Practice Sessions
Astrophysical Simulations

Part 3: C++ language basics

GHENT
UNIVERSITY

Master of Science in Physics and Astronomy
2018-2019
Peter Camps
peter.camps@ugent.be
S9, 1st floor, office 110.014

History of the C/C++ language family

History of C

- 1978: Kernighan & Ritchie publish “The C Programming Language”
- 1989/1990: standardization committees adopt the “ANSI C standard”
- 1999: updated C standard commonly referred to as “C99”
- 2011: updated C standard commonly referred to as “C11”

History of C++

- 1985: Bjarne Stroustrup publishes “The C++ Programming Language”
- 1998: ISO committee adopts the “C++98” standard
  - very widely implemented and used
- 2011: ISO committee adopts the “C++11” standard
  - substantial additions to the language and the standard library
  - implemented by all recent compilers
- 2014: ISO committee adopts the “C++14” standard
  - minor fixes and improvements; implemented by most compilers

The C++ language

Key features

- Compiled language (translated to machine code)
- Object-oriented (classes, methods, data-encapsulation, inheritance, …)
- Generic (templates with data type parameters)

Design goals

- Systems programming for resource-constrained and large systems
- Performance and efficiency (close to the hardware)
- Software design flexibility (modular, scalable, maintainable)

Disadvantages

- Limited scientific functionality in standard library
  (as compared to for example numpy and scipy in Python)
- Includes low-level features that should not be used by a high-level programmer (but are key for efficient library programming)
- Includes legacy features that should no longer be used at all

Applicability of C++ language features

- In the context of this course, C++ means C++11 or C++14
- This is not a programming course; the focus is on physics and methods
- So keep your programs nice and simple

Label	Description
Recommended	This language feature is the proper choice for use in your programs for this course
Not needed	You do not need this language feature in the programs for this course; you should use it only if you have a good reason to do so *and* if you know how to use it properly
Do not use	Do not use this language feature in the programs for this course; it is either a feature intended for low-level programming or it is a legacy feature that should no longer be used at all

Using these language features will almost certainly result in debugging nightmares, and it will make me very nervous

Program structure

```
#define _USE_MATH_DEFINES
#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std;

double f(double x) { return x*x; }

int main() {
    • • •
    double fx = f(5.0);
    • • •
}
```

Forces Microsoft compilers to include non-standard math symbols, e.g. M_PI

Included headers declare facilities provided by the standard library and used in your code

Allows omitting "std::" in front of standard library function and type names

Functions must be declared before they are used

The main function is called when your program starts

A semicolon terminates every statement

The main function automatically returns exit code for "success"

Curly braces are used to indicate scope; in this case the scope of the main function

Multiple files

functions.hpp

```
double f2(double x);
double f3(double x);
```

Header file provides function declarations (i.e. the **interface**)

functions.cpp

ccp file provides function definitions (i.e. the **implementation**)

```
#include "functions.hpp"

double f2(double x) { return x*x; }
double f3(double x) { return x*x*x; }
```

main.cpp

Other source files using the functions include just the interface rather than the complete implementation

```
#include "functions.hpp"

int main() {
    • • •
    double y = f1(5.0) + f2(3.77);
    • • •
}
```

Multiple files not needed

Uncouples usage and implementation, which is very important for large-scale programs

Numeric data types

Other numeric types not needed

```
int main() {
    int k = -57;
    int m = 3/2; // m = 1

    size_t index = 0;
    index++; // index = 1

    double x = 1.83e-12;
    double y = x * k;
    • • •
}
```

int is the integer data type of choice; integer division produces an integer result!

size_t is an unsigned integer data type that spans the addressable space; use for indexing (because indices can 't be negative)

double is a double-precision floating point data type; most scientific calculations need the extra number of significant digits

Data type	Size on 64 bit systems	Corresponding range
int	32 bits (4 bytes)	> ± 2×10^9
size_t	64 bits (8 bytes)	2^{64}-1 (addresses many Terrabytes…)
double	64 bits (8 bytes)	> 15 digits precision; exponent up to ±300

String data types

Other string types not needed

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

int main() {
    char ch = 'a';
    char newline = '\n';

    string s1 = "Hello";
    string s2 = "World";
    string str = s1 + " " + s2;

    int k = -57;
    string sk = to_string(k);
    • • •
}
```

std::string is a library data type so you need to include its header (directly or indirectly)

char holds a single character; character constants use single quotes; escape sequences specify special characters

string holds a sequence of characters; string constants use double quotes; the plus operator concatenates strings

The *to_string* library function returns a human-readable string representation for any numeric value

String constants have a legacy C data type (const char*) rather than std::string, so you can't concatenate two string constants

Do not use the const char* data type; use std::string instead

Boolean data type and if statement

```
double posdiff(double x, double y) {

    if (x < y) {
        return y-x;
    } else {
        return x-y;
    }
}

int main() {
    bool flag = false;
    • • •
    if (flag && x<y) result += x;
    • • •
}
```

The *if* statement expects an expression of type *bool*

There can be multiple statements inside the scope indicated by the curly braces

The *else* clause is optional

A variable of type *bool* has a value *false* or *true*

For a single statement you can omit the curly braces; however be careful when chaining if-else statements, or when you add an extra statement later on

You can create logical expressions with the operators ! (not), && (and), || (or)
Do not use the bit-wise operators ~, &,|

Variable initialization

- Variables of primitive data types (including *int*, *size_t*, *double*, *bool*, *char*) are not automatically initialized – they receive a random value
  - » Always initialize variables in the declaration statement

```
double a = 99.;
int index = 0;
```

  - » Exception - when you stream data in on the very next line

```
double x, y;
infile >> x >> y;
```

Not initialized -> random value!

- Objects of library or user defined types are initialized to a default value

```
string buffer;
```

Initialized to empty string

Do not separate declaration and initialization

Looping constructs: the for statement

```
// return sum of first n positive integers
int sum(int n) {
    int result = 0;
    for (int i=0; i!=n; ++i) {
        result += i+1;
    }
    return result;
}
```

Declare and initialize loop variable; performed once at start of loop

Loop condition is tested *before* execution of each iteration; loop exits if condition is false

Increment loop variable; performed *after* execution of each iteration

You can use any expression (including the empty statement) for each of these three items, but most for loops have the simple form shown here

Looping constructs: the while statement

```
bool success = true;
while (success) {
    // code changes value
    // of succes at some point
    • • •
}
```

Loop condition (can be any expression) is tested *before* execution of each iteration; loop exits if condition is false

This code must change the loop condition or the loop will never exit

```
while (true) {
    • • •
    if (!success) break;
    • • •
}
```

The *break* statement causes the innermost loop (*while* or *for*) to exit; allows exit at any point in the loop body

You do not need the *switch* statement, but it is perfectly acceptable; Do not use the *do-while* looping statement

Do not use the *goto* and *continue* statements

Passing function arguments and results

```
// append n characters to string
string extend(string str, char ch, int n) {
    for (int i=0; i!=n; ++i) {
        str += ch;
    }
    return str;
}
```

Arguments are passed by value, i.e. a copy of the caller's values is placed in the corresponding function argument

Return values are passed by value as well, i.e. a full copy is returned

Thus it is perfectly legal to change the value of the arguments within the function; this does not affect the caller's values

Arguments passed by value can be any expression

```
int main() {
    cout << extend("pipo", '*', 3) << endl;
}
```

For complex objects, the copying during argument passing by value can represent an important performance hit. Mechanisms to avoid this problem will be discussed later.

Math functions in standard library

```
#include <cmath>
```

Function	Description
pow(x,y)	Power x^y (use x*x for square and sqrt(x) for square root)
sqrt(x)	Square root
log(x), log10(x)	Natural and 10-based logarithm
exp(x)	Exponential e^x
sin(x), cos(x)	Trigonometric functions
abs(x)	Absolute value
floor(x), round(x)	Round to the next lowest and nearest integer

and many more…

Output streams

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

double f(double x) { return x*x; }

int main() {
    cout << "Hello World" << endl;

    for (int i=0; i!=10; ++i) {
        cout << i << ' ' << f(i) << endl;
    }

    cout << "Done writing." << endl;
}
```

std::cout is the standard output stream, which is by default connected to the terminal

<< is the output stream insertion operator; it takes any standard data type and inserts a string representation into the stream

std::endl inserts a newline character '\n' into the stream **and** flushes the stream buffers to the output device

File output streams

```
#include <fstream>
#include <iomanip>
using namespace std;

double f(double x) { return 1/x; }

int main() {
    ofstream outfile("test.txt");
    outfile << setprecision(12);
    for (int i=0; i!=10; ++i) {
        outfile << i << ' ' << f(i) << '\n';
    }
    outfile.close();
}
```

Construct a variable called *outfile* of type *std::ofstream* (output file stream), with the filename specified as a string argument

std::setprecision tells the stream to format floating point numbers with the specified precision

An output file stream closes automatically when it goes out of scope (here at the end of the main function) but it is cleaner to close it explicitly

File input streams

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

int main() {
    ifstream infile("test.txt");
    while (true) {
        double x, y;
        infile >> x >> y;
        if (!infile.good()) break;
        cout << x << ' ' << y << endl;
    }
    infile.close();
}
```

Construct a variable called *infile* of type *std::ifstream* (input file stream), with the filename specified as a string argument

>> is the input stream extraction operator; it reads a string representation from the stream for any standard data type

ifstream::good returns false as soon as the end of the input stream is reached or an error occurs

An input file stream closes automatically when it goes out of scope (here at the end of the main function) but it is cleaner to close it explicitly

```
2  0.5
3  0.333
4  0.25
```

Values in file separated by white space (space, tab, newline)

Questions?