Basic C++ Syntax
Overview

Common set of basic features shared by a wide range of programming languages

- Built-in types (integers, characters, floating point numbers, etc.)
- Variables (“names” for entities)
- Expressions and statements to manipulate values of variables
- Control-flow constructs (if, for, etc.)
- Functions, i.e. units of computation

Supplemented by additional functionality (covered later)

- Programmer-defined types (struct, class, etc.)
- Library functions
The C++ Reference Documentation

C++ is in essence a simple language

- Limited number of basic features and rules
- **But:** There is a corner case to most features and an exception to most rules
- **But:** Some features and rules are rather obscure

These slides will necessarily be inaccurate or incomplete at times

- [https://en.cppreference.com/w/cpp](https://en.cppreference.com/w/cpp) provides an excellent and complete reference documentation of C++
- Every C++ programmer should be able to read and understand the reference documentation
- Slides that directly relate to the reference documentation contain the symbol with a link to the relevant webpage in the slide header

Look at these links and familiarize yourself with the reference documentation!
Fundamental Types

C++ defines a set of primitive types

- Void type
- Boolean type
- Integer types
- Character types
- Floating point types

All other types are composed of these fundamental types in some way
Void Type

The void type has no values

- Identified by the C++ keyword `void`
- No objects of type `void` are allowed
- Mainly used as a return type for functions that do not return any value
- Pointers to `void` are also permitted (more details later)

```c++
void* pointer; // OK: pointer to void
void object; // ERROR: object of type void
void doSomething() { // OK: void return type
    // do something important
}
```
Boolean Type

The boolean type can hold two values

- Identified by the C++ keyword `bool`
- Represents the truth values `true` and `false`
- Quite frequently obtained from implicit automatic type conversion (more details later)

```cpp
bool condition = true;
// ...
if (condition) {
  // ...
}
```
Integer Types (1)

The integer types represent integral values

- Identified by the C++ keyword `int`
- Some properties of integer types can be changed through modifiers
- `int` keyword may be omitted if at least one modifier is used

Signedness modifiers

- `signed` integers will have signed representation (i.e. they can represent negative numbers)
- `unsigned` integers will have unsigned representation (i.e. they can only represent non-negative numbers)

Size modifiers

- `short` integers will be optimized for space (at least 16 bits wide)
- `long` integers will be at least 32 bits wide
- `long long` integers will be at least 64 bits wide
Integer Types (2)

Modifiers and the `int` keyword can be specified in any order

```c
// a, b, c and d all have the same type
unsigned long long int a;
unsigned long long b;
long unsigned int long c;
long long unsigned d;
```

By default integers are `signed`, thus the `signed` keyword can be omitted

```c
// e and f have the same type
signed int e;
int f;
```

By convention modifiers are ordered as follows

1. Signedness modifier
2. Size modifier
3. `int`
# Integer Type Overview

Overview of the integer types as specified by the C++ standard

<table>
<thead>
<tr>
<th>Canonical Type Specifier</th>
<th>Minimum Width</th>
<th>Minimum Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>16 bit</td>
<td>$-2^{15}$ to $2^{15} - 1$</td>
</tr>
<tr>
<td>unsigned short</td>
<td></td>
<td>$0$ to $2^{15} - 1$</td>
</tr>
<tr>
<td>int</td>
<td>32 bit</td>
<td>$-2^{31}$ to $2^{31} - 1$</td>
</tr>
<tr>
<td>unsigned</td>
<td></td>
<td>$0$ to $2^{32} - 1$</td>
</tr>
<tr>
<td>long</td>
<td>32 bit</td>
<td>$-2^{31}$ to $2^{31} - 1$</td>
</tr>
<tr>
<td>unsigned long</td>
<td></td>
<td>$0$ to $2^{32} - 1$</td>
</tr>
<tr>
<td>long long</td>
<td>64 bit</td>
<td>$-2^{63}$ to $2^{63} - 1$</td>
</tr>
<tr>
<td>unsigned long long long</td>
<td></td>
<td>$0$ to $2^{64} - 1$</td>
</tr>
</tbody>
</table>

The exact width of integer types is **not** specified by the standard!
Fixed-Width Integer Types

Sometimes we need integer types with a guaranteed width

- Use fixed-width integer types defined in `<cstdint>` header
- `int8_t`, `int16_t`, `int32_t` and `int64_t` for signed integers of width 8, 16, 32 or 64 bit, respectively
- `uint8_t`, `uint16_t`, `uint32_t` and `uint64_t` for unsigned integers of width 8, 16, 32 or 64 bit, respectively

Only defined if the C++ implementation directly supports the type

```cpp
#include <cstdint>

long    a;  // may be 32 or 64 bits wide
int32_t b;  // guaranteed to be 32 bits wide
int64_t c;  // guaranteed to be 64 bits wide
```
Character Types

Character types represent character codes and (to some extent) integral values

- Identified by C++ keywords `signed char` and `unsigned char`
- Minimum width is 8 bit, large enough to represent UTF-8 eight-bit code units
- The C++ type `char` may either be `signed char` or `unsigned char`, depending on the implementation
- `signed char` and `unsigned char` are sometimes used to represent small integral values

Larger UTF characters are supported as well

- `char16_t` for UTF-16 character representation
- `char32_t` for UTF-32 character representation
Floating Point Types

Floating point types of varying precision

- `float` usually represents IEEE-754 32 bit floating point numbers
- `double` usually represents IEEE-754 64 bit floating point numbers
- `long double` is a floating point type with extended precision (varying width depending on platform and OS, usually between 64 bit and 128 bit)

Floating point types may support special values

- Infinity
- Negative zero
- Not-a-number
Implicit Conversions (1)

Type conversions may happen automatically
- If we use an object of type A where an object of type B is expected
- Exact conversion rules are highly complex (full details in the reference documentation)

Some common examples
- If one assigns an integral type to bool the result is false if the integral value is 0 and true otherwise
- If one assigns bool to an integral type the result is 1 if the value is true and 0 otherwise
- If one assigns a floating point type to an integral type the value is truncated
- If one assigns an out-of-range value to an unsigned integral type, the result is the original value modulo the maximum value of the target type
Implicit Conversions (2)

Example

```
uint16_t i = 257;
uint8_t j = i;  // j is 1

if (j) {
    /* executed if j is not zero */
}
```
Undefined Behavior (1)

In some situations the behavior of a program is not well-defined
- E.g. overflow of an unsigned integer is well-defined (see previous slide)
- **But:** Signed integer overflow results in **undefined behavior**
- We will encounter undefined behavior every once in a while

Undefined behavior falls outside the specification of the C++ standard
- The compiler is allowed to do anything when it encounters undefined behavior
- Fall back to some sensible default behavior
- Do nothing
- Print 42
- Do anything else you can think of

A C++ program must never contain undefined behavior!
Undefined Behavior (2)

Example

foo.cpp

```cpp
int foo(int i) {
    if ((i + 1) > i)
        return 42;
    return 123;
}
```

foo.o

```assembly
foo(int):
    movl $42, %eax
    retq
```
Variables need to be defined before they can be used

- Simple declaration: Type specifier followed by comma-separated list of declarators (variable names) followed by semicolon
- Variable names in a simple declaration may optionally be followed by an initializer

```c
void foo() {
    unsigned i = 0, j;
    unsigned meaningOfLife = 42;
}
```
Variable Initializers (1)

Initialization provides an initial value at the time of object construction

1. `variableName(<expression>)`
2. `variableName = <expression>`
3. `variableName{<expression>}`

Important differences

- Options 1 and 2 simply assign the value of the expression to the variable, possibly invoking implicit type conversions
- Option 3 results in a compile error if implicit type conversions potentially result in loss of information

A declaration may contain no initializer

- Non-local variables are default-initialized (to zero for built-in types)
- Local variables are usually **not** default-initialized

Accessing an uninitialized variable is **undefined behavior**
Variable Initializers (2)

```cpp
double a = 3.1415926;
double b(42);
unsigned c = a; // OK: c == 3
unsigned d(b); // OK: d == 42
unsigned e{a}; // ERROR: potential information loss
unsigned f{b}; // ERROR: potential information loss
```

Initializers may be arbitrarily complex expressions

```cpp
double pi = 3.1415926, z = 0.30, a = 0.5;
double volume(pi * z * z * a);
```
Integer Literals

Integer literals represent constant values embedded in the source code

- Decimal: 42
- Octal: 052
- Hexadecimal: 0x2a
- Binary: 0b101010

The following suffixes may be appended to a literal to specify its type

- **unsigned** suffix: 42u or 42U
- Long suffixes:
  - **long** suffix: 42l or 42L
  - **long long** suffix: 42ll or 42LL
- Both suffixes can be combined, e.g. 42ul, 42ull

Single quotes may be inserted between digits as a separator

- e.g. 1'000'000'000'000ull
- e.g. 0b0010'1010
Floating-point literals

Floating-point literals represent constant values embedded in the source code

- Without exponent: \( 3.1415926 \), \(.5\)
- With exponent: \(1e9\), \(3.2e20\), \(.5e-6\)

One of the following suffixes may be appended to a literal to specify its type

- \text{float} suffix: \(1.0f\) or \(1.0F\)
- \text{long double} suffix: \(1.0l\) or \(1.0L\)

Single quotes may be inserted between digits as a separator

- e.g. \(1'000.000'001\)
- e.g. \(.141'592e12\)
Character Literals

Character literals represent constant values embedded in the source code

- Any character from the source character set except single quote, backslash and newline, e.g. 'a', 'b', '€'
- Escape sequences, e.g. '\', '\\', '\n', '\u1234'

One of the following prefixes may be prepended to a literal to specify its type

- UTF-8 prefix: u8'a', u8'b'
- UTF-16 prefix: u'a', u'b'
- UTF-32 prefix: U'a', U'b'
Any type $T$ in C++ (except function and reference types) can be *cv-qualified*

- **const-qualified**: `const T`
- **volatile-qualified**: `volatile T`
- cv-qualifiers can appear in any order, before or after the type

**Semantics**

- **const** objects cannot be modified
- Any read or write access to a **volatile** object is treated as a visible side effect for the purposes of optimization
- **volatile** should be avoided in most cases (it is likely to be deprecated in future versions of C++)
- Use *atomics* instead (more details later)
only code that contributes to observable side-effects is emitted

```c
int main() {
    int a = 1;  // will be optimized out
    int b = 2;  // will be optimized out
    volatile int c = 42;
    volatile int d = c + b;
}
```

possible x86-64 assembly (compiled with -O1)

```
main:
    movl  $42, -4(%rsp)       # volatile int c = 42
    movl  -4(%rsp), %eax     # volatile int d = c + b;
    addl  $2, %eax           # volatile int d = c + b;
    movl  %eax, -8(%rsp)     # volatile int d = c + b;
    movl  %eax, -8(%rsp)     # volatile int d = c + b;
    movl  $0, %eax           # implicit return 0;
    ret
```
Expression Fundamentals

C++ provides a rich set of operators

- Operators and operands can be composed into expressions
- Most operators can be overloaded for custom types (more details later)

Fundamental expressions

- Variable names
- Literals

Operators act on a number of operands

- Unary operators: E.g. negation (−), address-of (&), dereference (*)
- Binary operators: E.g. equality (==), multiplication (*)
- Ternary operator: \( a \ ? \ b \ : \ c \)
Value Categories

Each expression in C++ is characterized by two independent properties

• Its type (e.g. unsigned, float)
• Its value category
• Operators may require operands of certain value categories
• Operators result in expressions of certain value categories

Broadly (and inaccurately) there are two value categories: lvalues and rvalues

• lvalues refer to the identity of an object
• rvalues refer to the value of an object
• Modifiable lvalues can appear on the left-hand side of an assignment
• lvalues and rvalues can appear on the right-hand side of an assignment

C++ actually has a much more sophisticated taxonomy of expressions

• Will (to some extent) become relevant later during the course
## Arithmetic Operators (1)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+a</td>
<td>Unary plus</td>
</tr>
<tr>
<td>-a</td>
<td>Unary minus</td>
</tr>
<tr>
<td>a + b</td>
<td>Addition</td>
</tr>
<tr>
<td>a - b</td>
<td>Subtraction</td>
</tr>
<tr>
<td>a * b</td>
<td>Multiplication</td>
</tr>
<tr>
<td>a / b</td>
<td>Division</td>
</tr>
<tr>
<td>a % b</td>
<td>Modulo</td>
</tr>
<tr>
<td>~a</td>
<td>Bitwise NOT</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a ^ b</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>a &lt;&lt; b</td>
<td>Bitwise left shift</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>Bitwise right shift</td>
</tr>
</tbody>
</table>

**C++** arithmetic operators have the usual semantics
Incorrectly using the arithmetic operators can lead to undefined behavior, e.g.

- Signed overflow (see above)
- Division by zero
- Shift by a negative offset
- Shift by an offset larger than the width of the type
## Logical and Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>Logical NOT</td>
</tr>
<tr>
<td>a &amp;&amp; b</td>
<td>Logical AND (short-circuiting)</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a == b</td>
<td>Equal to</td>
</tr>
<tr>
<td>a != b</td>
<td>Not equal to</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>Less than</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>Greater than</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

C++ logical and relational operators have the usual semantics.
# Assignment Operators (1)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
<td>Simple assignment</td>
</tr>
<tr>
<td>a += b</td>
<td>Addition assignment</td>
</tr>
<tr>
<td>a -= b</td>
<td>Subtraction assignment</td>
</tr>
<tr>
<td>a *= b</td>
<td>Multiplication assignment</td>
</tr>
<tr>
<td>a /= b</td>
<td>Division assignment</td>
</tr>
<tr>
<td>a %= b</td>
<td>Modulo assignment</td>
</tr>
<tr>
<td>a &amp;= b</td>
<td>Bitwise AND assignment</td>
</tr>
<tr>
<td>a</td>
<td>= b</td>
</tr>
<tr>
<td>a ^= b</td>
<td>Bitwise XOR assignment</td>
</tr>
<tr>
<td>a &lt;&lt;= b</td>
<td>Bitwise left shift assignment</td>
</tr>
<tr>
<td>a &gt;&gt;= b</td>
<td>Bitwise right shift assignment</td>
</tr>
</tbody>
</table>

**Notes**

- The left-hand side of an assignment operator must be a modifiable lvalue
- For built-in types `a OP= b` is equivalent to `a = a OP b` except that `a` is only evaluated once
Assignment Operators (2)

The assignment operators return a reference to the left-hand side

```c
unsigned a, b, c;
a = b = c = 42;  // a, b, and c have value 42
```

Usually rarely used, with one exception

```c
if (unsigned d = computeValue()) {
    // executed if d is not zero
} else {
    // executed if d is zero
}
```
Increment and Decrement Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>++a</td>
<td>Prefix increment</td>
</tr>
<tr>
<td>--a</td>
<td>Prefix decrement</td>
</tr>
<tr>
<td>a++</td>
<td>Postfix increment</td>
</tr>
<tr>
<td>a--</td>
<td>Postfix decrement</td>
</tr>
</tbody>
</table>

Return value differs between prefix and postfix variants

- Prefix variants increment or decrement the value of an object and return a reference to the result (more details later)
- Postfix variants create a copy of an object, increment or decrement the value of the original object, and return the copy
Ternary Conditional Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a?b:c</td>
<td>Conditional operator</td>
</tr>
</tbody>
</table>

Semantics

- a is evaluated and converted to `bool`
- If the result was `true`, b is evaluated
- Otherwise c is evaluated

The type and value category of the resulting expression depend on the operands

```cpp
int n = (1 > 2) ? 21 : 42;  // 1 > 2 is false, i.e. n == 42
int m = 42;
((n == m) ? m : n) = 21;    // n == m is true, i.e. m == 21

int k{(n == m) ? 5.0 : 21}; // ERROR: narrowing conversion
((n == m) ? 5 : n) = 21;    // ERROR: assigning to rvalue
```
Precedence and Associativity (1)

How to group multiple operators in one expression?

• Operators with higher precedence bind tighter than operators with lower precedence
• Operators with equal precedence are bound in the direction of their associativity
  • left-to-right
  • right-to-left
• Often grouping is not immediately obvious: Use parentheses judiciously!

Precedence and associativity do not specify evaluation order

• Evaluation order is mostly unspecified
• Generally, it is undefined behavior to refer to and change the same object within one expression
In some situations grouping is obvious

```c
int a = 1 + 2 * 3; // 1 + (2 * 3), i.e. a == 7
```

However, things can get confusing really quickly

```c
int b = 50 - 6 - 2; // (50 - 6) - 2, i.e. b == 42
int c = b & 1 << 4 - 1; // b & (1 << (4 - 1)), i.e. c == 8
```

// real-world examples from libdcrw
```c
diff = ((getbits(len-shl) << 1) + 1) << shl >> 1; // ???
yuv[c] = (bitbuf >> c * 12 & 0xfff) - (c >> 1 << 11); // ???
```

Bugs like to hide in expressions without parentheses

```c
// shift should be 4 if sizeof(long) == 4, 6 otherwise
unsigned shift = 2 + sizeof(long) == 4 ? 2 : 4; // buggy
```
# Operator Precedence Table (1)

<table>
<thead>
<tr>
<th>Prec.</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>::</td>
<td>Scope resolution</td>
<td>left-to-right</td>
</tr>
<tr>
<td>2</td>
<td>a++ a--</td>
<td>Postfix increment/decrement</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td>&lt;type&gt;() &lt;type&gt;{</td>
<td>Functional Cast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a()</td>
<td>Function Call</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a[]</td>
<td>Subscript</td>
<td></td>
</tr>
<tr>
<td></td>
<td>. -&gt;</td>
<td>Member Access</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>++a --a</td>
<td>Prefix increment/decrement</td>
<td>right-to-left</td>
</tr>
<tr>
<td></td>
<td>+a -a</td>
<td>Unary plus/minus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>! ~</td>
<td>Logical/Bitwise NOT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(&lt;type&gt;)</td>
<td>C-style cast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*a</td>
<td>Dereference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;a</td>
<td>Address-of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sizeof</td>
<td>Size-of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>new new[]</td>
<td>Dynamic memory allocation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete delete[]</td>
<td>Dynamic memory deallocation</td>
<td></td>
</tr>
</tbody>
</table>
## Operator Precedence Table (2)

<table>
<thead>
<tr>
<th>Prec.</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>.*  -&gt;*</td>
<td>Pointer-to-member</td>
<td>left-to-right</td>
</tr>
<tr>
<td>5</td>
<td>a*b  a/b  a%b</td>
<td>Multiplication/Division/Remainder</td>
<td>left-to-right</td>
</tr>
<tr>
<td>6</td>
<td>a+b  a-b</td>
<td>Addition/Subtraction</td>
<td>left-to-right</td>
</tr>
<tr>
<td>7</td>
<td>&lt;&lt;  &gt;&gt;</td>
<td>Bitwise shift</td>
<td>left-to-right</td>
</tr>
<tr>
<td>8</td>
<td>&lt;=&gt;</td>
<td>Three-way comparison</td>
<td>left-to-right</td>
</tr>
<tr>
<td>9</td>
<td>&lt;  &lt;=  &gt;  &gt;=</td>
<td>Relational &lt; and &lt;= Relational &gt; and &gt;=</td>
<td>left-to-right</td>
</tr>
<tr>
<td>10</td>
<td>==  !=</td>
<td>Relational = and ≠</td>
<td>left-to-right</td>
</tr>
</tbody>
</table>
### Operator Precedence Table (3)

<table>
<thead>
<tr>
<th>Prec.</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>left-to-right</td>
</tr>
<tr>
<td>12</td>
<td>^</td>
<td>Bitwise XOR</td>
<td>left-to-right</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>14</td>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>left-to-right</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>,</td>
<td>Comma</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td>a?b:c</td>
<td>Ternary conditional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>throw</td>
<td>throw operator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>Direct assignment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+= -=</td>
<td>Compound assignment</td>
<td>right-to-left</td>
</tr>
<tr>
<td></td>
<td>*= /= %=</td>
<td>Compound assignment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;= &gt;&gt;=</td>
<td>=</td>
<td>Compound assignment</td>
</tr>
<tr>
<td></td>
<td>&amp;= ^=</td>
<td>=</td>
<td>Compound assignment</td>
</tr>
</tbody>
</table>
Simple Statements

Declaration statement: Declaration followed by a semicolon

```c
int i = 0;
```

Expression statement: Any expression followed by a semicolon

```c
i + 5; // valid, but rather useless expression statement
foo(); // valid and possibly useful expression statement
```

Compound statement (blocks): Brace-enclosed sequence of statements

```c
{
    // start of block
    int i = 0; // declaration statement
}
    // end of block, i goes out of scope
int i = 1; // declaration statement
```
Names in a C++ program are valid only within their scope

- The scope of a name begins at its point of declaration
- The scope of a name ends at the end of the relevant block
- Scopes may be shadowed resulting in discontiguous scopes (bad practice)

```c
int a = 21;
int b = 0;
{
    int a = 1;       // scope of the first a is interrupted
    int c = 2;
    b = a + c + 39;  // a refers to the second a, b == 42
}
// scope of the second a and c ends
b = a;               // a refers to the first a, b == 21
b += c;              // ERROR: c is not in scope
```
If Statement (1)

Conditionally executes another statement

```c
if (init-statement; condition)
    then-statement
else
    else-statement
```

Explanation

- If `condition` evaluates to `true` after conversion to `bool`, `then-statement` is executed, otherwise `else-statement` is executed.
- Both `init-statement` and the else branch can be omitted.
- If present, `init-statement` must be an expression or declaration statement.
- `condition` must be an expression statement or a single declaration.
- `then-statement` and `else-statement` can be arbitrary (compound) statements.
If Statement (2)

The *init-statement* form is useful for local variables only needed inside the `if`

```c
if (unsigned value = computeValue(); value < 42) {
    // do something
} else {
    // do something else
}
```

Equivalent formulation

```c
{
    unsigned value = computeValue();
    if (value < 42) {
        // do something
    } else {
        // do something else
    }
}
```
If Statement (3)

In nested if-statements, the else is associated with the closest if that does not have an else

```c
// INTENTIONALLY BUGGY!
if (condition0)
    if (condition1)
        // do something if (condition0 && condition1) == true
    else
        // do something if condition0 == false
```

When in doubt, use curly braces to make scopes explicit

```c
// Working as intended
if (condition0) {
    if (condition1)
        // do something if (condition0 && condition1) == true
} else {
    // do something if condition0 == false
}
Switch Statement (1)

Conditionally transfer control to one of several statements

```
switch (init-statement; condition)
  statement
```

Explanation

- `condition` may be an expression or single declaration that is convertible to an enumeration or integral type
- The body of a `switch` statement may contain an arbitrary number of `case constant:` labels and up to one `default:` label
- The constant values for all `case:` labels must be unique
- If `condition` evaluates to a value for which a `case:` label is present, control is passed to the labelled statement
- Otherwise, control is passed to the statement labelled with `default`
- The `break;` statement can be used to exit the `switch`
Switch Statement (2)

Regular example

```cpp
switch (computeValue()) {
    case 21:
        // do something if computeValue() was 21
        break;
    case 42:
        // do something if computeValue() was 42
        break;
    default:
        // do something if computeValue() was != 21 and != 42
        break;
}
```
Switch Statement (3)

The body is executed sequentially until a `break;` statement is encountered

```c
switch (computeValue()) {
    case 21:
    case 42:
        // do something if computeValue() was 21 or 42
        break;
    default:
        // do something if computeValue() was != 21 and != 42
        break;
}
```

Compilers may generate warnings when encountering such fall-through behavior

- Use special `[[fallthrough]]`; statement to mark intentional fall-through
While Loop

Repeatedly executes a statement

```
while (condition)
    statement
```

Explanation

- Executes `statement` repeatedly until the value of `condition` becomes `false`. The test takes place before each iteration.
- `condition` may be an expression that can be converted to `bool` or a single declaration
- `statement` may be an arbitrary statement
- The `break;` statement may be used to exit the loop
- The `continue;` statement may be used to skip the remainder of the body
Do-While Loop

Repeatedly executes a statement

```
do
    statement
while (condition);
```

Explanation

- Executes `statement` repeatedly until the value of `condition` becomes `false`. The test takes place after each iteration.
- `condition` may be an expression that can be converted to `bool` or a single declaration
- `statement` may be an arbitrary statement
- The `break;` statement may be used to exit the loop
- The `continue;` statement may be used to skip the remainder of the body
While vs. Do-While

The body of a do-while loop is executed at least once

```c
unsigned i = 42;

do {
    // executed once
} while (i < 42);

while (i < 42) {
    // never executed
}
```
For Loop (1)

Repeatedly executes a statement

```
for (init-statement; condition; iteration-expression)
    statement
```

Explanation

- Executes *init-statement* once, then executes *statement* and *iteration-expression* repeatedly until *condition* becomes *false*
- *init-statement* may either be an expression or declaration
- *condition* may either be an expression that can be converted to *bool* or a single declaration
- *iteration-expression* may be an arbitrary expression
- All three of the above statements may be omitted
- The *break*; statement may be used to exit the loop
- The *continue*; statement may be used to skip the remainder of the body
For Loop (2)

```cpp
for (unsigned i = 0; i < 10; ++i) {
    // do something
}
for (unsigned i = 0, limit = 10; i != limit; ++i) {
    // do something
}
```

Beware of integral overflows (signed overflows are undefined behavior!)

```cpp
for (uint8_t i = 0; i < 256; ++i) {
    // infinite loop
}
for (unsigned i = 42; i >= 0; --i) {
    // infinite loop
}
```
Basic Functions (1)

Functions in C++

- Associate a sequence of statements (the function body) with a name
- Functions may have zero or more function parameters
- Functions can be invoked using a function-call expression which initializes the parameters from the provided arguments

Informal function definition syntax

```
return-type name ( parameter-list ) {
   statement
}
```

Informal function call syntax

```
name ( argument-list );
```
Basic Functions (2)

Function may have \texttt{void} return type

\begin{verbatim}
void procedure(unsigned parameter0, double parameter1) {
    // do something with parameter0 and parameter1
}
\end{verbatim}

Functions with non-\texttt{void} return type must contain a \texttt{return} statement

\begin{verbatim}
unsigned meaningOfLife() {
    // extremely complex computation
    return 42;
}
\end{verbatim}

The \texttt{return} statement may be omitted in the main-function of a program (in which case zero is implicitly returned)

\begin{verbatim}
int main() {
    // run the program
}
\end{verbatim}
Basic Functions (3)

Function parameters may be unnamed, in which case they cannot be used

```c
unsigned meaningOfLife(unsigned /*unused*/) {
    return 42;
}
```

An argument must still be supplied when invoking the function

```c
unsigned v = meaningOfLife();       // ERROR: expected argument
unsigned w = meaningOfLife(123);    // OK
```
Argument Passing

Argument to a function are passed **by value** in C++

```c
unsigned square(unsigned v) {
    v = v * v;
    return v;
}

int main() {
    unsigned v = 8;
    unsigned w = square(v);  // w == 64, v == 8
}
```

C++ differs from other programming languages (e.g. Java) in this respect

- Parameters can *explicitly* be passed by reference (more details later)
- Essential to keep argument-passing semantics in mind, especially when used-defined classes are involved (more details later)
Default Arguments

A function definition can include default values for some of its parameters

- Indicated by including an initializer for the parameter
- After a parameter with a default value, all subsequent parameters must have default values as well
- Parameters with default values may be omitted when invoking the function

```cpp
int foo(int a, int b = 2, int c = 3) {
    return a + b + c;
}

int main() {
    int x = foo(1);       // x == 6
    int y = foo(1, 1);    // y == 5
    int z = foo(1, 1, 1); // z == 3
}
```
Function Overloading (1)

Several functions may have the same name (*overloaded*)

- Overloaded functions must have distinguishable parameter lists
- Calls to overloaded functions are subject to *overload resolution*
- Overload resolution selects which overloaded function is called based on a set of complex rules

Informally, parameter lists are distinguishable

- If they have a different number of non-defaulted parameters
- If they have at least one parameter with different type
Function Overloading (2)

Indistinguishable parameter lists

```c++
void foo(unsigned i);
void foo(unsigned j); // parameter names do not matter
void foo(unsigned i, unsigned j = 1);
void foo(uint32_t i); // on x64
```

Valid example

```c++
void foo(unsigned i) { /* do something */ }
void foo(float f) { /* do something */ }

int main() {
    foo(1u); // calls foo(unsigned)
    foo(1.0f); // calls foo(float)
}
```
Facilities for printing to and reading from the console

- Use \textit{stream objects} defined in \texttt{<iostream>} header
- \texttt{std::cout} is used for printing to console
- \texttt{std::cin} is used for reading from console

The left-shift operator can be used to write to \texttt{std::cout}

```cpp
#include <iostream>

// ----------------------------------

int main() {
    unsigned i = 42;
    std::cout << "The value of i is " << i << std::endl;
}
```
The right-shift operator can be used to read from `std::cin`

```cpp
#include <iostream>

int main() {
    std::cout << "Please enter a value: " ;
    unsigned v;
    std::cin >> v;
    std::cout << "You entered " << v << std::endl;
}
```

The `<iostream>` header is part of the C++ standard library
- Many more interesting and useful features
- More details later
- In the meantime: Read the documentation!