\n';

    changeValue(&x); // we're passing the address of x to the function

    std::cout << "x = " << x << '\n';

    return 0;
}
```

This prints:

```
x = 5
x = 6
```

As you can see, the argument is modified and this modification persists even after `changeValue()` has finished running.

If a function is not supposed to modify the object being passed in, the function parameter can be made a pointer to const:

```
void changeValue(const int* ptr) // note: ptr is now a pointer to a const
{
    *ptr = 6; // error: can not change const value
}
```

Null checking

Now consider this fairly innocent looking program:

```
#include <iostream>

void print(int* ptr)
{
        std::cout << *ptr << '\n';
}

int main()
{
        int x{ 5 };
        print(&x);

        int* myPtr {};
        print(myPtr);

        return 0;
}
```

When this program is run, it will print the value `5` and then most likely crash.

In the call to `print(myPtr)`, `myPtr` is a null pointer, so function parameter `ptr` will also be a null pointer. When this null pointer is dereferenced in the body of the function, undefined behavior results.

When passing a parameter by address, care should be taken to ensure the pointer is not a null pointer before you dereference the value. One way to do that is to use a conditional statement:

```
#include <iostream>

void print(int* ptr)
{
    if (ptr) // if ptr is not a null pointer
    {
        std::cout << *ptr << '\n';
    }
}

int main()
{
        int x{ 5 };

        print(&x);
        print(nullptr);

        return 0;
}
```

In the above program, we're testing `ptr` to ensure it is not null before we dereference it. While this is fine for such a simple function, in more complicated functions this can result in redundant logic (testing if ptr is not null multiple times) or nesting of the primary logic of the function (if contained in a block).

In most cases, it is more effective to do the opposite: test whether the function parameter is null as a precondition (9.6 -- Assert and static_assert) and handle the negative case immediately:

```
#include <iostream>

void print(int* ptr)
{
    if (!ptr) // if ptr is a null pointer, early return back to the caller
        return;

    // if we reached this point, we can assume ptr is valid
    // so no more testing or nesting required

    std::cout << *ptr << '\n';
}

int main()
{
        int x{ 5 };

        print(&x);
        print(nullptr);

        return 0;
}
```

If a null pointer should never be passed to the function, an `assert` (which we covered in lesson 9.6 -- Assert and static_assert) can be used instead (or also) (as asserts are intended to document things that should never happen):

```
#include <iostream>
#include <cassert>

void print(const int* ptr) // now a pointer to a const int
{
	assert(ptr); // fail the program in debug mode if a null pointer is passed (since this should never happen)

	// (optionally) handle this as an error case in production mode so we don't crash if it does happen
	if (!ptr)
		return;

	std::cout << *ptr << '\n';
}

int main()
{
	int x{ 5 };

	print(&x);
	print(nullptr);

	return 0;
}
```

Prefer pass by (const) reference

Note that function `print()` in the example above doesn't handle null values very well -- it effectively just aborts the function. Given this, why allow a user to pass in a null value at all? Pass by reference has the same benefits as pass by address without the risk of inadvertently dereferencing a null pointer.

Pass by const reference has a few other advantages over pass by address.

First, because an object being passed by address must have an address, only lvalues can be passed by address (as rvalues don't have addresses). Pass by const reference is more flexible, as it can accept lvalues and rvalues:

```
#include <iostream>

void printByValue(int val) // The function parameter is a copy of the argument
{
    std::cout << val << '\n'; // print the value via the copy
}

void printByReference(const int& ref) // The function parameter is a reference that
binds to the argument
{
    std::cout << ref << '\n'; // print the value via the reference
}

void printByAddress(const int* ptr) // The function parameter is a pointer that holds
the address of the argument
{
    std::cout << *ptr << '\n'; // print the value via the dereferenced pointer
}

int main()
{
    printByValue(5);     // valid (but makes a copy)
    printByReference(5); // valid (because the parameter is a const reference)
    printByAddress(&5);  // error: can't take address of r-value

    return 0;
}
```

Second, the syntax for pass by reference is natural, as we can just pass in literals or objects. With pass by address, our code ends up littered with ampersands (&) and asterisks (*).

In modern C++, most things that can be done with pass by address are better accomplished through other methods. Follow this common maxim: “Pass by reference when you can, pass by address when you must”.

Best practice

Prefer pass by reference to pass by address unless you have a specific reason to use pass by address.