



## Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

### GPU & FPGA module training

Week 2 : Introduction to C++

Lecture 4 - February 1<sup>st</sup> 2023

# What we learnt in the previous lecture

- History of C++
- Brushed up on :
  - Core syntax
  - Variables & Operators
  - Control instructions
  - Functions



# Today

- Scopes / namespaces
- Compound data types
- Object Orientation
- The C++ compilation chain



# Scopes and namespaces

# Scope in C++

**Scope** → portion of the source code where a given variable can be accessed / declared / used etc :

- Typically
  - simple block of code, within {}
  - function, class, namespace
  - translation unit for global declarations
- Resources are allocated when a e.g variable is declared
- Resources are then freed at the end of a scope

```
{  
    int a;  
    {  
        int b;  
    } // end of b scope  
} // end of a scope
```

# Scope of variables in C++

Mainly two types of variable scopes :

- **Local Variables**

- Are declared inside a block
- Cannot be accessed or used outside that block

- **Global Variables**

- Declared outside of all of the functions and blocks, at the top of the program.
- Can be accessed from any portion of the program.

```
#include<iostream>
using namespace std;
int global = 1; // Global variable
int main()
{
    int local = 2; // Local variable
    cout<<"Global var : "<<global;
    cout<<" and Local var : "<<local;
    return 0;
}
```

# Namespaces (1)

- declarative region that provides a scope to the identifier
- allow to segment code to avoid name clashes
- especially useful when your code base includes multiple libraries
- can be embedded to create hierarchies using the '::' separator

## Syntax

```
namespace namespace_name  
{  
    declarations  
}
```

## Usage

```
namespace_name::namespace_members
```

### std namespace

- The std is a short form of standard
- the std namespace contains the built-in classes and declared functions.
- e.g. list , vector , cout etc.

# Namespaces (2)

Namespaces can also be nested. Usage :

- **A::a** (outer namespace)
- **A::B::a** (inner namespace)

```
namespace A {  
    int a;  
    namespace B {  
        int a;  
    }  
}
```

```
namespace A {  
    int a;  
    void func() {  
        cout << "Namespace A" << endl;  
    }  
}  
  
namespace B {  
    int a;  
    void func() {  
        cout << "Namespace B" << endl;  
    }  
}
```

## Exercise : Lets try this out!

- Open [onlinegdb](#)
- Copy the above two namespaces
- Let's call them from main and checkout the results!





# Compound data types



# What are compound data types?

A compound type is a **type that is defined in terms of another type**.

There are many compound data types in C++ :

- Arrays
- Functions
- Pointers
- References
- Structs & class types
- Enumerations

# Arrays

- Series of elements of the same type placed in contiguous memory locations
- Elements can be individually referenced by adding an index to a unique identifier.
- Can be **static** or **dynamic**
  - Size of static arrays is determined when the data structure is defined or allocated.
  - Dynamic arrays allows individual elements to be added or removed (e.g. `std::list` or `std::vector`)
- **Syntax** of static arrays: type name [elements]
- Arrays can also be multidimensional

```
// Defined values
int values[] = {0, 1, 2};
// Fixed-length with
// undefined values
int values[3];
```

```
          i   j
          ↓   ↓
int mult[2][3]
```

	(0,0)	(0,1)	(0,2)
i	(1,0)	(1,1)	(1,2)
		j	

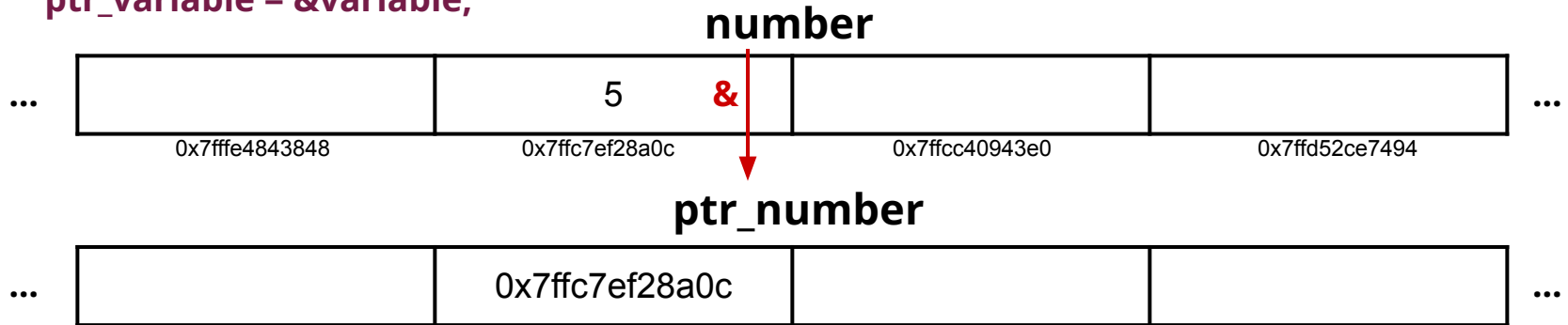
# Pointers (1)

**Pointer** → variable that stores the memory address of a variable as its value.

- **Syntax** `type* pointer_variable;`
  - \* → **dereference operator**
- Variable address can be obtained by preceding the name of a variable with the **Address-of** operator (&) e.g. **`ptr_variable = &variable;`**

```
int number = 5;
```

```
int* ptr_number = &number;
```



# Pointers (2)

**Pointer** → variable that stores the memory address of a variable as its value.

- **Syntax** `type* pointer_variable;`
  - \* → **dereference operator**
- Variable address can be obtained by preceding the name of a variable with the **Address-of** operator (&) e.g. **`ptr_variable = &variable;`**
- Pointer should be initialized to point to a valid address
- If a pointer doesn't point to anything, set it to **`nullptr`** (i.e. `int* ip = nullptr`)

```
int number = 5;
int* ptr_number = &number;
cout << number << "\n";
cout << &number << "\n";
cout << ptr_number << "\n";
```



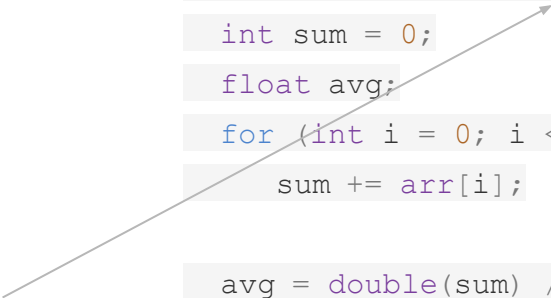
## **Exercise : Lets try this out!**

- Open [onlinegdb](#)
- Copy the snippet
- Lets see what is printed out

# Pointers (3)

A function which accepts a pointer, can also accept an array as an argument :

```
float getAverage(int *arr, int size) {  
    int sum = 0;  
    float avg;  
    for (int i = 0; i < size; ++i)  
        sum += arr[i];  
  
    avg = double(sum) / size;  
    return avg;  
}
```



```
int main () {  
    int balance[5] = {1, 2, 3, 4, 5};  
    float avg;  
    // pass pointer to the array as an argument.  
    avg = getAverage( balance, 5 ) ;  
    cout << "Average value is: " << avg << endl;  
    return 0;  
}
```

# References

- References allow for direct access to another object
- They can be used as shortcuts / better readability
- References should always refer to an object and should be initialized when created
- Once a reference is initialized to an object, it cannot be changed to refer to another object.
- They can be declared const to allow only read access
- They can be used as function arguments

```
int a;
```

```
int& ref_a = a;
```

# Pointers vs References

**References are preferred  
with respect to pointers**

## Pointers

- Pointers can be null
- We can change the variable that a pointer points to
- They indicate that the value of a variable can be modified
- If memory is not released, a memory leak can develop
- Prone to segfaults

## References

- References can never be null & needs to be initialized during declaration.
- After initialization, cannot change the reference to reference another variable.
- Can be declared const to allow only read access



# Structures

- A data structure (**struct**) is a group of data elements grouped together under one name.
- These data elements are referred to as members
- Members can have different types and different lengths.

Specifying Object\_names  
is **optional**



## Syntax :

```
struct type_name {  
    member_type1 member_name1;  
    member_type2 member_name2;  
    member_type3 member_name3;  
    ....  
} object_names;
```

## Declaration :

```
struct Person {  
    unsigned char age;  
    float weight;  
};
```

## Usage :

```
Person charis;  
charis.age = 29;  
charis.weight = 55.5;
```

# Enumerations

- An enumeration (**enum**) is a distinct type whose value is restricted to a range of values
- These may include several explicitly named constants → enumerators
- An enum variable takes only one value out of many possible values

```
#include <iostream>
using namespace std;

enum month { January, February, March, April,
            May, June, July, August, September,
            October, November, December };

int main()
{
    month thisMonth;
    thisMonth = January;
    cout << "Month " << thisMonth;
    return 0;
}
```

**What do you think will be printed? Lets see :**

- Open [onlinegdb](#)
- Copy the snippet & run!


# Enumerations

- An enumeration (**enum**) is a distinct type whose value is restricted to a range of values
- These may include several explicitly named constants → enumerators
- An enum variable takes only one value out of many possible values

```
#include <iostream>
using namespace std;

enum month { January, February, March, April,
            May, June, July, August, September,
            October, November, December };

int main()
{
    month thisMonth;
    thisMonth = January;
    cout << "Month " << thisMonth;
    return 0;
}
```



**You can change the default value of an enum element during declaration e.g.**

```
enum month { January=1, February=2, March=3, April=5,
            May=8, June=15, July=20, August=35, September=40,
            October=50, November=60, December=70
};
```




# Object Orientation




# Object Oriented vs Functional programming

- **Object oriented programming** (OOP) groups related functions and their variables into objects.
- Imperative programming paradigm
  - Update the running state of the program
- Based on the following principles :
  - Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism
- **Functional programming** is paradigm where programs are constructed by composing and making use of functions
- Declarative programming paradigm
  - Maps values to other values
- Efficient
  - Code is reusable
- Allows parallel programming
- Allows for modular code


# Object Oriented Programming principles

- OOP groups related functions and their variables into objects.
- Imperative programming paradigm
  - Update the running state of the program
- Based on the following principles :
  - **Encapsulation** 
  - Abstraction
  - Inheritance
  - Polymorphism
- Ability to group data along with properties and methods that operate on the data in a common unit

# Object Oriented Programming principles


- OOP groups related functions and their variables into objects.
- Imperative programming paradigm
  - Update the running state of the program
- Based on the following principles :
  - Encapsulation
  - **Abstraction** 
  - Inheritance
  - Polymorphism
- Ability to represent data at a very conceptual level without any details.

# Object Oriented Programming principles

- OOP groups related functions and their variables into objects.
- Imperative programming paradigm
  - Update the running state of the program
- Based on the following principles :
  - Encapsulation
  - Abstraction
  - **Inheritance** 
  - Polymorphism
- A class can be derived from a base class with all features of base class and some of its own.
- Increases code reusability



# Object Oriented Programming principles

- OOP groups related functions and their variables into objects.
  - Imperative programming paradigm
    - Update the running state of the program
  - Based on the following principles :
    - Encapsulation
    - Abstraction
    - Inheritance
    - **Polymorphism**
  - Ability to exist in various forms
  - Functions with the same name can be overloaded to perform different tasks
- 

# Classes

- Expanded concept of C structures:
  - Contain data members but also **contain functions as members** → **methods**
  - Access control (public/private/protected)
  - Inheritance
- **Object** : Class instance
- A class encapsulates a concept :
  - implementation
  - properties
  - possible interactions
  - construction and destruction

- **Syntax:**

```
class class_name {  
    access_specifier_1:  
        member1;  
    access_specifier_2:  
        member2;  
    ...  
} object_names;
```

**Members : data or  
function declarations**

**Access specifiers**

# Implementing methods

```
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
    width = x;
    height = y;
}

int main () {
    Rectangle rect;
    rect.set_values (3,4);
    cout << "area: " << rect.area();
    return 0;
}
```

Class **Rectangle** has 2 data members **width, height** and two methods **set\_values & area**

# Implementing methods

```
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
    width = x;
    height = y;
}

int main () {
    Rectangle rect;
    rect.set_values (3,4);
    cout << "area: " << rect.area();
    return 0;
}
```

Class **Rectangle** has 2 data members **width, height** and two methods **set\_values & area**

## Methods :

- Usually implemented outside of class declaration
- Use the class name as namespace
- When reference to the object is needed, use **this** keyword

# Implementing methods

```
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
    width = x;
    height = y;
}

int main () {
    Rectangle rect;
    rect.set_values (3,4);
    cout << "area: " << rect.area();
    return 0;
}
```

Class **Rectangle** has 2 data members **width, height** and two methods **set\_values & area**

## Methods :

- Usually implemented outside of class declaration
- Use the class name as namespace
- When reference to the object is needed, use **this** keyword
  - `this->width = x;`

In *main*, syntax to construct a class object and access its members & methods

# Overloading

- A class can have multiple functions with the same name but different parameters

## Exercise : Lets try this out!

- Open [onlinegdb](#)
- Copy the snippet
- Lets see what is printed out
- Can you think of/try out another function overload?

```
#include <iostream>
using namespace std;
```

```
class Rectangle {
    int width, height;
public:
    void set_values (int,int);
    void set_values (int);
    int area() {return width*height;}
};
```

```
void Rectangle::set_values (int x, int y) {
    width = x;
    height = y;
}
```

```
void Rectangle::set_values (int x) {
    width = x;
    height = x;
}
```

```
int main () {
    Rectangle rect;
    rect.set_values(3);
    cout << "area: " << rect.area();
    rect.set_values(3,4);
    cout << "area: " << rect.area();
    return 0;
}
```

# Inheritance

- C++ classes can be extended creating new classes which retain characteristics of the original class
- **Base class**
  - Original class from which the derived class inherits the members
- **Derived class:**
  - The derived class inherits the members of the base class
  - Additionally can have its own new members
- Derived classes are defined using the following syntax :

```
class derived_class_name: public base_class_name
```

```
class Polygon {  
protected:  
    int width, height;  
public:  
    void set_values (int a, int b)  
        { width=a; height=b;}  
};
```

```
class Rectangle: public Polygon {  
public:  
    int area ()  
        { return width * height; }  
};
```

New member

Derived class

# Constructors & Destructors

- Special functions used for building/destroying an object
- A class can have several constructors
- The constructors have the same name as the class
- The constructors have the same name as the class but have a leading ~

**Rectangle class has 2 constructors**

**Rectangle class destructor**

```
class Rectangle {  
    int width, height;  
public:  
    Rectangle ();  
    Rectangle (int,int);  
    int area (void) {return (width*height);}  
    ~Rectangle () {}  
};  
  
Rectangle::Rectangle () {  
    width = 5;  
    height = 5;  
}  
  
Rectangle::Rectangle (int a, int b) {  
    width = a;  
    height = b;  
}
```



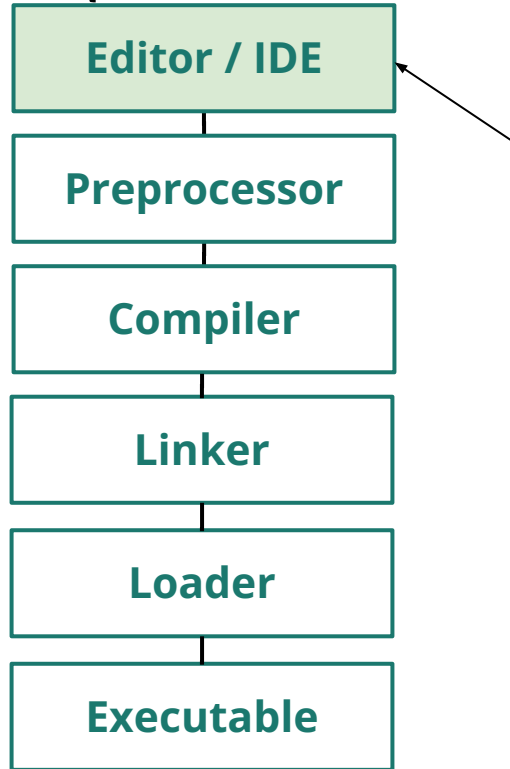
# Compilation

# How is a C++ code compiled

## Compiler

- Utility program that translate user code into machine code
- Compilation is performed in several steps
- Simplest command :

```
g++ helloWorld.cpp -o helloWorld
```



## Editor / IDE

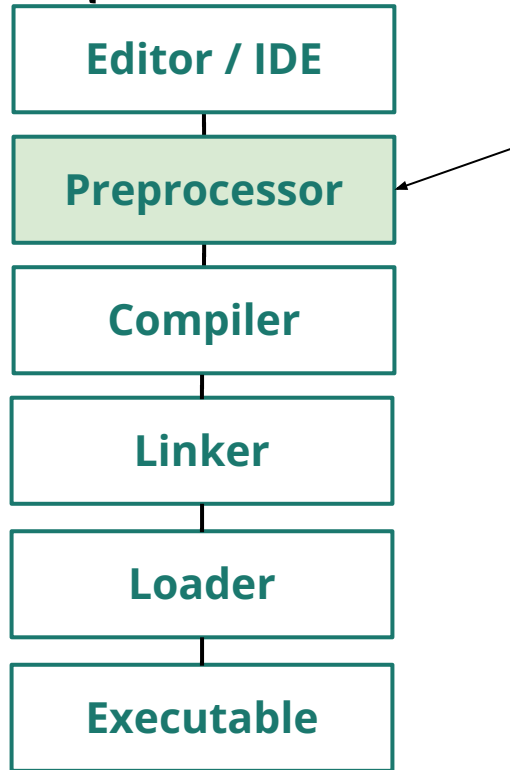
- First step is to write our C++ code
- We can use a text editor (vim, nano etc.) or an IDE (Integrated development environment i.e. vscode, eclipse etc. )

# How is a C++ code compiled

## Compiler

- Utility program that translate user code into machine code
- Compilation is performed in several steps
- Simplest command :

```
g++ helloWorld.cpp -o helloWorld
```



## Preprocessor

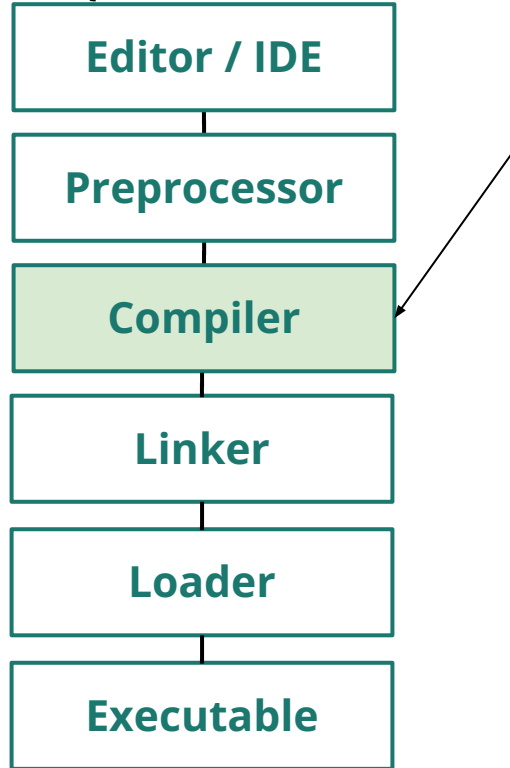
- Performed before compilation
- The result of preprocessing is a single file which is then passed to the actual compiler
- The preprocessor handles the # directives (macros, includes) and creates the source code.

# How is a C++ code compiled

## Compiler

- Utility program that translate user code into machine code
- Compilation is performed in several steps
- Simplest command :

```
g++ helloWorld.cpp -o helloWorld
```



## Compiler

- The compilation takes place on the preprocessed files.
- The compiler parses the C++ source code and converts it into assembly & machine code.
- The produced object file contains the compiled code (in binary form)

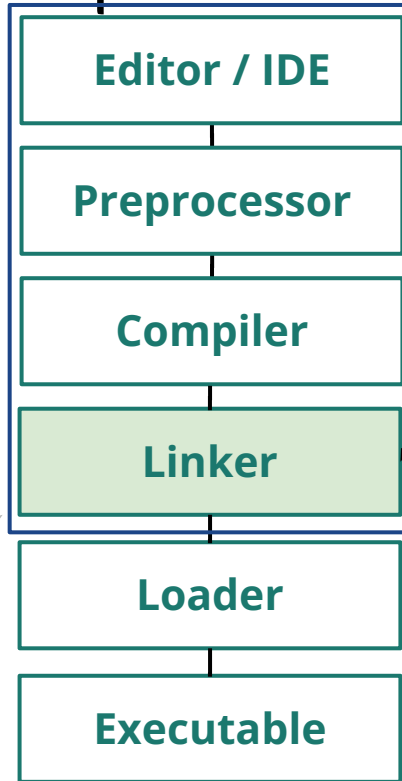
# How is a C++ code compiled

## Compiler

- Utility program that translate user code into machine code
- Compilation is performed in several steps
- Simplest command :

```
g++ helloWorld.cpp -o helloWorld
```

Build steps



## Linker

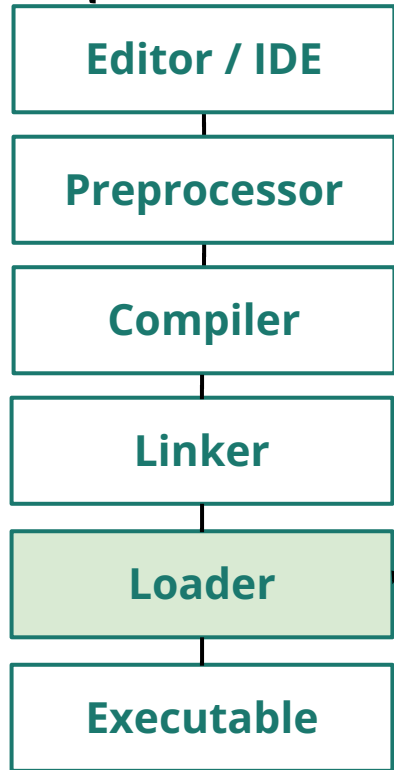
- Takes all the object files generated by the compiler and combine them into a single executable program.
- Links external library files.
- Makes sure all cross-file dependencies are properly resolved.

# How is a C++ code compiled

## Compiler

- Utility program that translate user code into machine code
- Compilation is performed in several steps
- Simplest command :

```
g++ helloWorld.cpp -o helloWorld
```



## Loader

- Loader is generally part of the operating system
- Loads the executable into memory

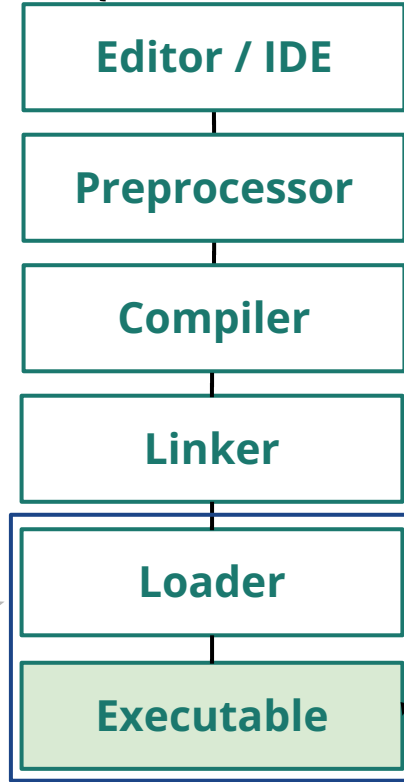
# How is a C++ code compiled

## Compiler

- Utility program that translate user code into machine code
- Compilation is performed in several steps
- Simplest command :

```
g++ helloWorld.cpp -o helloWorld
```

Run steps



## Executable

We are now ready to run the executable!

To do so :  
`./helloWorld`

# Wrapping-up



# Overview of today's lecture

- We learnt about what a scope is in C++
- We discussed about namespaces
  - std → standard library in C++ provides many facilities that can be used
- We learnt about compound data types
- We got familiar with Object Oriented programming and classes
- We got familiar with the C++ compilation chain

# Assignment for next week

- Assignment can be found here :

<https://github.com/ckoraka/tac-hep-gpus>

- To clone :

- `git clone git@github.com:ckoraka/tac-hep-gpus.git`

- **Due Friday February 10<sup>th</sup>**

- Please upload assignment here :

- <https://pages.hep.wisc.edu/~ckoraka/assignments/TAC-HEP/>

- Upload only 1 .pdf file with all exercises

- If you also have your code on git, please add the link to your repository in the pdf file you upload.

# Next week

- We will get introduced to the CUDA programming model :
  - Concept of parallelization
  - Threads & blocks
  - CUDA core syntax
  - GPU memory hierarchy
  - Basic memory management
  - Error handling





Back-up



# Resources

- cplusplus docs [link](#)
- cppreference docs [link](#)
- CERN C++ course [link](#)