# CSCE 121 — Intro to Programming Design and Concepts

Notes taken by Lukas Zamora

Fall 2018

# Contents

# 1 Computer Architecture and Compilation Process

## 1.1 Basic Architecture

The majority of modern computers are built using the Von Neumann architecture, where the CPU is where the computational power resides, the memory unit stores program code and data, and the two are connected by a "bus."

During each computation cycle, the machine retrieves the next instruction from the memory unit. Then subsequently executes the computation associated with the retrieved instruction. This process is repeated until the machine is told to halt.

## 1.2 Memory

The smallest unit of memory is the *bit*, which can be one of two states.

Computers use transistor circuits known as 'flip-flops' to store bits. It can either be on (1) or off (0).

The byte (8 bits) is the smallest accessable unit of memory in a computer.

Programming languages provide abstractions of these memory cells through variables and types.

- A *variable* is a named memory cell: we bind an identifier (name) to a memory cell by associating that identifier with the base-address of the respective memory cell.
- When we name a memory cell, we must always specify its type. The type determines the number of units of memory composing it and how its bit-pattern is to be interpreted.

## 1.3 Compilation

The compiler translates high-level programming language statements into an appropriate sequence of instructions in machine language. Several low-level instructions are typically required to express a single high-level statement.

The C++ compiler process proceeds by:

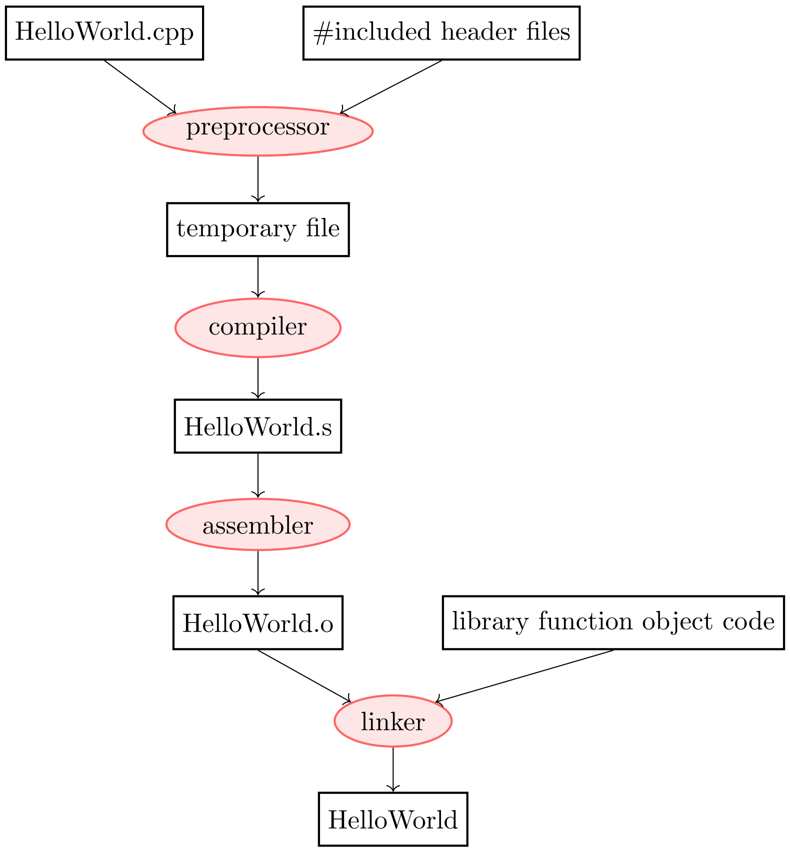
(i) Preprocessing the source file

(ii) Translating the source code to assembly code

(iii) Translating the assembly code to machine code (object code)

(iv) Linking necessary object code together into an executable file

**C++ Compilation Process**



# 2 Software Development Process

There are 4 key steps to the software development process: (1) Analysis, (2) Design, (3) Implementation, and (4) Repeat.

## 2.1 Analysis

First, we need to figure out what should be done (requirements/specifications)

- What are the possible problems that need to be solved?
- What is your current understanding of those problems?
- What is the process that you must go through to solve these problems?
- Are there any edge cases that must be considered/are there any constraints that must be acknowledged?

## 2.2 Design

Then we need to create an overall structure for the system

- How does the program flow?
- Which parts should the system have?
- How should those parts communicate?
- Can any libraries help you solve the problem?

We can capture design details using pseudocode and flowcharts.

## 2.3 Implementation

This step consists of three stages:

- Writing code
- Debugging the code that we've written
- Testing the code to ensure that it actually does what it is supposed to do

## 2.4 Repeat

We frequently build a small, limited version of our programs first. This helps us bring out programs in our understanding, ideas, and tools. It also helps us see if details of the problem statement need changing to make the problem manageable. It is rare to find that we had anticipated everything when we analyzed the problem and made the initial design. So we frequently have multiple iterations through the analysis, design, and implementation steps.

# 3 Data Representation

## 3.1 Positional Number System

Any positive integer $b > 1$ can be chosen as a base for a positional number system. For example, $b = 10$ denotes the decimal number system, $b = 2$ is the binary number system, etc.

Any integer $N$ is represented by a sequence of base-$b$ digits such that $b^k$ is the place value of $a_k$ and

$$N = a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b^1 + a_0 b^0$$

## 3.2 Binary Number System

The binary number system is a base 2 positional number system. The two digits used are 0 and 1. These numbers are called *bits*. Binary numbers are a sequence of bits which can have an embedded binary point. Binary integers are numbers without a fractional part.

A binary number $N_2$ can be converted to base-10 by writing $N_2$ in expanded notation. We can also use the following algorithm:

**Integral part**

1.) Double the leftmost digit

2.) Add the result to next digit to the right

3.) Double that sum

4.) Add the result to the next digit

5.) Repeat until the last digit is added; the final sum being the decimal equivalent

**Fractional part**

1.) Multiply the rightmost fractional digit by 1/2

2.) Add the next digit to the left of that product

3.) Multiply that sum by 1/2

4.) Add the next digit to that product

5.) Repeat until the leftmost digit is added

6.) Multiply that sum by 1/2

Given a decimal number $N$ with integer part $N_I$ and fractional part $N_F$, $N$ can be converted to base-2 using the following algorithm:

**Integral part**

1.) Subtract the longest possible power of base-2 from $N_I$

2.) Subtract the largest possible power of base-2 from that result

3.) Repeat until a difference of zero is obtained

4.) Place a bit value of 1 in the place values of those powers subtracted and a bit value of 0 everywhere else

**Fractional part**

1.) Multiply $N_F$ and the fractional portion of each succeeding product by 2 until a zero (or repeating) fractional part is observed

2.) The resultant sequence of integral parts in-order gives the corresponding representation of $N_F$ in base-2

**Binary Addition Properties**

- $0 + 0 = 0$
- $1 + 0 = 0$
- $0 + 1 = 0$
- $1 + 1 = 0$ with carry 1

# 4 Data Representation in the Computer

## 4.1 Sign Magnitude Notation

Reserve the most significant bit as a sign bit, with 1 being negative and 0 being positive. For example, $-3$ would be $1011 : -1 \cdot 1 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -3$ and 3 would be $0011 : -1 \cdot 0 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 3$.

This is problematic with arithmetic since $-3+3 = 1011+0011 = 1110 = -6 \neq 0$. Another problem is that we have 2 encodings for 0: 0000 and 1000 which gives us 0 and $-0$ (doesn't make sense).

To get around this, we use *two's compliment*, where we flip the bits (1's compliment) and add 1.

To find the largest positive $n$-bit number, we compute $2^{n-1} - 1$. To find the smallest positive $n$-bit number, we compute $-2^{n-1} - 1$.

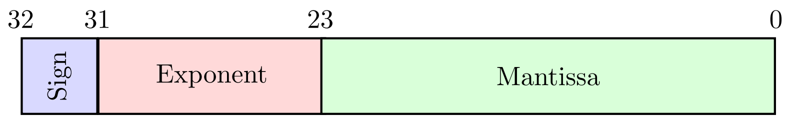
## 4.2 Floating-Point Numbers

Every decimal number $N_D$ can be written as a power of ten in exponential form:

$$N_D = 111 = 0.111 \cdot 10^3 = 1.11 \cdot 10^2 = 11.1 \cdot 10^1 = 111000 \cdot 10^{-2} \ldots$$

This form is not unique. However, we can write any non-zero $N_D$ uniquely as a number $M$ multiplied by a power of ten $e$ by ensuring that the decimal point appears directly in front of the first non-zero digit in $N_D$. This is known as *normalized exponential form*, where $M$ is called the *mantissa* of $N_D$.

Binary numbers can be written in normalized exponential form, using powers of 2 instead of 10.



– First bit denotes the sign $s$ of the number

– Second field is the exponent $e$ of the number

– Third field is the mantissa $m$ of the number, it contains the fractional part of the number in normalized binary form

Thus the floating-point value is encoded and calculated as

$$(-1)^s 2^{e-bias}(1+m)$$

Floating point numbers are fundamentally imprecise because they represent fractional values with a finite number of bits.

# 5 Objects, Values, and Types

The meaning of bits in memory is completely dependent on the *type* used to access it.

Some terminology:

- Type: Defines a set of possible values and a set of operations for an object.
- Object: Memory that holds a value of a given type.
- Value: Set of bits in memory interpreted according to type.
- Variable: Named object.
- Declaration: Statement that gives a name to an object.
- Definition: Declaration that sets aside memory for an object.

**Primitive Built-In Types**

- Boolean (`bool`): logical values
- Character (`char`): characters
- Integer (`int`): integer values
- Floating-Point (`double`): floating-point values

## 5.1 Boolean

The possible values of a boolean are true or false.

In both arithmetic and logical expressions:

- Bools are converted to integers
- Arithmetic and/or logical operations are performed on the converted values
- If the result is converted back to bool, a nonzero value is converted to true whereas a zero value to false
- True has the value 1, false has the value 0.

## 5.2 Character

Can hold a character of the implementation's character set. Each character constant has an integer value; however, whether char is signed or unsigned is implementation-defined. Chars are also integral types, so arithmetic and logical operations apply.

**Character Literal**: Notation for representing a fixed value. Also known as character constants. Enclosed by single quotes.

`char` ch = 'A';

## 5.3 Integer

Three integer types that vary size:

- `short int`
- plain `int`
- `long int`

Plain integers are always signed.

**Integer Literals**: Decimal, octal, hexidecimal, character literals.

## 5.4 Floating-Point

Three floating-point types that vary in size:

- `float` (single-precision)
- `double` (double-precision)
- `long double` (extended-precision)

The default floating-point type is `double`.

## 5.5 Variables

A program variable is an abstraction of a computer memory cell or collection of program cells. A variable can be characterized as a sextuple of attributes:

- Name
- Address
- Value
- Type
- Lifetime
- Scope

#### Name

Composed of a sequence of letters/characters. The first character in the name must be a letter. Keywords such as `return` or `string` cannot be used as variable names.

#### Address

The machine memory address with which the variable is associated. It is possible to have multiple names with the same address (aliases).

**Value**

The contents of the memory cell(s) associated with the variable. It appears on the right side of the assignment statement.

**Type**

Determines the range of values the variable can store and the set of operations that are defined for the values of that type.

**Lifetime**

The time during which the variable is bound to a specific memory location (allocation). Begins when the variable is bound to a specific cell. Ends when the variable is unbound from that cell.

**Scope**

Part of the program in which a name has a particular meaning. Names are visible from the point where they are declared to the end of the scope in which their declaration disappears.

## 5.6 Declarations

Each named object has a specific type associated with it, which determines the values put in it. Before a name can be used, we inform the compiler of its type through *declarations*. Most declarations are definitions, especially for arithmetic types. There must always be one definition for each name entity.

# 6 Expressions and Statements

**Statement**: A complete and meaningful command that can be given to a computer. In C++, a semicolon denotes the end of a statement.

**Unary Operator**: Acts on 1 operand ($++, --$)
**Binary Operator**: Acts on 2 operands ($+, -, *, \div$)

## 6.1 If Statement

Conditionally executes another statement based on whether a specified condition is true.

```
if (condition)
        statement
```

```
if (condition)
        statement
else
        statement
```

**Iterative Statements**: Provide for repeated execution until a condition is true.

## 6.2 While Statement

Repeatedly executes a statement as long as a condition is true.

```
while (condition)
        statement
```

For example, these are used when reading an input from the user.

## 6.3 For Statement

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

## 6.4 Do-While

The loop body is executed at least once, regardless of the value of the condition.

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

## 6.5 Debugging

Always write readable code (comment!)

– Use meaningful names

– Indent

– Use consistent layout

Break code into small functions

# 7 Compound Types, Compound Data

**Compound Type**: Defined in terms of another type.

## 7.1 References

Creates an alias for an object, allowing indirect access to that object.

```
int i = 11;
int &r = i;
```

So `i` and `r` refer to the same object (11).

A reference is another name for an already existing object.

## 7.2 Pointers

Compound data type that "points" to another type.

## 7.3 Array

Sequence of objects allocated in contiguous memory (all same type).

```
int arr[7];
```

This creates an array named `arr` that has 8 `int` objects.

## 7.4 Vector

Like arrays, but can hold an arbitrary number of objects. You can add objects to vectors with `push_back()`.

# 8 Type Conversions

## 8.1 Narrowing Conversions

Converts a value to a type that cannot store even approximations of all the values of the original type. For example, converting a `double` to a `float`.

## 8.2 Widening Conversions

Converts a value to a type that can include at least approximations of all of the values of the original type. For example, `int` to `double`.

Narrowing conversions aren't always safe whereas widening conversions are almost always safe.

**Coercion**: Implicit type conversion that is initiated by the compiler or runtime system.

**Explicit Type Conversions**: Casts. Use the command

```
static_cast<type> (value to cast)
```

Helpful for say floating-point division between two integers.

| Safe Type Conversions | Unsafe Type Conversions |
|---|---|
| `bool` to `char` | `double` to `int` |
| `bool` to `int` | `double` to `char` |
| `bool` to `double` | `double` to `bool` |
| `char` to `int` | `int` to `char` |
| `char` to `double` | `int` to `bool` |
| `int` to `double` | `char` to `bool` |

# 9 Errors

## 9.1 Compile Time Errors

- Syntax errors
- Type errors

## 9.2 Link Time Errors

Error found by linker when trying to combine object files into an executable, e.g an undefined function.

## 9.3 Run-Time Errors

Errors found by checks made during a running program

- The computer (hardware)
- A library
- User code (divide by zero)

## 9.4 Logic Errors

Found by the programmer looking for the causes of erroneous results.

# 10 I/O Streams

**Stream**: A programming language construct that provides you with a character based interface to I/O devices.

**Stream Buffer**: Houses a fixed amount of extracted stream data.

## 10.1 OStream

Turns values of various types into character sequences, sends them somewhere. The command `std::cout` is an ostream typed object that provides character sequences to standard output.

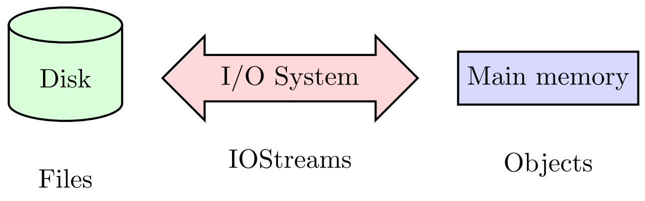
## 10.2 IStream

Turns character sequences into values of various types, gets those characters from somewhere. The command `std::cin` is an istream typed object that consumes character sequences from standard input.

## 10.3 The I/O Classes

- IOStream
  - `istream` reads from a stream
  - `ostream` writes to a stream
- FStream
  - `ifstream` reads from a file
  - `ofstream` writes a file
  - `fstream` reads and writes a file
- Sstream
  - `istringstream` reads from a string
  - `ostringstream` writes a string
  - `stringstream` reads and writes a string

**General Model**



To read a file, we must:

- Know its name
- Open it for reading
- Ensure that it opened successfully
- Read it
- Close it

To write a file, we must:

- Know its name
- Open it for writing
- Ensure that it opened successfully
- Write to it
- Close it

To construct an ifstream and open a file, we write

```
ifstream in(file);
```

To construct an ofstream, we write

```
ofstream out(file);
```

## 10.4 Stream State Flags

The I/O stream types each provide a collection of bits that are used to convey information about the state of a stream.

- `goodbit`: Set when the stream is not in an error state
- `badbit`: Set when an unrecoverable failure has occurred
- `failbit`: Set when a recoverable error has occurred
- `eofbit`: Set when the stream has hit the end of the file

# 11 Function Basics

## 11.1 Function Declarations

We can declare a function by writing a declaration of the form `f(args)`, where `f` is the name being introduced and `args` is a list of zero or more parameters.

```
double mult(double, double);
```

The base type specifies the return type. Put `void` if you don't want to return anything.

## 11.2 Function Definition

A declaration that fully specifies the entity being declared

```
double mult(double x, double y) {
    return x*y;
}
```

A declaration introduces a name to the compiler and how that name can be used. A *header file* is a file containing declarations, these files end with the extension `.h`. Header guards make up the `.h` file:

```
#ifndef DESCRIPTIVE_NAME_H
#define DESCRIPTIVE_NAME_H
// function declarations go here
#endif
```

These correspond to the file `descriptive_name.h`. We would then write a `descriptive_name.cpp` that contains our function definitions. Then we can use these new functions in `main.cpp` by writing `#include "descriptive_name.h"` in the header.

If done wrong, this would result in a linker error.

# 12 Function Argument Passing

**Pass-by-Value**: A value is copied into an object for initialization.
**Pass-by-Reference**: The parameter binds to the object being passed: the parameter becomes an alias for the object to which it is bound.

## 12.1 Pass-by-Value

Working with a copy of the argument

## 12.2 Pass-by-Reference

Changes that the function "makes" on a reference parameter will always be reflected in the object bound to that reference. The reference parameter is simply another name for the object for which it is initialized.

Pass large objects by reference to avoid copying large arguments.

Used to return multiple objects in functions.

## 12.3 Pass-by-Constant-Reference

Used when passing large objects and want the benefit of pass-by-reference, but d not need to modify arguments.

## 12.4 Guidance for Passing Arguments

- Use pass-by-value for small objects (primitive data types)
- Use pass-by-const-reference for large objects (vectors)

Never return a reference or pointer to a local variable, problems will occur.

## 13 Functions and the Stack

The stack manages function calls that are made during executions. Each time a function is called, an *activation record* for that function is pushed (added) to the stack.

Activation records store:

- Return location
- Arguments passed to the function
- Local variables defined in the function

When the function returns, its activation record is popped (removed) from the stack.

Code defining each function is stored in the code/static region of the program's address space.

# 14 Recursive Functions

**Recursive Function**: A function that is defined in terms of itself.

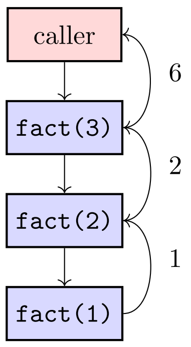
For example, the factorial function.

$$n! = \begin{cases} 1 & \text{if } n < 2 \\ n \cdot (n-1)! & \text{if } n > 2 \end{cases}$$

The base case is the non recursive case. For the factorial function, the base case is when $n < 2$.

## 14.1 Recursive Function and the Stack

To better illustrate what's going on, we use these function trees. For example, say we have a function called `fact()` that computes the factorial. The function tree for `fact(3)` looks like



The boxes represent stack frames, straight arrows are function calls, and curved arrows are return values.

In general, anything solved recursively has an iterative solution.

# 15 Functions and Exceptions

## 15.1 Exceptions

C++'s means of separating error reporting from error handling in a general way.

Introduce a try block around the code where an exception might occur. Then catch the exception of a specific type.

```
try {
    //stuff
} catch (const runtime_error& e) {
    cerr << "Exception: " << e.what() << endl;
}
```

## 16 Function Overloading

**Overloaded Functions**: Functions that have the same name but different parameters list and appear in the same scope.

Eliminates the need to defined different names for different functions that perform the same general action but on different parameter types.

**Overload Resolution**: The process by which the compiler determines which specific function is called from a set of overloaded functions.

- The compiler determines this by comparing the arguments against the parameters of each function in the set of overloaded functions.

Use function overloading when a name is semantically significant amongst different data types.

When we overload functions, we are creating multiple functions that have the:

- Same name
- Different parameter configurations
  - Number of parameters
  - Types of parameters
  - Order for parameter types

# 17 Dynamic Memory

### Stack

The size of variables must be known at compile-time.

A new block of memory called the stack frame (activation record) is added to the stack to hold automatic variables each time you call a method.

### Heap/Free Store

Size of variables may be unknown at compile-time.

Allocation is performed at run-time.

Dynamically created objects are stored on the free store.

## 17.1 Allocation of Memory to Variables (stack)

When we write something like `int k = 11`, when the compiler sees this, it determines the amount of memory to hold the value of an `int`, and adds the identifier `k` to a symbol table with its memory address in which the object will become accessible during run-time.

When a function is called, the stack pointer is moved in one direction to allocate memory on the stack for the local variables associated with the called function.

When the function finishes execution, the stack pointer is moved back in the other direction, and memory is deallocated.

If the sizes of the local variables are fixed at compile-time and stored in the same order in the activation record each time the function is invoked, then the location of each variable will be a fixed offset from the stack pointer.

## 17.2 Dynamically Allocated Objects (heap/free store)

We dynamically allocate memory on the free store using the `new` operator.

- The `new` operator allocates memory for an object of a specified type.
- The `new` operator returns the address of the region of memory allocated for that object.

It's our responsibility to explicitly deallocate that memory when we are finished using it; we do this by using the address as an operand to the `delete` operator.

| **Clause** | **Result of Violation** |
|---|---|
| You will eventually return the memory you borrow | Memory Leak |
| You will immediately stop using that memory that you've returned | Dangling Pointer |
| You will not return memory that you did not borrow (and you will not twice return memory that you've borrowed once) | Corrupted Heap |

**Memory Leak**: Occurs when we dynamically allocate memory for an object, but fail to ever deallocate that space when we're done using it.

**Dangling Pointer**: When a pointer to a piece of memory that has deallocated is used. Until that memory is actually allocated again, you may continue to get the value that was stored in the object that used to reside there.

**Corrupted Heap**: Happens when something is deallocated that is not allocated.

# 18 Classes

**Class**: A user-defined type: composed of built-in types, other compositions, and functions.

A class provides the description for how objects of that type are to be represented.

- The representation of the user-defined types is composed of built-in types and other user-defined types that are known as *data members*
- *Function members* are written to provide the operations that we will be able to apply to the objects of our user-defined type.

## Why classes?

- *Data abstraction* allows us to focus on what operations that will be performed on the data members opposed to how we will perform those operations; hidden is underlying structure, increased is the modularity and transparency.
- *Encapsulates* data together with the operations that can be performed on that data.
- *Data hiding* can be accomplished using member access specifiers: restrict interaction with class members across a well-defined public interface; present only the fundamental facilities that the user needs for use, and hide all implementation details within the class itself.

`class` members are private by default, whereas `struct` members are public by default

## Data members: Public or Private?

- If the value assigned to an object will work regardless, declare public.
- If the value assigned to an object needs to be checked, or must conform to the same requirements, declare private.

Making attributes private can help maintain the integrity of our data members by inhibiting the direct manipulation of their values; interaction with private data members are limited to the extent provided by the public interface.

If you want private attributes to be "public", use accessors/mutators.

## 18.1 Accessors(getters) and Mutators(setters)

**Accessor(getter)**: A function that returns the value stored in a private data member.

**Mutator(setter)**: A function that stores a value in a private data member or mutates its state.

Member function declarations for your class should be stored in the `ClassName.h` file. Member function definitions for your class should be stored in the `ClassName.cpp` file with scope resolution (e.g. `ClassName::getx() {return x;}`).

## 18.2 Constructors

In order to initialize our data members upon object instantiation, we write a special member function known as the *constructor*.

- The *constructor* is implicitly invoked whenever an object of the user-defined type is created.
  - Its job is to construct an object and do initialization if necessary
- We can also acquire resources in the constructor; maybe allocate dynamic memory for some object, or open a file

We can overload constructors of a user-defined type; we typically use *parameterized constructors* to initialize data members.

### Destructors

Much like how a constructor is called whenever an object of a user-defined type is created, a *destructor* is called when an object's lifetime is up.

- One primary use for a destructor is to release resources that the object had required during construction.

There is only one destructor per class(can't be overloaded).

## 19 Overloaded Operators for User-Defined Types

An overloaded operator has the same number of parameters as the operator has operands (single parameter for unary, two parameters for binary).

– For binary, the left-hand argument is used to initialize the first parameter and the right-hand argument the second.

We can only overload existing operators: can't invent new operator symbols.

If the overloaded operator is defined as a member function, the first operand is bound to the implicit `this` pointer.

– Due to this, a member operator function will have one less (explicit) parameter than the operator has operands.

If the overloaded operator is defined as a non-member function, at least one of its parameters must be a user-defined type

– This implies that we cannot change the meaning of an operator when applied to operands of built-in type.