# Syntax and Data Types

C++ for Developers

# Content

- Syntax
- Basic Data Types
- Derived Data Types
- User Defined Data Types

# Syntax

C++ for Developers

# Syntax

```cpp
#include <iostream>

int main(void) {

    std::string name = "Carl Broman";
    int age = 32;

    std::cout << name << " is " << age << " years old" << std::endl;

    return 0;
}
```

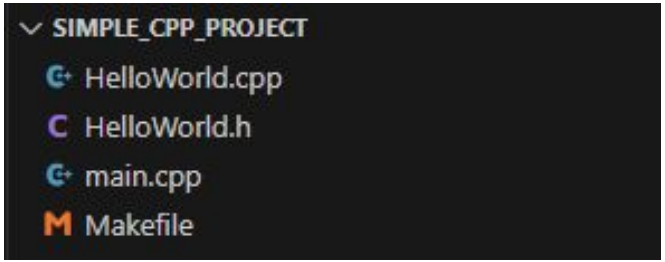
# Syntax

**#include**
This is where we include important files that are used in our programs. <iostream> defines the basic I/O-operations.

**main**
The entry point of the program. This must be defined for the program to run.

**Returning a number**
This is to signal the success of the program. 0 usually means all went well (nothing went wrong).

# Folder structure




- **main.cpp** - contains our main function.
- **.h files** - declarations of variables and functions
- **.cpp files** - definitions and declarations
- **Makefile** - build instructions to the compiler

lafftale for developers

# Folder structure

**main.cpp**
main function

```cpp
#include <iostream>
#include "HelloWorld.h"

int main(void) {

    sayHelloWorld();

    return 0;
}
```

**HelloWorld.h**
Declaration of sayHelloWorld()

```cpp
#ifndef HELLO_WORLD_H
#define HELLO_WORLD_H

void sayHelloWorld();

#endif
```

**HelloWorld.h**
Definition of sayHelloWorld()

```cpp
#include "HelloWorld.h"
#include <iostream>

void sayHelloWorld() {
    std::cout << "Hello, world" << std::endl;
}
```

**Makefile**
Instructions to our compiler.

```
PROG = main.exe
SRC = main.cpp HelloWorld.cpp
CFLAGS = -g -std=c++17 -Wall -Werror
LIBS =
CC=g++

all: $(PROG)

$(PROG): $(SRC)
    $(CC) -o $@ $(CFLAGS) $(SRC) $(LIBS)

clean:
    rm -f $(PROG)
```

# Naming Conventions

**Variables**
Names should clearly describe what the variable represents or is used for. Prefer descriptive nouns (e.g., userCount, filePath, averageScore). Short names can be used for short-lived variables.

**Functions**
Names should express what the function does or returns. Use verbs for clarity (e.g., calculateAverage, loadFile, isValid).

# Naming Conventions

## User-Defined Data Types (classes, structs, enums, typedefs)

Names should describe what the type represents. Use nouns that capture the concept (e.g., UserProfile, Transaction, ColorMode). Names should be clear enough that their purpose is obvious without additional comments.

# camelCase / PascalCase

**Variables:** personCounter;

**Functions:** increasePersonCounter();

**User Defined:** Person;

**Contants / Macros:** PERSON_MAX_LIMIT

# snake_case

Typically is everything named in snake_case except:

**Constants / Macros:** PERSON_MAX_LIMIT

# Follow the codebase

The most important is to follow the agreed naming convention already existing in the codebase and staying consistent.

# Data Types

C++ for Developers

# Variables

A variable is a name storage location holding a value.

**Declaring**
Creates a space in memory for the variable of this size.

**Initialize**
Assigning a value in the memory space we declared.

```
int x;          // Declaration
x = 20;         // Initialization

int y = 30;     // Both
```

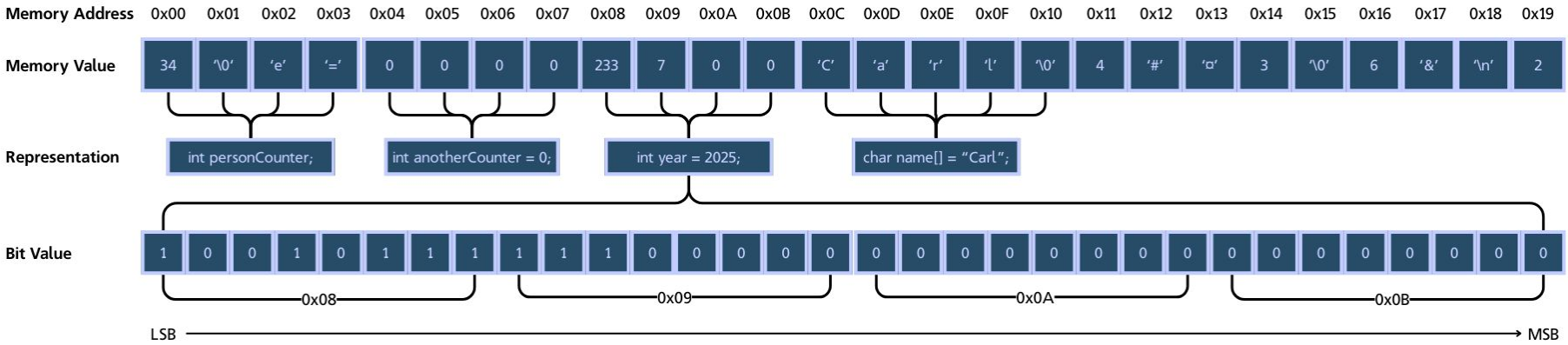
lafftale for developers

# What happens when we declare and/or initializes a variable?

Memory can be imagined as a collection of labeled boxes that can hold data.

- Declaring a variable is like choosing a box of the right size (based on the data type) and giving it a label (the variable name).
- Initializing the variable means putting a specific value into that box.

# What happens when we declare and/or initializes a variable?




lafftale for developers

# Arithmetic operators

For basic arithmetic operations you can use:

+ - / *

Sometimes you want to perform operations on the variable itself:

personCounter = personCounter + 1;

Then you can use: personCounter += 1;

*This works for all operators*

# Incrementing / Decrementing

Often times we use variables that increases or decreases with 1. These can also be done with the increment and decrement symbol ‘++’ or ‘--’. Depending on the position:

**Inc/Dec after reading the value:** counter++; or counter--;

**Inc/Dec before reading the value:** ++counter; or --counter;

# Basic Input / Output Operations

C++ for Developers

# Output std::cout

Writes out to console. Will implicitly cast everything to string.

```
int age = 29;
char name[] = "Carl Broman";

std::cout << name << " is " << age << " years old." << std::endl;
```

# Input std::cin

Takes an input from the console and casts it to the data type of the variable.

```
int age;
char name[20];

std::cout << "Whats your name?" << std::endl;
std::cin >> name;

std::cout << "How old are you?" << std::endl;
std::cin >> age;

std::cout << name << " is " << age << " years old." << std::endl;
```

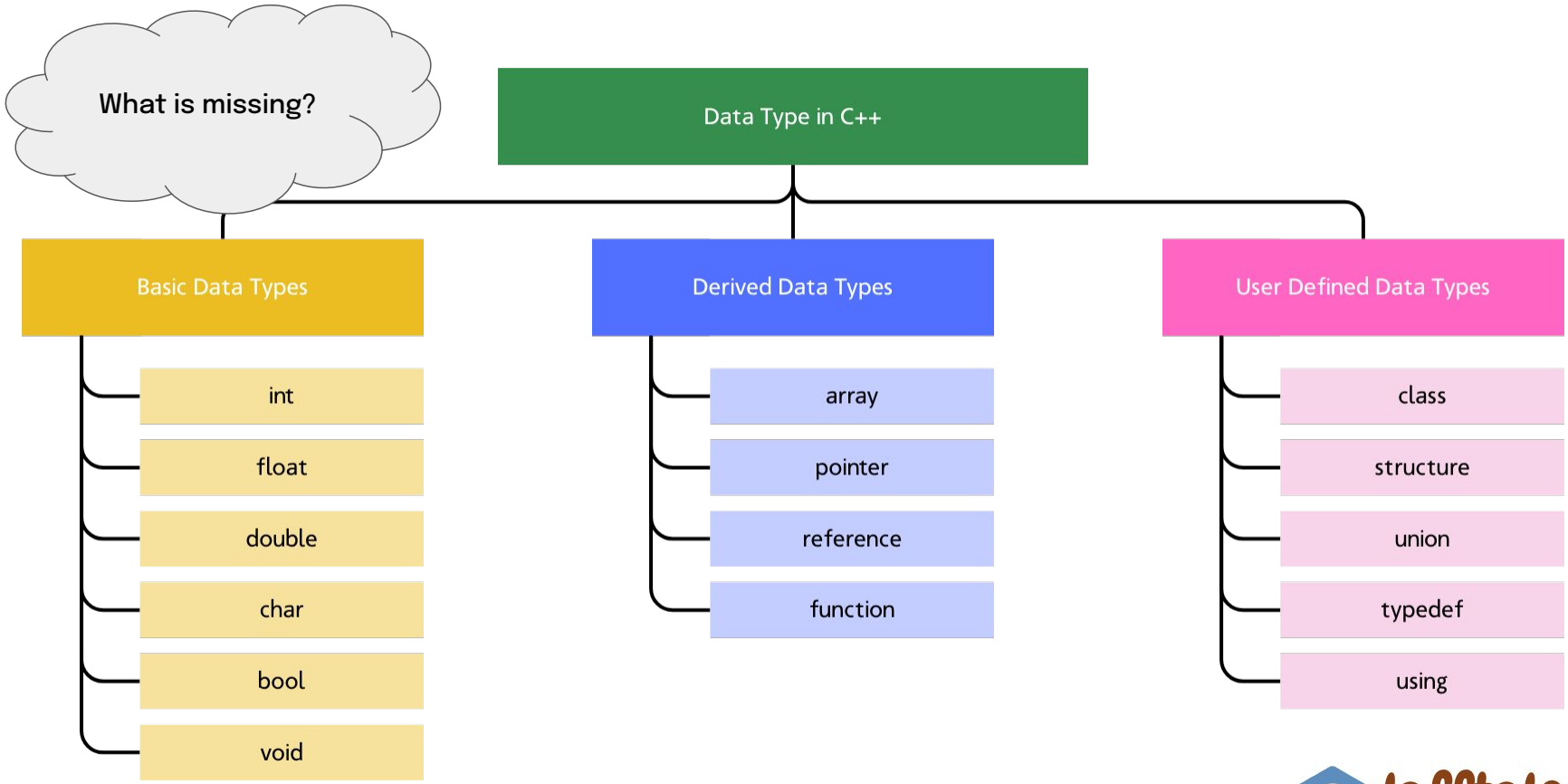
# Input std::getline(std::cin, std::string)

Defined in <string>. Unlike std::cin, which only takes one word, std::getline takes in the whole line.

```
std::string firstAndLastName;

std::cout << "What is your full name? ";

std::getline(std::cin, firstAndLastName);

std::cout << "Your full name is: " << firstAndLastName << std::endl;
```

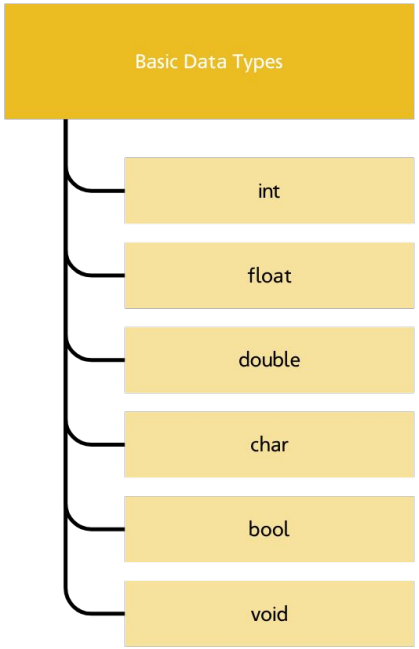
# Basic Data Types

C++ for Developers




lafftale for developers

# Basic Data Types




Built in data types in the C++ language.

https://github.com/lafftale1999/cpp_for_developers/blob/main/week_1/3_data_types.cpp

lafftale for developers

# Basic Data Types Size

| Type | Size |
|---|---|
| char | 1 byte |
| bool | 1 byte |
| int | 2 bytes / 4 bytes |
| short int | 2 bytes |
| long int | 4 bytes / 8 bytes |
| long long int | 8 bytes |
| float | 4 bytes |
| double | 8 bytes |
| long double | 12 bytes |

Size that is reserved in memory for the data type.

Depending on system, some types can vary in size.

# Basic Data Types

```
char char_ex = 'H';           // 1 Byte   -127 - 127 or 0 - 255
unsigned char us_char = 'H';  // 1 Byte   0 - 255
signed s_char = 'H';  ;       // 1 Byte   -127 - 127
```

## Signed values

The value has a range from negative max to positive max.

## Unsigned values

The value has a range from 0 to positive max.

A variable is by standard signed. Except for char that is dynamically decided.

# Casting

To convert data types into others, we “cast” them either explicitly or implicitly.

Explicit casting means, we do it clearly - ordering a cast.

Implicit casting means it happens automatically behind the scenes.

# Explicit casting

Large to small:

```
double aNumber = 3.14159265359;
int aNumberCasted = (int) aNumber;
```

Small to large:

```
int anotherNumber = 323932012;
long long int anotherNumberCasted = (long long int) anotherNumber;
```

What can happen here?

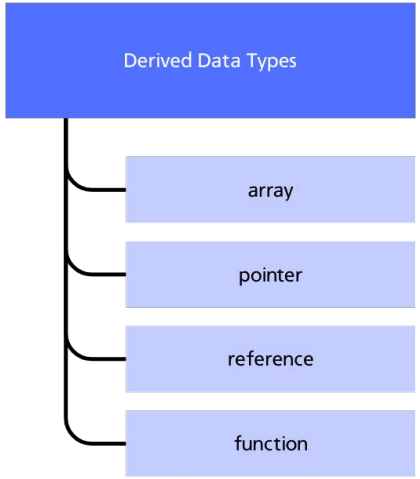
# Implicit casting

This happens in the background indirectly, like when we use std::cout - it automatically casts numbers into characters instead.

# Derived Data Types

C++ for Developers

# Derived Data Types




Derived data types are built from the basic data types to provide more complex ways of storing, organizing, or working with data.

Link to examples:
https://github.com/lafftale1999/cpp_for_developers/blob/main/week_1/4_derived_data_types.cpp

lafftale
for developers

# Derived Data Types - Array

An array is a container of defined values, stored contiguously in memory.

To declare an array you need to specify its data type and typically its size. If you initialize it at the same time, you don't have to assign its size.

```
char anotherName[] = "Carl";
```

```
char someName[5];
someName[0] = 'C';
someName[1] = 'a';
someName[2] = 'r';
someName[3] = 'l';
someName[4] = '\0';
```

lafftale for developers

# Derived Data Types - Array

## Valid syntax

### Declaring

```
char someName[5];
```

| | |
|---|---|
| '#' | 0x04 |
| '\n' | 0x03 |
| E | 0x02 |
| 7 | 0x01 |
| 'f' | 0x00 |

### Declaring & Initializing

```
char anotherName[] = "Carl";
```

| | |
|---|---|
| '\0' | 0x04 |
| 'l' | 0x03 |
| 'r' | 0x02 |
| 'a' | 0x01 |
| 'C' | 0x00 |

### Initialization

```
someName[0] = 'C';
someName[1] = 'a';
someName[2] = 'r';
someName[3] = 'l';
someName[4] = '\0';
```

| | |
|---|---|
| '\0' | 0x04 |
| 'l' | 0x03 |
| 'r' | 0x02 |
| 'a' | 0x01 |
| 'C' | 0x00 |

### Declaring & Initializing

```
char *name = "Carl";
```

| | |
|---|---|
| '\0' | 0x04 |
| 'l' | 0x03 |
| 'r' | 0x02 |
| 'a' | 0x01 |
| 'C' | 0x00 |

## Not valid syntax

### Declaring without size

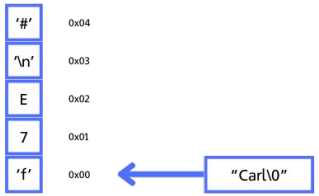
```
char noSizeName[];
```

No spaces reserved in memory.

### Assigning the whole array at once, after declaring it

```
someName = "Carl";
```

someName is pointing to the first place in memory. This syntax is trying to fit 5 characters into the memory space of 1.




lafftale for developers

# Derived Data Types - Pointers and References

Data Types to navigate in memory. We will go into these further ahead in the course.

```
int x = 23;

int *pointerToX = &x;        // pointer to X
int &referenceToX = x;       // reference to x

x = 32;

std::cout << *pointerToX << std::endl;
std::cout << referenceToX << std::endl;
```

lafftale for developers

# Derived Data Types - Functions

A function is built by defined data types:

- Return value
- Parameter

```
// Return value | Function name | (parameters) { BODY OF CODE}
char getStudentGrade(int studenId) {
    // BODY OF CODE
}
```

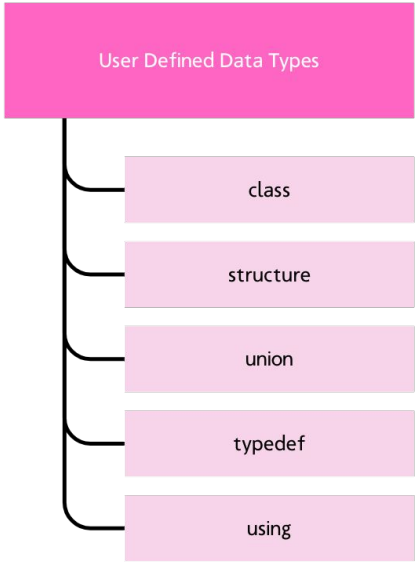
Functions consists of:

- Return type
- Name (symbol)
- Parameters
- Body (code)

# User Defined Data Types

C++ for Developers

# User Defined Data Types




Data types defined by the user. Providing structure to create complex representation of data.

Link to examples: https://github.com/lafftale1999/cpp_for_developers/blob/main/week_1/5_user_defined_data_types.cpp

lafftale for developers

# Class, Struct and Unions

A collection of variables to describe complex objects. We will go deeper into classes further ahead in the course.

# Structures

A structure is a collection of user defined values to create a complex data type.

You can access the attributes of the struct by using ‘.’ after the variable name.

```
std::cout << "Name: " << student.name << std::endl;
std::cout << "Age: " << student.age << std::endl;
std::cout << "Grade: " << student.grade << std::endl;
```

```
struct Student {
    char firstName[20];
    short int age;
    char grade;
};

Student student = {
    {"Ada Lovelace"},
    25,
    'A'
};
```

# Enumerations

Enumerations is an user defined data type consisting of named values that represent integer constants.

By standard, the integer values start from 0 and upwards. But you can also choose your own values.

```
enum AuthenticationType {
    RFID,   // 0
    PIN,    // 1
    RETINA  // 2
};

AuthenticationType type = RFID;
```

```
enum AuthenticationType {
    RFID = 25,
    PIN = 32,
    RETINA = 29
};

AuthenticationType type = RFID;
```

lafftale for developers

# Unions

In C++, a union is a user-defined type where all members share the same memory space.

Therefore, unlike a structure, unions can only hold one value at the time. It is up to the caller to decide how to parse the union.

```
struct Authentication {
    AuthenticationType type;

    union {
        char biometricHash[65];
        char pinCode[5];
        char keyID[10];
    };
};

Authentication auth;
auth.type = RFID;
snprintf(auth.keyID, sizeof(auth.pinCode), "3E 54 9F");

if(auth.type == RFID) {
    std::cout << auth.pinCode << std::endl;
}
```

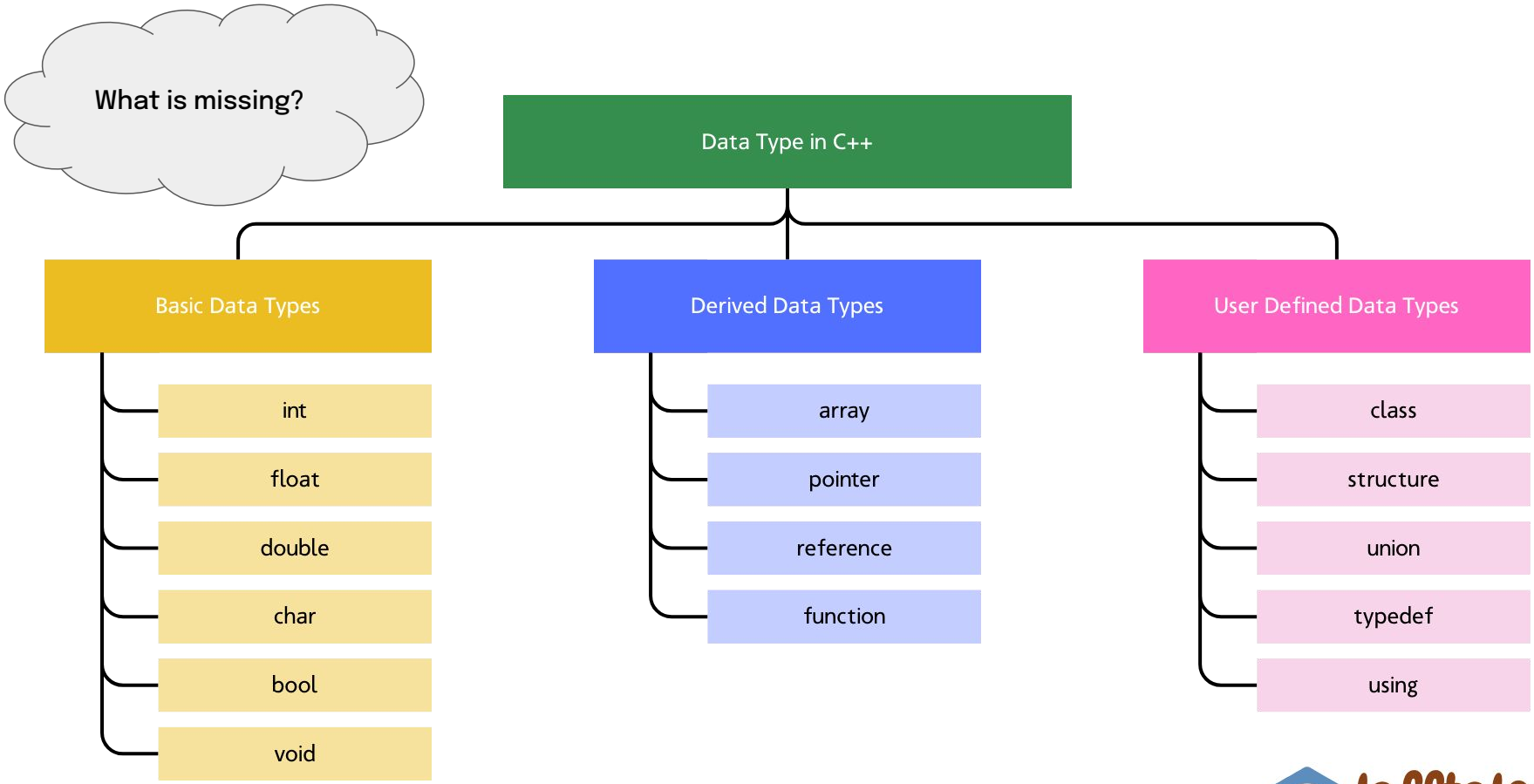
# Aliases - typedef and using

A typedefinition (typedef or using) in C++ creates an alias for an existing data type.

It does not define a new type, but provides an alternative name that can make code more readable, easier to maintain, or less error-prone.

```
typedef char Name[20];

Name name = "Carl";
```

```
using uint = unsigned int;

uint unsignedInteger = 54320;
```



# Extra

C++ for Developers

# String

String is not a built in data type in C++, it is an User Defined Class. It is often used in high level applications, but avoided in low level because of its memory allocation.

For low level applications, using an array is preferred to control the memory.

included in <iostream>

# Useful methods for std::string

Examples of useful methods for std::string:

https://github.com/lafftale1999/cpp_for_developers/blob/main/week_1/6_string_class.cpp

lafftale for developers

# Vector

A vector in C++ is a dynamically allocated array that can hold elements of any type. Unlike regular arrays, it can automatically grow or shrink as you add or remove elements.

Vectors store their data contiguously in memory, which makes iteration very fast and efficient. However, inserting or removing elements (especially in the middle) can be costly, since memory may need to be shifted or reallocated.

In embedded systems, vectors are rarely used because they rely on heap allocation, which can cause fragmentation and unpredictable memory usage.

# Vector

Vector is defined in the <vector> library, and it must be include for you to use its functionality.

Examples of useful methods for std::vector<T>:

https://github.com/lafftale1999/cpp_for_developers/blob/main/week_1/7_vector.cpp

lafftale for developers

# The ‘const’ keyword

The const keyword makes a variable read-only: once it’s initialized, its value cannot be changed.

- A const variable must be initialized at the moment it is declared.
- Any attempt to assign a new value later will cause a compile error.

```
const int constInteger = 32;
```

# #define macros

These use text replacements during compilation and expands the macros to the code they represent.

# Exercises

Data types

- 1. Hello World
- 2. Year Calculator
- 3. Present yourself
- 4. Calculator: Addition
- 5. Calculator: Division


lafftale
for developers