C/C++ **PRIMER**

LECTURE 3: C++ REFERENCES, OPERATOR PRECEDENCE, FUNCTIONS

Fabian Wermelinger
Harvard University

OUTLINE

- Address-of operator (recap)
- `C++` references
- Basic operators and operator precedence
- Functions and function pointers
- `C++` anonymous functions (`lambda`)

ADDRESS-OF AND C++ REFERENCES

ADDRESS-OF OPERATOR (RECAP)

Recall from last lecture, we can use the address-of operator `&` to obtain the *memory address* from any variable:

```
#include <iostream>
int main(void)
{
    int a;
    std::cout << &a << std::endl; // print the address of a to stdout
    return 0;
}
```

Compile this code and check the output on standard output (`stdout`)

```
$ g++ address_of.cpp
$ ./a.out
0x7ffec6d2d5e4  <- actual memory address of a (may change for different runs)
```

C++ REFERENCES

`C++` has introduced another way to reference objects. They are simply called *references*.

- A reference is a hybrid between a pointer and array.
- A reference **must** be *initialized* when it is *defined*. Like for an array, you can not change its reference later on.
- At the low-level, a reference is simply an address, like a pointer.
- References exist in `C++` because copy-by-value can be very expensive for custom types (like a `class`). More on this soon
- Pointers could have been used for this but they add complexity due to their dual character and are not considered safe in `C++`.

C++ REFERENCES

EXAMPLE: C++ REFERENCES

```cpp
#include <iostream>
int main(void)
{
    double a = 1.0;     // allocate storage for one double precision number
    double &ref_a = a; // reference to a (no new storage allocated for a)

    a = 2.0;      // set value of a to 2.0
    ref_a = 3.0; // set value of a to 3.0 (via reference)

    std::cout << &a << '\n' << &ref_a << '\n';
    return 0;
}
```

We can check the memory addresses of `a` and `ref_a`:

```
$ ./a.out
0x7ffd66b86608
0x7ffd66b86608
```

They are *identical*! Both variables *share* one memory instance.

C++ REFERENCES

EXAMPLE: C++ REFERENCES AND TYPE QUALIFIERS

You can use the type qualifiers with references too:

```cpp
#include <iostream>
int main(void)
{
    double a = 1.0;    // allocate storage for one double precision number
    const double &ref_a = a; // const reference to a (no new storage allocated for a)

    a = 2.0;     // set value of a to 2.0
    // ref_a = 3.0; // this is not OK! ref_a is a const reference (read-only)

    return 0;
}
```

BASIC OPERATORS AND OPERATOR PRECEDENCE

BASIC OPERATORS

The usual operators are available:

<table>
<tr><th>Operator</th><th>Type</th></tr>
<tr><td><code>++, --</code></td><td>Unary operator</td></tr>
<tr><td><code>+, -, *, /, %</code></td><td>Arithmetic operator</td></tr>
<tr><td><code><, <=, >, >=, ==, !=</code></td><td>Relational operator</td></tr>
<tr><td><code>&&, ||, !</code></td><td>Logical operator</td></tr>
<tr><td><code>&, |, <<, >>, ~, ^</code></td><td>Bitwise operator</td></tr>
<tr><td><code>=, +=, -=, *=, /=, %=</code></td><td>Assignment operator</td></tr>
<tr><td><code>(expression) ? (expression) : (expression)</code></td><td>Ternary operator</td></tr>
</table>

Unary operators | Binary operators | Ternary operators

OPERATOR PRECEDENCE

These operators have a *precedence*. It is very important that you know basic precedence rules of the various operators. If you are unsure, surround the expression in parenthesis `(...)`.

Let's have a look at `C++` operator precedence here:
https://en.cppreference.com/w/cpp/language/operator_precedence

Hands-On: 20 minutes (`hands-on/01/README.md`)
Work through the tasks in the `README.md` file.

HANDS-ON RECAP

OPERATOR PRECEDENCE

```cpp
#include <iostream>
int main(void)
{
    const char *ptr = "Hello World!"; // strings in C/C++ are nul terminated '\0'
                    //"Hello World!\0" how string looks like in memory
    for (; *ptr;) { // the loop is terminated when *ptr returns \0, which is the
                    // same as 'false'

        // The post-increment operator '++' has higher precedence than the
        // dereference operator '*'.  See
        // https://en.cppreference.com/w/cpp/language/operator_precedence
        //
        // Here is what happens:
        //
        // 1. ptr++ is applied.  This is the same as 'ptr = ptr + 1' BUT this
        // operator stores a copy of the old value before the increment.  After
        // the increment it returns the OLD value
        //
        // 2. The dereference operator is applied to the old value that has been
        // returned by ptr++
```

https://en.cppreference.com/w/cpp/language/operator_precedence

HANDS-ON RECAP

TERNARY OPERATOR

```
int main(int argc, char* argv[])
{
    // Write the following code in lines 8-13 with only one single statement,
    // that is, you can only use one semi-colon ';'
    int a; // statement 1
    if (argc > 0) {
        a = 10; // statement 2
    } else {
        a = -10; // statement 3
    }

    // solution:
    int a = (argc > 0) ? 10 : -10;

    // benefit: you could also declare 'a' const. You can't do that with the if-else.
    const int a = (argc > 0) ? 10 : -10;
```

FUNCTIONS, FUNCTION POINTERS AND LAMBDA'S

FUNCTION SIGNATURES

We want to compute the following integral using the mid-point rule:

$$\int_a^b \sin(x)\,\mathrm{d}x \approx \Delta x \sum_{i=0}^{n-1} \sin\big((i + 1/2)\Delta x\big),$$

where $\Delta x = (b - a)/n$ and $n \in \mathbb{N}_+$.

Possible implementation:

```cpp
#include <cmath>
double mid_point(const double a, const double b, const int n)
{
    const double dx = (b - a) / n; // integration interval
    double sum = 0;                // summation variable (must be initialized!)
    for (int i = 0; i < n; ++i) {
        sum += std::sin((i + 0.5) * dx);
    }
    return dx * sum;
}
```

FUNCTION SIGNATURES

Possible implementation:

```cpp
#include <cmath>
double mid_point(const double a, const double b, const int n)
{
    const double dx = (b - a) / n; // integration interval
    double sum = 0;                // summation variable (must be initialized!)
    for (int i = 0; i < n; ++i) {
        sum += std::sin((i + 0.5) * dx);
    }
    return dx * sum;
}
```

Function prototype:

```cpp
// function prototype:
ReturnType FunctionName(Parameter list);

// in example above:
double mid_point(const double a, const double b, const int n);
// ReturnType:     double
// FunctionName:   mid_point
// Parameter list: const double a, const double b, const int n
// The function body is what follows inside the block delimited by {...}
```

FUNCTION SIGNATURES

Function prototype:

```
// function prototype:
ReturnType FunctionName(Parameter list);

// in example above:
double mid_point(const double a, const double b, const int n);
// ReturnType:     double
// FunctionName:   mid_point
// Parameter list: const double a, const double b, const int n
// The function body is what follows inside the block delimited by {...}
```

Function signature:

```
// function signature:
FunctionName(Parameter list)
```

The *function signature* is the function prototype without the return type. Function signatures are important such that the compiler can *overload* functions that have the same name but different parameter lists.

THE TYPE OF A FUNCTION

Functions do have a *type*, just like any other variable has a type.

```
// function prototype:
double mid_point(const double a, const double b, const int n);

// its type is:
double(const double a, const double b, const int n)
```

Indeed we can take the address of a function:

```
#include <cstdio>
extern double mid_point(const double a, const double b, const int n);
int main(void)
{
    printf("%p\n", mid_point);
    return 0;
}
```

```
$ g++ function.cpp
$ ./a.out
0x555a286fa149
```

FUNCTION POINTERS

Because a function has a type and address, we can declare pointers to functions. (A function basically is a pointer.)

```
// function prototype:
double mid_point(const double a, const double b, const int n);

// its type is:
double(const double a, const double b, const int n)

// function pointer:
double (*fcn_pointer)(const double a, const double b, const int n);
fcn_pointer = &mid_point; // now the function pointer points to mid_point

// we can now use the function pointer like a normal function call
double result = fcn_pointer(0, 2*M_PI, 100); // compute integral of sin(x) in [0, 2pi]
```

FUNCTION POINTERS

`C` syntax for function pointers can be confusing when reading code. You can create a type alias for its type using a `typedef` or the `using` operator in `C++`:

```
extern double mid_point(const double a, const double b, const int n);
int main(void)
{
    // type alias the C way:
    typedef double(MidPointType)(const double a, const double b, const int n);

    // type alias the C++ way:
    using MidPointType = double(const double a, const double b, const int n);

    // use the new type alias in your code:
    MidPointType *fcn_pointer = &mid_point;
    // double result = fcn_pointer(...);

    return 0;
}
```

FUNCTION POINTERS

You can create a *pointer* type alias directly:

```
extern double mid_point(const double a, const double b, const int n);
int main(void)
{
    // type alias the C way: (MidPointType is a pointer to this type)
    typedef double(*MidPointType)(const double a, const double b, const int n);

    // type alias the C++ way: (MidPointType is a pointer to this type)
    using MidPointType = double(*)(const double a, const double b, const int n);

    // use the new type alias in your code:
    MidPointType fcn_pointer = &mid_point; // note the difference here!
    // double result = fcn_pointer(...);

    return 0;
}
```

C++ FUNCTIONALS AND LAMBDA

`C++` provides the `<functional>` header to hide these complexities:

```cpp
#include <functional>
extern double mid_point(const double a, const double b, const int n);
int main(void)
{
    std::function f = mid_point;

    // use the functional in your code
    // double result = f(...);

    return 0;
}
```

Note that the compiler is clever and it can deduce the type of `mid_point` *automatically*. This works because `std::function` is a *template* (we look at templates soon).

C++ FUNCTIONALS AND LAMBDA

You still have the option to be explicit with what you *mean* in your code (but here it is cleaner code if you don't):

```cpp
#include <functional>
extern double mid_point(const double a, const double b, const int n);
int main(void)
{
    // alternatively specify the function type
    std::function<double(const double a, const double b, const int n)> f = mid_point;

    // use the functional in your code
    // double result = f(...);

    return 0;
}
```

C++ LAMBDA'S

Lambda functions are *anonymous* functions. They are convenient if you only need a function locally in your code. The concept is the same as in `python`:

```python
import numpy as np
f = lambda x: np.sin(x)  # declare f as sin(x)
```

And the same in `C++`:

```cpp
#include <cmath>
auto f = [](double x) { return std::sin(x); };
```

More detail: https://en.cppreference.com/w/cpp/language/lambda.

C++ LAMBDA'S

```cpp
#include <cmath>
auto f = [](double x) { return std::sin(x); };
```

- `[]`: defines captures in the local scope. E.g., `[&]` would capture all variables in the local scope *by reference*. You could access and modify them inside the lambda without passing them in the parameter list.
- `auto`: is a special type placeholder since `C++11`. The actual type will automatically be inferred. *This can be convenient in some places but it also introduces some dangers. Code comprehension becomes more difficult and unexpected types may be used.*

C++ LAMBDA'S

Lets generalize our mid-point rule:

```cpp
#include <functional>
#include <cmath>
double mid_point(const std::function<double(double)> &f,
                 const double a, const double b, const int n)
{
    const double dx = (b - a) / n; // integration interval
    double sum = 0;                // summation variable (must be initialized!)
    for (int i = 0; i < n; ++i) {
        sum += f((i + 0.5) * dx); // f can be any function with a scalar argument
    }
    return dx * sum;
}
```

We can use it with a `lambda` function:

```cpp
int main(void)
{
    auto f = [](double x) { return std::sin(x) * std::cos(x); };
    // now use mid_point to integrate any integrand f(x)
    double result = mid_point(f, 0, 2 * M_PI, 100);
    return 0;
}
```