

# 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:

```
1 #include <iostream>
2 int main(void)
3 {
4     int a;
5     std::cout << &a << std::endl; // print the address of a to stdout
6     return 0;
7 }
```

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

```
1 $ g++ address_of.cpp
2 $ ./a.out
3 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

```
1 #include <iostream>
2 int main(void)
3 {
4     double a = 1.0;    // allocate storage for one double precision number
5     double &ref_a = a; // reference to a (no new storage allocated for a)
6
7     a = 2.0;           // set value of a to 2.0
8     ref_a = 3.0;       // set value of a to 3.0 (via reference)
9
10    std::cout << &a << '\n' << &ref_a << '\n';
11    return 0;
12 }
```

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

```
1 $ ./a.out
2 0x7ffd66b86608
3 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:

```
1 #include <iostream>
2 int main(void)
3 {
4     double a = 1.0;    // allocate storage for one double precision number
5     const double &ref_a = a; // const reference to a (no new storage allocated for a)
6
7     a = 2.0;          // set value of a to 2.0
8     // ref_a = 3.0; // this is not OK! ref_a is a const reference (read-only)
9
10    return 0;
11 }
```

# BASIC OPERATORS AND OPERATOR PRECEDENCE



# BASIC OPERATORS

The usual operators are available:

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

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](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

```
1 #include <iostream>
2 int main(void)
3 {
4     const char *ptr = "Hello World!"; // strings in C/C++ are nul terminated '\0'
5                                     // "Hello World!\0" how string looks like in memory
6     for (; *ptr;) { // the loop is terminated when *ptr returns \0, which is the
7                     // same as 'false'
8
9         // The post-increment operator '++' has higher precedence than the
10        // dereference operator '*'. See
11        // https://en.cppreference.com/w/cpp/language/operator_precedence
12        //
13        // Here is what happens:
14        //
15        // 1. ptr++ is applied. This is the same as 'ptr = ptr + 1' BUT this
16        // operator stores a copy of the old value before the increment. After
17        // the increment it returns the OLD value
18        //
19        // 2. The dereference operator is applied to the old value that has been
```

[https://en.cppreference.com/w/cpp/language/operator\\_precedence](https://en.cppreference.com/w/cpp/language/operator_precedence)

# HANDS-ON RECAP

## TERNARY OPERATOR

```
1 int main(int argc, char* argv[])
2 {
3     // Write the following code in lines 8-13 with only one single statement,
4     // that is, you can only use one semi-colon ';'
5     int a; // statement 1
6     if (argc > 0) {
7         a = 10; // statement 2
8     } else {
9         a = -10; // statement 3
10    }
11
12    // solution:
13    int a = (argc > 0) ? 10 : -10;
14
15    // benefit: you could also declare 'a' const. You can't do that with the if-else.
16    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](#):

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

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

Possible implementation:

```
1 #include <cmath>
2 double mid_point(const double a, const double b, const int n)
3 {
4     const double dx = (b - a) / n; // integration interval
5     double sum = 0;                // summation variable (must be initialized!)
6     for (int i = 0; i < n; ++i) {
7         sum += std::sin((i + 0.5) * dx);
8     }
9     return dx * sum;
10 }
```

# FUNCTION SIGNATURES

Possible implementation:

```
1 #include <cmath>
2 double mid_point(const double a, const double b, const int n)
3 {
4     const double dx = (b - a) / n; // integration interval
5     double sum = 0;                // summation variable (must be initialized!)
6     for (int i = 0; i < n; ++i) {
7         sum += std::sin((i + 0.5) * dx);
8     }
9     return dx * sum;
10 }
```

Function prototype:

```
1 // function prototype:
2 ReturnType FunctionName(Parameter list);
3
4 // in example above:
5 double mid_point(const double a, const double b, const int n);
6 // ReturnType:      double
7 // FunctionName:    mid_point
8 // Parameter list:  const double a, const double b, const int n
9 // The function body is what follows inside the block delimited by {...}
```

# FUNCTION SIGNATURES

## Function prototype:

```
1 // function prototype:
2 ReturnType FunctionName(Parameter list);
3
4 // in example above:
5 double mid_point(const double a, const double b, const int n);
6 // ReturnType:      double
7 // FunctionName:    mid_point
8 // Parameter list:  const double a, const double b, const int n
9 // The function body is what follows inside the block delimited by {...}
```

## Function signature:

```
1 // function signature:
2 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.

```
1 // function prototype:
2 double mid_point(const double a, const double b, const int n);
3
4 // its type is:
5 double(const double a, const double b, const int n)
```

Indeed we can take the address of a function:

```
1 #include <stdio>
2 extern double mid_point(const double a, const double b, const int n);
3 int main(void)
4 {
5     printf("%p\n", mid_point);
6     return 0;
7 }
```

```
1 $ g++ function.cpp
2 $ ./a.out
3 0x555a286fa149
```

# FUNCTION POINTERS

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

```
1 // function prototype:
2 double mid_point(const double a, const double b, const int n);
3
4 // its type is:
5 double(const double a, const double b, const int n)
6
7 // function pointer:
8 double (*fcn_pointer)(const double a, const double b, const int n);
9 fcn_pointer = &mid_point; // now the function pointer points to mid_point
10
11 // we can now use the function pointer like a normal function call
12 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++:

```
1 extern double mid_point(const double a, const double b, const int n);
2 int main(void)
3 {
4     // type alias the C way:
5     typedef double(MidPointType)(const double a, const double b, const int n);
6
7     // type alias the C++ way:
8     using MidPointType = double(const double a, const double b, const int n);
9
10    // use the new type alias in your code:
11    MidPointType *fcn_pointer = &mid_point;
12    // double result = fcn_pointer(...);
13
14    return 0;
15 }
```

# FUNCTION POINTERS

You can create a *pointer* type alias directly:

```
1 extern double mid_point(const double a, const double b, const int n);
2 int main(void)
3 {
4     // type alias the C way: (MidPointType is a pointer to this type)
5     typedef double(*MidPointType)(const double a, const double b, const int n);
6
7     // type alias the C++ way: (MidPointType is a pointer to this type)
8     using MidPointType = double(*)(const double a, const double b, const int n);
9
10    // use the new type alias in your code:
11    MidPointType fcn_pointer = &mid_point; // note the difference here!
12    // double result = fcn_pointer(...);
13
14    return 0;
15 }
```

# C++ FUNCTIONALS AND LAMBDA

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

```
1 #include <functional>
2 extern double mid_point(const double a, const double b, const int n);
3 int main(void)
4 {
5     std::function f = mid_point;
6
7     // use the functional in your code
8     // double result = f(...);
9
10    return 0;
11 }
```

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):

```
1 #include <functional>
2 extern double mid_point(const double a, const double b, const int n);
3 int main(void)
4 {
5     // alternatively specify the function type
6     std::function<double(const double a, const double b, const int n)> f = mid_point;
7
8     // use the functional in your code
9     // double result = f(...);
10
11     return 0;
12 }
```

# 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:

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

And the same in C++:

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

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

# C++ LAMBDA'S

```
1 #include <cmath>
2 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:

```
1 #include <functional>
2 #include <cmath>
3 double mid_point(const std::function<double(double)> &f,
4                 const double a, const double b, const int n)
5 {
6     const double dx = (b - a) / n; // integration interval
7     double sum = 0;                // summation variable (must be initialized!)
8     for (int i = 0; i < n; ++i) {
9         sum += f((i + 0.5) * dx); // f can be any function with a scalar argument
10    }
11    return dx * sum;
12 }
```

We can use it with a lambda function:

```
1 int main(void)
2 {
3     auto f = [](double x) { return std::sin(x) * std::cos(x); };
4     // now use mid_point to integrate any integrand f(x)
5     double result = mid_point(f, 0, 2 * M_PI, 100);
6     return 0;
7 }
```