



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

GPU programming module

Week 2 : Introduction to C++

Lecture 3 - September 17th 2024



What we learnt in the previous lecture

- Hardware accelerators can be used in combination with CPUs to executing specific tasks more efficiently
- GPUs are hardware accelerators that follow the SIMT paradigm
 - Have thousands of cores and therefore can provide massive parallelization
 - Can provide more FLOPS/watt than CPUs
- The next decades will pose a significant computing challenge for HEP experiments
 - Many HEP experiments are already exploring the use of accelerators and heterogeneous computing

Today : Some brushing up of C++

- History of C++
- Core syntax
- Variables & Operators
- Control instructions
- Functions

History of C++

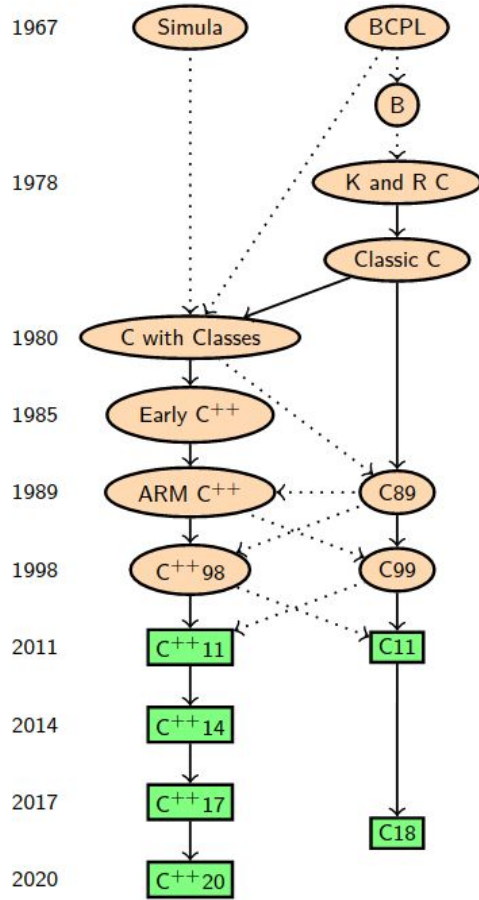
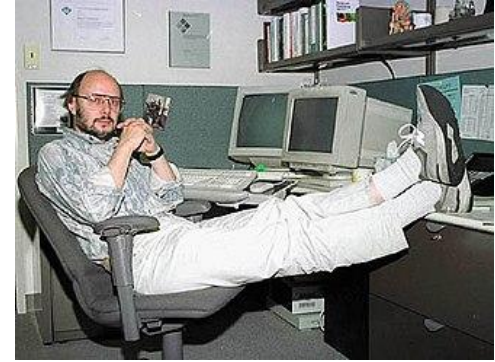


Image taken from [3]



C inventor



C++ inventor

- Multi-paradigm programming language that supports object-oriented programming
- Based on C language developed by Dennis Ritchie
- Designed at Bell labs in the late 70s by Bjarne Stroustrup

Why is C++ so widely used

- **Fast**

- C++ is a compiled language unlike other languages e.g. python / Java which are interpreted

- **Object oriented**

- Modular and reusable code

- **Low level**

- Closer to hardware / allows low level optimization

- **Many available libraries**

- Standard Template Library (STL) provides template that can be used from the developer and make coding faster



Core syntax

Structure of a C++ program

Let's look into the main structure and components of a C++ program by checking out a simple program that print out "Hello world" :

```
#include <iostream>

int main()
{
    std::cout << "Hello World!";
}
```


Structure of a C++ program

#include <iostream>

- Special lines interpreted before compilation
- Instruct the preprocessor to include a section of standard C++ code
- e.g. *iostream* allows standard I/O operations



```
#include <iostream>

int main()
{
    std::cout << "Hello World!";
}
```

Structure of a C++ program

`#include <iostream>`

- Special lines interpreted before compilation
- Instruct the preprocessor to include a section of standard C++ code
- e.g. *iostream* allows standard I/O operations

`int main()`

- Special C++ function
- All C++ programs start execution from main

```
#include <iostream>

int main()
{
    std::cout << "Hello World!";
}
```

Structure of a C++ program

`#include <iostream>`

- Special lines interpreted before compilation
- Instruct the preprocessor to include a section of standard C++ code
- e.g *iostream* allows standard I/O operations

```
#include <iostream>

int main()
{
    std::cout << "Hello World!";
}
```

`int main()`

- Special C++ function
- All C++ programs start execution from main

`std::cout << "Hello World!";`

- C++ statement :
 - `std::cout` **standard character output**
 - **"Hello World"** string of characters that will be outputted
 - `<<` insertion operator
 - `;` every statement should end with a semi-colon

Structure of a C++ program

`#include <iostream>`

- Special lines interpreted before compilation
- Instruct the preprocessor to include a section of standard C++ code
- e.g *iostream* allows standard I/O operations

`{.....}`

Curly braces enclose the body of a function

```
#include <iostream>

int main()
{
    std::cout << "Hello World!";
}
```

`int main()`

- Special C++ function
- All C++ programs start execution from main

`std::cout << "Hello World";`

- C++ statement :
 - `std::cout` **standard character output**
 - **"Hello World"** string of characters that will be outputted
 - `<<` insertion operator
 - `;` every statement should end with a semi-colon

Comments in C++

Line comment

```
// This is a line comment
```

Block comment

```
/* This is a block comment  
It can span on more than 1 lines  
*/
```

Useful and important tool since
it makes code more readable
and easier to share

doxygen compatible comments : tool for generating documentation from annotated C++ sources [[documentation](#)]

```
/**  
 * doxygen compatible comments  
 * \fn bool isOdd(int i)  
 * \brief checks whether i is odd  
 * \param i input  
 * \return true if i is odd, otherwise false  
 */
```



Variables and operators



Variables

- **Variable** → portion of memory used to store a value.
- Name of variable → **Identifier**
 - Combination of letters, digits, or underscore characters
 - C++ keywords cannot be used

Size	Number
8-bit	2^8
16-bit	2^{16}
32-bit	2^{32}
64-bit	2^{64}

Variable types	Names	Example
Character	char char16_t char32_t	<code>char c = 'a'</code> 16-bit wide 32-bit wide
Integer	int (un)signed char (un)signed int short/long (int)	<code>int i = 2023</code> 8-bit wide 32-bit wide 16-bit wide
Floating-point	float double	<code>float f = 2.023f</code> 32-bit wide <code>double d = 2.023</code> 64-bit wide
Boolean	bool	<code>bool a = true</code> <code>bool b = false</code>

Operators

Operators can operate on variables

There are many types some of which are summarized in the table

Types	Operators	Usage
Assignment operator	=	Assign value to variable
Arithmetic operators	+, -, *, /, %	Mathematical operations
Compound assignment	+=, -=, *=, /=, %= >>=, <<=, &=, ^=, =	modify the current value by performing an operation
Increment and decrement	++,--	equivalent to +=1 & -=1
Relational and comparison	==, !=, >, <, >=, <=	Comparisons of two expressions
Logical	!, &&,	not / and / or
Conditional ternary operator	?	Returns different value if expression is true or false Syntax : condition ? result1 : result2
Bitwise operators	&, , ^, ~, <<, >>	modify variables considering the bit patterns

Operators

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

int main ()
{
    int a,b,c;
    bool d;

    // Assignment & arithmetic
    a=2;
    b=7+3;
    // Assignment, logical & comparison
    d = !(7 == 5);
    // Conditional & relational
    c = (a>b) ? a : b;
    // Compound assignment & increment
    a+=2;
    b = ++a;
    // What is the value of each variable?
    cout <<" a : " << a << '\n';
    cout <<" b : " << b << '\n';
    cout <<" c : " << c << '\n';
    cout <<" d : " << d << '\n';
}
```

**What are the values
of variables a,b,c & d?**

**Let's check using
[onlinegdb](#)**

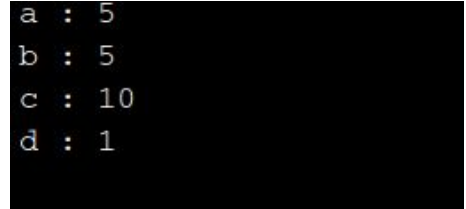


Operators

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

int main ()
{
    int a,b,c;
    bool d;

    // Assignment & arithmetic
    a=2;
    b=7+3;
    // Assignment, logical & comparison
    d = !(7 == 5);
    // Conditional & relational
    c = (a>b) ? a : b;
    // Compound assignment & increment
    a+=2;
    b = ++a;
    // What is the value of each variable?
    cout <<" a : " << a << '\n';
    cout <<" b : " << b << '\n';
    cout <<" c : " << c << '\n';
    cout <<" d : " << d << '\n';
}
```



```
a : 5
b : 5
c : 10
d : 1
```

Control flow

Flow control instructions

Statement :

- Individual instructions of the program
- End with a semicolon (;)
- Executed in the order in which they appear in the program

Control instructions :

- Redirect the flow of a program
- Many types - some include :
 - if/else
 - Conditional operator (?)
 - switch
 - for loop / range based loops / while loops

if...else

```
if (x > 0)
    cout << "x is positive";
else if (x < 0)
    cout << "x is negative";
else
    cout << "x is 0";
```

- **Syntax**: if (*condition*) statement
 - *condition* is evaluated
 - If *condition* true, statement is executed
- `else` and `else if` are optional
- `else if` can be repeated
- braces are optional if there is a single instruction

switch

```
switch (oper) {  
    case '+':  
        cout << a + b;  
        break;  
    case '-':  
        cout << a - b;  
        break;  
    case '*':  
        cout << a * b;  
        break;  
    case '/':  
        cout << a / b;  
        break;  
    default:  
        cout << "Incorrect operator";  
        break;  
}
```

Let's add this to our
code on [onlinegdb](#)

- **Syntax**:
switch(*identifier*) {
 case c1 : *instructions1*; break;
 case c2 : *instructions2*; break;
 ...
 default : *instructionsd*; break;
}
- switch evaluates expression / checks if it is equivalent to case c1
- If true, *instructions1* are executed
- After break the program jumps to the end of switch
- Execution carries on with the next case if no break is present
- Default is optional

for loop

```
for (int n=10; n>0; n--) {  
    cout << n << ", ";  
}
```

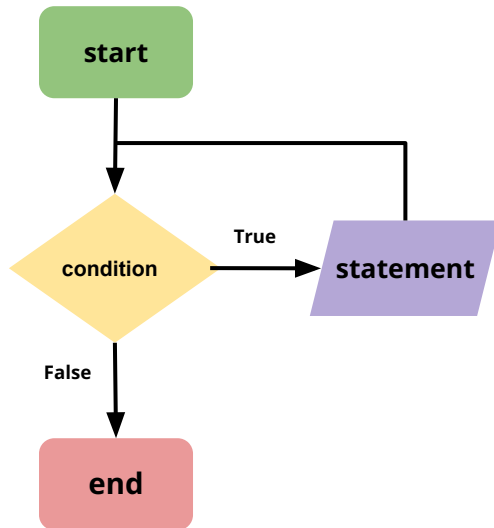
```
for ( n=0, i=100 ; n!=i ; ++n, --i ){  
    cout << "n=" << n << " and i=" << i << "\n";  
}
```

```
for(int i = 0, j = 0 ; i < 10 ; i++, j = 2*i)  
    cout << "2*" << i << " is " << j << "\n";
```

Syntax:

for(initializations; condition; increments) {statement}

- Initializations and increments are separated by a comma
- Initializations can contain declarations



Range based loop

```
string str {"Hello World!"};
for (char c : str)
{
    cout << "[" << c << " ";
}
```

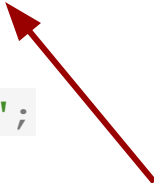
Syntax:

for (type iterator : container) statement;

- iterates over all the elements in the container
- simplifies loops tremendously especially with STL container

Range based loop

```
string str {"Hello World!"};  
for (char c : str)  
{  
    cout << "[" << c << "];"  
}
```



Syntax:

for (type iterator : container) statement;

- iterates over all the elements in the container
- simplifies loops tremendously especially with STL container

Exercise : Lets try this out!

- Open a new window in [onlinegdb](#)
- Create an array with 5 elements - your favorite integer numbers
- Calculate their sum using a range based loop and print out the result!

While loop

```
int n = 10;
while (n>0) {
    cout << n << ", ";
    --n;
}
```

```
int n = 10;
do {
    cout << n << ", ";
    --n;
} while (n>0);
```

Syntax :

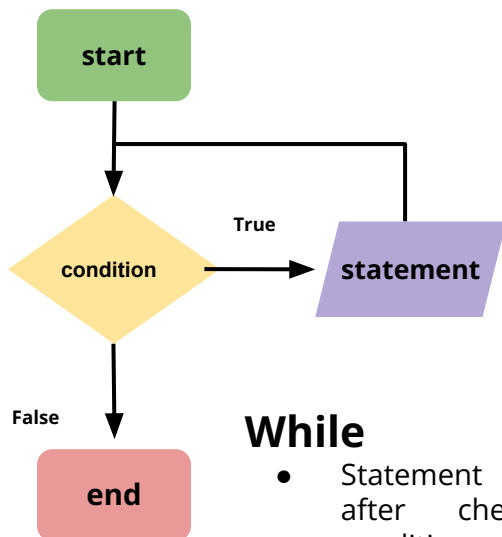
while (*condition*) statement

- Condition evaluated **before** first iteration

do statement while (*condition*);

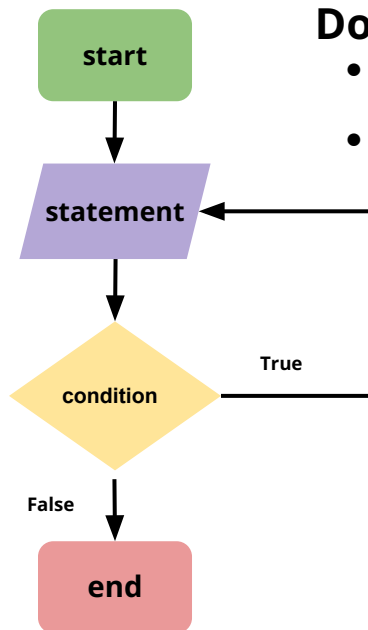
- Condition evaluated **after** first iteration

While loop



While

- Statement is executed after checking the condition
- Similar to for loop flowchart



Do — While

- Statement is always executed once
- Condition is checked after the statement is executed

Functions

What is a function

- Group of statements that is given a name and can be called from some point of the program
- Allow to structure programs in segments of code
- Make code reusable

Syntax :

`type name ( parameter1, parameter2, ...) { statements }`

- **type** : type of the value returned by the function.
- **name** : function identifier
- **parameters** : type followed by an identifier, (e.g. `int parameter1`) arguments are passed to the function from the location where the function is called from.
- **statements** : block of statements surrounded by curly braces

Some examples of functions

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

Function that takes two arguments
and returns an integer

```
int addition (int a, int b)
{
    int r;
    r=a+b;
    return r;
}
```

Function that takes one arguments
and returns nothing (void)

```
void print (int a)
{
    cout<<"The number is " <<a<<endl;
}
```

```
int main ()
{
    int z;
    z = addition (5, 3);
    print (z);
}
```

main function → program always
starts from main

Function parameters

Function parameters can be passed :

- **By value**
- **By reference**



```
int addition (int a, int b)
{
    int r;
    r=a+b;
    return r;
}
```



```
int addition (int &a, int &b)
{
    int r;
    r=a+b;
    return r;
}
```

Function parameters

- **Passed by value :**

- Parameters are copied into the variables represented by the function parameters
- Modifications of these variables within the function has no effect on the values of the variables outside the function
- By default arguments are passed by value (= copy, good for small types, e.g. numbers)

```
int addition (int a, int b)
{
    int r;
    r=a+b;
    return r;
}
```


Function parameters

- **Passed by reference :**

- also called pass by address
- The parameters a and b are still local to the function, but they are *reference* variables (i.e. nicknames to the original variables passed)
- Allows the function to modify a variable without having to create a copy of it
- references are preferred to avoid copies
- **const** can be used for safety e.g.
 - `int addition (const int &a)`
 - Ensures that variable cannot be changed

```
int addition (int &a, int &b)
{
    int r;
    r=a+b;
    return r;
}
```

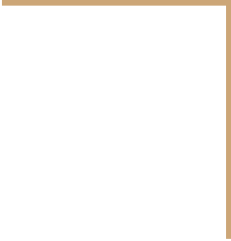
Function parameters

- **Passed by reference :**

- also called pass by address
- The parameters a and b are still local to the function, but they are *reference* variables (i.e. nicknames to the original variables passed)
- Allows the function to modify a variable without having to create a copy of it
- references are preferred to avoid copies
- **const** can be used for safety e.g.
 - `int addition (const int &a)`
 - Ensures that variable cannot be changed

Exercise : Lets try this out!

- Write a function that takes two integer arguments and returns nothing
- Change the value of each variable to its square
- Print the values of the argument in the main function
- Try passing the variables by value and by reference - what do you observe?
- Try making a variable const. What do you observe?



Wrapping-up



Overview of today's lecture

- Learnt about the history of C++ and why it is widely used
- Brushed up C++ core syntax
- Went through the different variables types & operators
- Were reminded of C++ flow control instructions & functions

Tomorrow

- We will continue with :
 - Scopes / namespaces
 - Compound data types
 - Object Orientation
 - The C++ compilation chain





Back-up



Resources

1. cplusplus docs [link](#)
2. cppreference docs [link](#)
3. CERN C++ course [link](#)