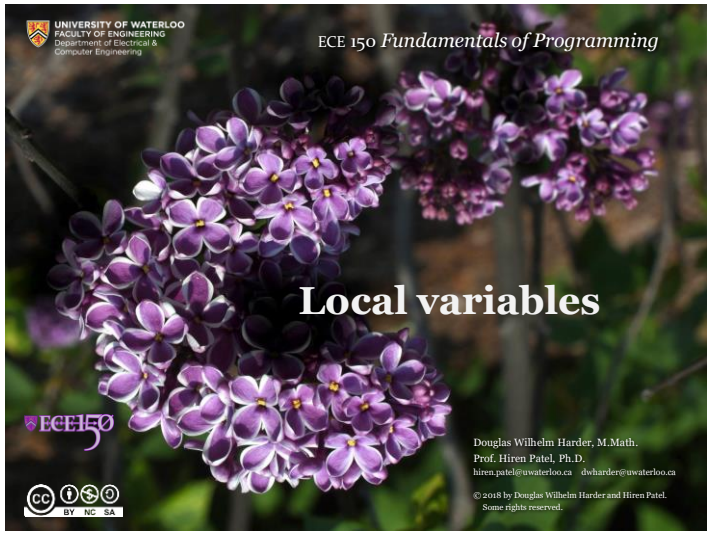

Outline

- In this lesson, we will:
  - Introduce the need for temporary memory storage
    - Local variables
  - Describe initialization of local variables
  - Learn the scope of local variables
  - Describe assigning to local variables
    - Use swap as an example
  - See what is not assignable
  - Look at the automatic assignment operators
    - This includes auto-increment and -decrement operators

Background

- Within a function, we have already seen one category of variable: the function parameters
  - The parameter takes the value of the argument passed to the function
- It may, however, be necessary for the function to temporarily store data while it is being executed
  - The function will temporarily store data while it calculates what the function should return
  - When the function is finished, this temporary data is no longer needed
  - We call such temporary data *local variables*

Example

- Suppose we want to author a function that returns the maximum root of a quadratic:

```
//                                                  2
// Return the maximum root of the polynomial ax  + bx + c
double max_root( double a, double b, double c );
```

- To determine this, we determine the largest of the roots:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Example

- Here is an inefficient implementation:

```
//                                              2
// Return the maximum root of the polynomial ax  + bx + c
double max_root( double a, double b, double c ) {
    assert( (a != 0.0) && ((b*b - 4.0*a*c) >= 0.0) );

    return max( (-b + std::sqrt(b*b - 4.0*a*c))/(2*a),
                (-b - std::sqrt(b*b - 4.0*a*c))/(2*a) );
}
```

- Why is this inefficient?
  - We are calculating `b*b - 4.0*a*c` three times
  - We are calculating `2.0*a` twice

Example

- Whatever value `b*b - 4.0*a*c` evaluates to, we need to store that value temporarily in memory
  - We must specify the type of this temporary value
  - Later, we will refer to that stored value with an identifier
  - The identifier cannot be a keyword and should not be a reserved word

- Declaration and initialization of local variables:

```
double two_a{2.0*a};
double disc{b*b - 4.0*a*c};
```

The type — The identifier — The initial value

Example

- We can temporarily store the values in a *local variables*:

`two_a` *and* `disc`

```
//                                              2
// Return the maximum root of the polynomial ax  + bx + c
double max_root( double a, double b, double c ) {
    // Calculate the discriminant
    double two_a{2.0*a};
    double disc{b*b - 4.0*a*c};

    assert( (a != 0.0) && (disc >= 0.0) );

    double sqrt_disc{std::sqrt( disc )};

    return max( (-b + sqrt_disc)/two_a,
                (-b - sqrt_disc)/two_a );
}
```


Declaring local variables with initial values

- Any identifier that is not a keyword or a reserved word may be used as a local variable
  - Just like the compiler must know the type of a parameter, you must also specify the type of a local variable
  - The initial value is specified in braces
    - The initial value of an integer type or floating-point type can be any arithmetic expression
    - The initial value of a Boolean type (`bool`) can be any logical expression
  - Examples:

```
unsigned int count{0};
double PI{3.1415926535897932};
bool found{false};
```

Casting initial values

- Suppose the initial value is the wrong type:

```
double x{42};                // treated as 42.0
int pi{3.1415926535897932};  // treated as 3
```

- Fortunately, there is a compiler warning for the second:

```
example.cpp: In function 'int main()':
example.cpp:10:27: warning: narrowing conversion of '3.1415926535897931e+0'
from 'double' to 'int' inside { } [-Wnarrowing]
  int pi{3.1415926535897932};
                           ^
```

  - It still compiles and produces an executable file

- For `bool`,
  - An integer `0` or floating-point `0.0` is interpreted as `false`
  - Any other integer or floating-point value is interpreted as `true`

Examples

- Consider this program that gives every item a default value

```
#include <iostream>
void with_initial_values( int m );
int main();

void with_initial_values( int m ) {
    int n{m + 5};
    double x{2*m + 1.5};
    bool b{m};
    std::cout << n << ", " << x << ", " << b << std::endl;
}

int main() {
    with_initial_values( 42 );
    return 0;
}
```

The output is `47, 85.5, 1`

Declaring local variables with default initial values

- Every type has a default initial value:
  - The default value for an integer type is 0
  - The default value for a floating-point number is 0.0
  - The default value of a Boolean type is `false`

- If you want the default type, just use empty braces
  - Examples:

```
int count{};         // 'count' is initialized to 0
double sum{};        // 'sum' is initialized to 0.0
bool found{};        // 'found' is initialized to 'false'
```

- It is always clearer for the reader if you provide the initial value
  - It is clearer even if the initial value is the default

Examples

- Consider this program:

```
#include <iostream>
void with_default_values();
int main();

void with_default_values() {
    int n{};
    double x{};
    bool b{};
    std::cout << n << ", " << x << ", " << b << std::endl;
}

int main() {
    with_default_values();
    return 0;
}
```

The output is `0, 0, 0`

Declaring local variables with no initial values

- It is even possible to not initialize a local value
  - Examples:

```
int count;
double sum;
bool found;
```

  - Whatever `0`s and `1`s are there are interpreted as the type you specified
  - Often, but not always, this will be `0`, `0.0` or `false`
- Almost always, this will be a bad idea
  - On one platform, the default value may be consistently zero
  - On another, under different circumstances, the default value may be some random value

Examples

- Now, consider this program:

```
#include <iostream>
void with_no_initial_values();
int main();

void with_no_initial_values() {
    int n;
    double x;
    bool b;
    std::cout << n << ", " << x << ", " << b << std::endl;
}

int main() {
    with_no_initial_values();
    std::cout << "Pi = " << 3.14 << std::endl;
    with_no_initial_values();

    return 0;
}
```


Examples

- Running this on *ecelinux* yields:

```
4196861, 9.88131e-324, 0
Pi = 3.14
6295680, 2.07323e-317, 0
```

- Running this on *cpp.sh* yields a different result

```
0, 0, 0
Pi = 3.14
0, 0, 0
```

- Running this on a Windows computer yields yet a different result

```
1627411690, 2.122e-314, 0
Pi = 3.14
1817066142, 5.26696e+213, 97
```


Warning on initial values

- Apart from an embedded real-time systems where the wasted time of an unnecessary initialization may actually cause a signification delay...

...always initialize your variables, even if it is with the default.

- If you think you have a scenario where it is not necessary...

...document why in the comments.

Multiple declarations

- Within the same block of statements, you may never declare a local variable more than once
  - The compiler will return an error—even if the value is the same
  - For example:

```
int main() {
    double x{3.2};
    double x{3.2};

    return 0;
}
```

  - The error is

```
example.cpp: In function 'int main()':
example.cpp:3:12: error: redeclaration of 'double x'
     double x{3.2};
            ^
example.cpp:2:12: error: 'double x' previously declared here
     double x{3.2};
            ^
```


Scope

- Local variables are meant to be temporary
  - They have a well-defined *lifetime*
- The lifetime of a parameter is the function definition
  - The block of statements defining the function
    - The statements wrapped in braces { } in the function definition
  - This lifetime is defined by the programming language and enforced by the compiler
- The lifetime of a local variable
  - Starts at the point at which it is declared, and
  - Ends at the end of the inner-most block of statements in which it is defined

Example

- This function only prints those local variables that are *in scope*:

```
void f( int x ) {
    int a{0};
    if ( x < a ) {
        int b{1};
        std::cout << a << "," << b << std::endl;

        if ( a + b <= x ) {
            int c{2};
            std::cout << a << "," << b << "," << c << std::endl;
        }

        int d{3};
        std::cout << a << "," << b << "," << d << std::endl;

        if ( x > a + b - d ) {
            int e{4};
            std::cout << a << "," << b << "," << d << "," << e << std::endl;
        }

        int f{5};
        std::cout << a << "," << b << "," << d << "," << f << std::endl;
    }

    std::cout << a << std::endl;
}
```


Example

- The local variable `'a'` is in scope throughout the function

```
void f( int x ) {
    int a{0};
    if ( x < a ) {
        int b{1};
        std::cout << a << "," << b << std::endl;

        if ( a + b <= x ) {
            int c{2};
            std::cout << a << "," << b << "," << c << std::endl;
        }

        int d{3};
        std::cout << a << "," << b << "," << d << std::endl;

        if ( x > a + b - d ) {
            int e{4};
            std::cout << a << "," << b << "," << d << "," << e << std::endl;
        }

        int f{5};
        std::cout << a << "," << b << "," << d << "," << f << std::endl;
    }

    std::cout << a << std::endl;
}
```

Example

- The local variable `'b'` is in scope only in the consequent block

```
void f( int x ) {
    int a{0};
    if ( x < a ) {
        int b{1};
        std::cout << a << "," << b << std::endl;

        if ( a + b <= x ) {
            int c{2};
            std::cout << a << "," << b << "," << c << std::endl;
        }

        int d{3};
        std::cout << a << "," << b << "," << d << std::endl;

        if ( x > a + b - d ) {
            int e{4};
            std::cout << a << "," << b << "," << d << "," << e << std::endl;
        }

        int f{5};
        std::cout << a << "," << b << "," << d << "," << f << std::endl;
    }

    std::cout << a << std::endl;
}
```


Example

- The local variable `'c'` is even more restricted

```
void f( int x ) {
    int a{0};
    if ( x < a ) {
        int b{1};
        std::cout << a << "," << b << std::endl;

        if ( a + b <= x ) {
            int c{2};
            std::cout << a << "," << b << "," << c << std::endl;
        }

        int d{3};
        std::cout << a << "," << b << "," << d << std::endl;

        if ( x > a + b - d ) {
            int e{4};
            std::cout << a << "," << b << "," << d << "," << e << std::endl;
        }

        int f{5};
        std::cout << a << "," << b << "," << d << "," << f << std::endl;
    }

    std::cout << a << std::endl;
}
```


Example

- The local variable `'d'` is restricted to the end of this consequent block

```
void f( int x ) {
    int a{0};
    if ( x < a ) {
        int b{1};
        std::cout << a << "," << b << std::endl;

        if ( a + b <= x ) {
            int c{2};
            std::cout << a << "," << b << "," << c << std::endl;
        }

        int d{3};
        std::cout << a << "," << b << "," << d << std::endl;

        if ( x > a + b - d ) {
            int e{4};
            std::cout << a << "," << b << "," << d << "," << e << std::endl;
        }

        int f{5};
        std::cout << a << "," << b << "," << d << "," << f << std::endl;
    }

    std::cout << a << std::endl;
}
```


Example

- The local variable `'e'` is restricted to the block in which it is declared

```
void f( int x ) {
    int a{0};
    if ( x < a ) {
        int b{1};
        std::cout << a << "," << b << std::endl;

        if ( a + b <= x ) {
            int c{2};
            std::cout << a << "," << b << "," << c << std::endl;
        }

        int d{3};
        std::cout << a << "," << b << "," << d << std::endl;

        if ( x > a + b - d ) {
            int e{4};
            std::cout << a << "," << b << "," << d << "," << e << std::endl;
        }

        int f{5};
        std::cout << a << "," << b << "," << d << "," << f << std::endl;
    }

    std::cout << a << std::endl;
}
```

Example

- The local variable `'f'` is only defined to the end of its block

```
void f( int x ) {
    int a{0};
    if ( x < a ) {
        int b{1};
        std::cout << a << "," << b << std::endl;

        if ( a + b <= x ) {
            int c{2};
            std::cout << a << "," << b << "," << c << std::endl;
        }

        int d{3};
        std::cout << a << "," << b << "," << d << std::endl;

        if ( x > a + b - d ) {
            int e{4};
            std::cout << a << "," << b << "," << d << "," << e << std::endl;
        }

        int f{5};
        std::cout << a << "," << b << "," << d << "," << f << std::endl;
    }

    std::cout << a << std::endl;
}
```


Parameters as local variables

- You can consider a parameter to be a local variable of a function where:
  - The scope of the parameter is the entire function
  - The parameter is initialized not within the function but by the value of the argument passed when the function is called
- The function body cannot have a local variable declared that has the same name as a parameter

Using the same identifier

- There are specific circumstances where you can declare two different local variables with the same identifier
  - In general, this is a bad idea, and leads to poor programming practices
  - One situation where it is permissible is if:
    - A local variable is required in two or more consequent or alternative blocks of a (possibly cascading) conditional statement
    - That local variable serves the same or similar purposes in each of those blocks
- Otherwise, please avoid using the same identifier twice in the same function

Scope

- If you define a local variable in one function, you cannot access that local variable from another function
  - The variable is *local* to the function it is defined in
- Reminder:
  - If another function needs a value stored in a local variable, it should be passed as an argument to the function being called
  - If a local variable is required by the function that called the function the local variable is defined in, the local variable should be returned

Assigning to local variables

- After we have initialized a local variable, it may be necessary to change or update its value
  - This is done through *assignment* using the binary assignment operator =

```
variable_name = expression;
```

- Important: read this as

  "The variable **is assigned the value of the right-hand** expression"

- As before,
  - a numeric variable can be assigned an arithmetic expression
  - a Boolean variable can be assigned a logical expression
- If the type of the expression differs, it will be appropriately cast

Assigning to local variables

- The left-hand side of an assignment operator is never evaluated
  - If you write

```
m = 3;
```

  this says "assign the local variable m the value of 3."

- The right-hand side, however, is always evaluated first before the assignment is made; for example:

```
int m{42};
m = m + 1;
std::cout << m << std::end;
```

m is now assigned the value 43
– the value 42 is lost

Assigning to local variables

- Once a local variable is assigned a new value, any reference to that local variable will result in that new value until the local variable goes out of scope
  - Local variables can be assigned new values arbitrarily many times
- Once a local variable is assigned a new value, the previous value is lost—there is no way to recover it unless you saved it elsewhere:

```
int main() {
    int n{10};
    int m{n};
    n = 17;
    std::cout << m << ", " << n << std::endl;

    return 0;
}
```

This prints `10, 17`

Assigning to local variables

- Remember that a local variable is like a box on a piece of paper where you record something
  - Every time you refer to that box, you get exactly what is there
  - If you change what is in the box, you then get the new value
  - If you copy that value to another box, the value in that box hasn't changed

Swapping two values

- Suppose we have to local variables assigned 10 and 20, respectively

```
int m{10};
int n{20};
```

- How can we swap those two values?
  - That is, how do we end up with `n` being assigned `10` and `m` assigned `20`?
- You cannot simply assign one to the other:

```
int m{10};
int n{20};
m = n;
```

  - Now, both `m` and `n` have the value `20`, and the value `10` is lost

Swapping two values

- We require a temporary variable:

```
int m{10};
int n{20};

int tmp{m};
m = n;
n = tmp;
```


What is not assignable?

- To this point, only variables can appear on the left-hand side:
  - Consider this program:

```
int main() {
        int n{10};
        10 = 17;
        10 = n;
        n + 1 = 20;
        return 0;
}
```

```
example.cpp: In function 'int main()':
example.cpp:4:5: error: lvalue required as left operand of assignment
  10 = 17;
     ^
example.cpp:5:5: error: lvalue required as left operand of assignment
  10 = n;
     ^
example.cpp:6:8: error: lvalue required as left operand of assignment
  n + 1 = 20;
        ^
```


What is not assignable?

- The error messages constantly refer to requiring an *lvalue*
  - Think if this as a *l*eft-hand side *value*
  - The construct to the left-hand side of the assignment operator must be something to which an assignment can be made
- The compiler does not solve problems for you:
  - For example, this does not assign the value `43` to `n`:

```
n - 1 = 42;
```

- Always remember: `=` is the *assignment operator*; it is **not** an equals sign

What does assignment evaluate to?

- Recall that every operator we have examined so far has a return value—what does the assignment operator = evaluate to?

```
#include <iostream>
int main();

int main() {
    int m{0};

    std::cout << (m = 10) << std::endl;
    std::cout << m << std::endl;

    return 0;
}
```

In both cases, the value `10` is printed

What does assignment evaluate to?

- Assignment evaluates to the value that is assigned
- This allows the following construct:

```
double a{3}, b{4}, c{5};
a = b = c = 10;

std::cout << a << "," << b << "," << c << std::endl;
```

- Think of this as — All three values are assigned `10`

```
a = (b = (c = 10));
```

  - The local variable `c` is assigned `10`, and `10` is the result
  - Next, the local variable `b` is assigned `10`, and `10` is the result
  - Finally, the local variable `a` is assigned `10`, and `10` is the result

Do you really understand operators?

- What is assigned to `a`, `b` and `c` after this?

```
double a{3}, b{4}, c{5};
std::cout << ((a = 3*(b = 2*(c = a + 2*b*c - 1) + 7) + 2*b - 8*c + 131) - 100)
          << std::endl;
std::cout << a << ", " << b << ", " << c << std::endl;
```

- Important: Just because you can write code like this, and just because you want to write code like this, please do not...

...you'll get a 0 in this course

Only ever use assignment as the first operator in a statement

Do you really understand operators?

- If you had to write code like this, is this not clearer?

```
double a{3}, b{4}, c{5};
c = a + 2*b*c - 1;
b = 2*c + 7;
a = 3*b + 2*b - 8*c + 131;
std::cout << (a - 100) << std::endl;
std::cout << a << ", " << b << ", " << c << std::endl;
```

Automatic assignment operators

- The C++ programming language has further binary *automatic assignment* operators that provide syntactic short-cuts :

Assignment	Automatic assignment	Name
`a = a + 32`	`a += 32`	auto-addition
`b = b - 41`	`b -= 41`	auto-subtraction
`c = 2*c`	`c *= 2`	auto-multiplication
`d = d/10`	`d /= 10`	auto-division
`e = e % 100`	`e %= 100`	auto-remainder

- For each of these automatic assignment operators, the left-hand side must be assignable (at this point, a variable)
  - Here, "automatic" harkens back to its Greek root: *automatos* meaning "acting of itself"

Automatic assignment operators

- Each of these binary operator evaluates to a reference to the left-hand variable being assigned to
  - This result can further be assigned to…
  - What is the output of this program?

```
#include <iostream>
int main();

int main() {
    int a{2}, b{3};

    a += ((b += 5*7 + 11) *= 2) + 13;
    std::cout << a << ", " << b << std::endl;

    return 0;
}
```


Automatic assignment operators

- Please, don't use code like this: we present this so that you understand how C++ works
  - Always use these automatic assignment operators as a single statement

```
int a{2}, b{3};

b += 5*7 + 11;
b *= 2;
a += b + 13;
```

- Originally, these automatic assignment operators helped guide the compiler in generating assembly code
  - Today's optimizers don't require this anymore:

Don't be fancy, be clear

Auto-increment and -decrement operators

- C++ has a more compact unary automatic assignment operators:

```
a = a + 1;      a += 1;      ++a;      a++;
b = b - 1;      b -= 1;      --b;      b--;
```

- These are the *unary automatic-increment* and *automatic-decrement* operators

	Evaluated result	Name
`++a`	adds 1 to `'a'` and evaluates to the variable `'a'` – with its new value	*auto-pre-increment*
`a++`	adds 1 to `'a'` but evaluates to the original value `'a'`	*auto-post-increment*
`--b`	subtracts 1 from `'b'` and evaluates to the variable `'b'` – with its new value	*auto-pre-decrement*
`b--`	subtracts 1 from `'b'` but evaluates to the original value `'b'`	*auto-post-decrement*

Auto-increment and -decrement operators

- Again, these originally guided the compiler
  - Today's optimizers don't require this anymore—don't be fancy, be clear
- These unary automatic operators should always be used as a single statement
  - In class, we will always use the *pre-* form: `++a` and `--b`

Thought experiment…

- What is the output of this program?

```
#include <iostream>
void f( int m );
int main();

void f( int m ) {
    int n;  // Uninitialized!
    std::cout << n << std::endl;
    n = m;
}

int main() {
    std::cout << "Hello world!" << std::endl;
    f( 42 );
    f( 91 );
    f( 150 );
    return 0;
}
```

The output is:

```
0
42
91
```

In an upcoming lesson, we will see why

Summary

- Following this lesson, you now:
  - Understand the need for local variables
  - Understand how to initialize local variables and why this is important
  - Know that the scope of the local variable is within the block of statements
  - Understand how to assign to local variables
  - Know so far what is assignable and what is not
  - Know the automatic assignment operators and their behavior

References

[1] Wikipedia, https://en.wikipedia.org/wiki/Local_variable

[2] CodesDope https://www.codesdope.com/cpp-scope-of-variables/

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.



Disclaimer