13.12 — Member selection with pointers and references

Member selection for structs and references to structs

In lesson 13.7 -- Introduction to structs, members, and member selection, we showed that you can use the member selection operator (.) to select a member from a struct object:

```
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe { 1, 34, 65000.0 };

    // Use member selection operator (.) to select a member from struct object
    ++joe.age; // Joe had a birthday
    joe.wage = 68000.0; // Joe got a promotion

    return 0;
}
```

Since references to an object act just like the object itself, we can also use the member selection operator (.) to select a member from a reference to a struct:

```
#include <iostream>

struct Employee
{
    int id{};
    int age{};
    double wage{};
};

void printEmployee(const Employee& e)
{
    // Use member selection operator (.) to select member from reference to struct
    std::cout << "Id: " << e.id << '\n';
    std::cout << "Age: " << e.age << '\n';
    std::cout << "Wage: " << e.wage << '\n';
}

int main()
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    printEmployee(joe);

    return 0;
}
```

Member selection for pointers to structs

However, use of the member selection operator (.) doesn’t work if you have a pointer to a struct:

```
#include <iostream>

struct Employee
{
    int id{};
    int age{};
    double wage{};
};

int main()
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    Employee* ptr{ &joe };
    std::cout << ptr.id << '\n'; // Compile error: can't use operator. with pointers

    return 0;
}
```

With normal variables or references, we can access objects directly. However, because pointers hold addresses, we first need to dereference the pointer to get the object before we can do anything with it. So one way to access a member from a pointer to a struct is as follows:

```
#include <iostream>

struct Employee
{
    int id{};
    int age{};
    double wage{};
};

int main()
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    Employee* ptr{ &joe };
    std::cout << (*ptr).id << '\n'; // Not great but works: First dereference ptr,
then use member selection

    return 0;
}
```

However, this is a bit ugly, especially because we need to parenthesize the dereference operation so it will take precedence over the member selection operation.

To make for a cleaner syntax, C++ offers a **member selection from pointer operator (->)** (also sometimes called the **arrow operator**) that can be used to select members from a pointer to an object:

```
#include <iostream>

struct Employee
{
    int id{};
    int age{};
    double wage{};
};

int main()
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    Employee* ptr{ &joe };
    std::cout << ptr->id << '\n'; // Better: use -> to select member from pointer to
object

    return 0;
}
```

This member selection from pointer operator (->) works identically to the member selection operator (.) but does an implicit dereference of the pointer object before selecting the member. Thus `ptr->id` is equivalent to `(*ptr).id`.

This arrow operator is not only easier to type, but is also much less prone to error because the indirection is implicitly done for you, so there are no precedence issues to worry about. Consequently, when doing member access through a pointer, always use the -> operator instead of the . operator.

Best practice

When using a pointer to access the value of a member, use the member selection from pointer operator (->) instead of the member selection operator (.).

Mixing pointers and non-pointers to members

The member selection operator is always applied to the currently selected variable. If you have a mix of pointers and normal member variables, you can see member selections where . and -> are both used in sequence:

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

struct Paw
{
    int claws{};
};

struct Animal
{
    std::string name{};
    Paw paw{};
};

int main()
{
    Animal puma{ "Puma", { 5 } };

    Animal* ptr{ &puma };

    // ptr is a pointer, use ->
    // paw is not a pointer, use .

    std::cout << (ptr->paw).claws << '\n';

    return 0;
}
```

Note that in the case of `(ptr->paw).claws`, parentheses aren't necessary since both `operator->` and `operator.` evaluate in left to right order, but it does help readability slightly.