Basics // Modern C++ in a minute

Main program

> write in editor, save
> compile `hello.cpp`
> execute `hello`

```
make hello
./hello
Hello, world!
```

```
#include <iostream>
// ... declarations,
// function definitions

int main()
{
  // ... statements
  std::cout << "Hello, world!\n";
}
```

Data types and values

	`auto`	deduced from assigned value
logical value	`bool`	`true`, `false`
integer number	`int`, `long`	`-1` (dez), `0b10` (bin), `0123` (oktal), `0x1ABC` (hex), `2017L` (long)
floating point	`float`, `double`	`-0.5f`, `31.f`, `3.1415`, `-1.602e-19`
character	`char`	`'A'`, `'7'`, `'*'`, `'\n'` (new line), `'\t'` (tab), `'\\'`, `'\''`, `'\"'`, `'\x4A'` (ASCII)
character string	`std::string`	`"Hallo"`
placeholder	`void`	"empty", no type applies

Constants

```
const auto expectedAnswer = 42;
const double LIGHTYEAR  = 9.46e12; // m
const double MASS_EARTH = 5.98e24; // kg
```

Variables

```
int x, y = 3;
x = y + 5;
++x; y--;
```

Operations

arithmetic	`x+y x-y x*y x/y x%y` (modulus)
add 1 / subtract 1	`++x --x` (before evaluation) `x++ x--` (after evaluation)
assign short hand	`x = wert` `x += wert` d.h. `x = x + wert`
compare	`x<y x<=y x>=y x>y` `x==y` (equal), `x!=y` (not equal)

Logic

a	b	NOT !b	AND a && b	OR a || b
false	false	true	false	false
false	true	false	false	true
true	false		false	true
true	true		true	true

Standard output (console)

```
std::cout << x << '\n';
```

Standard input (keyboard)

```
std::cin >> x;
```

Function definition

```
resultType functionName(parameterList)
{ // ... statements
  return result;
}
```

comma-separated parameter list:
`Type parameter` or `Type& parameter` (reference)

Function declaration and call

```
resultType functionName(parameterList);
// or definition

result = functionName(argumentList);
```

argument list: number and type of arguments according to parameter list

Control statements (iteration/selection)

```
for (int i = 0; i < 5; ++i)
{
  std::cout << i << "  ";

}
```

```
for (auto n : {1,4,9,16})
{
  std::cout << n << "  ";
}
```

```
while (x > 0)   // test before
{  --x;
}
```

```
if (x < 5)
{  ++x;
}
else   // optional
{      // do something else
}
```

Selected libraries

<string> Character strings

instead of `char` arrays

```
std::string s = "Hello";
s += ", world!";
```

`s.size()`	number of chars
`s[index]`	access to one char, `index < s.size()`
`s == s2`	same content?
`s < s2`	`s` before `s2`?
`s + s2`	concatenation

<vector> Array containers

of any type, variable number of elements

```
std::vector<int> v = { 1, 2, 3 };
for(auto& e : v) e += 10;
```

`v.size()`	number of elements
`v[index]`	access one element, `index < v.size()`
`v == v2`	same content?
`v < v2`	`v` before `v2`?
`v.push_back(wert)`	append value at the end

<cmath> Math functions

`fabs(x)`	absolute value $\|x\|$
`sqrt(x)`	$\sqrt{x}$
`pow(x,y)`	x^y
`exp(x)`	e^x
`log(x)`	$\ln x$
`sin(x)`	trigonometric functions
`cos(x)`	inverse `asin(x)`, `acos(x)`
`tan(x)`	inverse `atan(x)`, `atan2(y,x)`

<algorithm> Algorithms

on half-open range `[begin, end)`

```
std::sort(v.begin(), v.end());
std::sort(s.begin(), s.end());

std::for_each(v.begin(), v.end(),
    [](int e) { std::cout << e << ' '; }
  );
```

<fstream> File I/O streams

```
std::ofstream output("filename");
if (!output) /* can't open file */;
output << "Hello 123\n";
output.close();

std::ifstream input("filename");
if (!input) /* can't open file */;
input >> s >> x;
```

Open file automatically closes at end of block.

<sstream> String streams

```
std::ostringstream outbuffer;
outbuffer << "Hello 123";
s = outbuffer.str();

std::istringstream inbuffer(s);
while (inbuffer >> s >> x)
{ // ... process data
}
```

Stream as condition: „read was successful“.

Classes

User-defined types

```
class Particle      // or struct ...
{
  // ... attributes
public:
  // ... constructors, destructor
  // ... methods (declaration)
};
```

Attributes

```
private: // no access from outside class
  double x, y;
protected: // access for derived classes
  double mass;
```

Constructors, Destructor

```
  Particle (double x, double y, double m)
  : x(x), y(y), mass(m)
  {  // ... initialize attribute values
  }

  // destructor, if needed
  ~Particle () { /* free ressources */ }
```

Methods

```
void Particle::move(double dx, double dy)
{
  x += dx; // this->x, if shadowed
}
```

defined outside class with class name qualified
`const`-methods enforce read-only access

```
  double whereX() const // in class
  {
    return x;
  }
```

Objects

- instances of class type,
- created by constructor call,
- accessible through variable,
- react to method call (message).

```
  Particle sun(0, 0, 3.32e5*MASS_EARTH);
  sun.move(10000 * LIGHTYEAR, 0);
```

Inheritance

derive class from base class(es)

```
class BlackHole : public Particle
{
  // ... additional attributes
public:
  Blackhole(double x, double y, double m)
  : Particle(x, y, m)
    // base constructor call
  { // ...
  }
  // ... overriding / new methods
}
```

override method (declared `virtual` in base class)

```
  void move(double dx, double dy)
  { // call base method (optional):
    Particle::move(dx, dy);
    // ...
  }
```

Pointers and dynamic polymorphism

Destructor `virtual ~Particle()` is required.

```
  Particle* ufo = new Blackhole(0, 0, 1);
  ufo->move(0, 1);
  delete ufo;
```

Prefer "smart pointers" like `std::shared_ptr<T>`
or `std::unique_ptr<T>` from `<memory>`.

```
  std::shared_ptr<Particle> ufo =
    std::make_shared<Particle>(0, 0, 1));
  // no delete
```

downcast

```
  if (auto hole =
        std::dynamic_pointer_cast
                  <Blackhole>(ufo))
  { hole->collapse();
  }
```

Miscellaneous

Exception handling

```
try
{ // section possibly throws exception
}
catch (Particle& e) { /* handle it */ }
```

Throw exception, if

- action cannot continue and
- situation cannot be rasolved locally.

```
if (sun.whereX() == 0) throw sun;
```

Overloading operators

```
bool operator<(Particle a, Particle b)
{ return a.mass < b.mass;
}
```

grant access to `private` parts for `friend`

```
  // in class Particle:
  friend
  bool operator<(Particle a, Particle b);
```

stream output

```
std::ostream&
operator<<(std::ostream& os, Particle p)
{
  return os << p.x << ' '
            << p.y << ' '
            << p.mass;
}
```

stream input (change value after success only)

```
std::istream&
operator>>(std::istream& is, Particle& p)
{ double x, y, mass;
  if (is >> x >> y >> mass)
    p = Particle(x,y, mass);
  return is;
}
```

Arrays

```
const int m = 3, n = 4;  // fixed number
Type array[n];  // array[0]...array[n-1]
Type matrix[m][n];  // m rows, n columns
```

Parameter-less constructor `Type()` is required.

`matrix[0][0]`	...	`matrix[0][n-1]`
...		...
`matrix[m-1][0]`	...	`matrix[m-1][n-1]`

Functions templates

```
template<typename T>
void swap(T& a, T& b)
{
  T t(a); a = b; b = t;
}
```

```
swap(x, y);
```

Class templates

```
template <typename T, int N>
struct Array
{
  T elements[N];
};
```

```
Array<Particle,9> solarsystem;
solarsystem.elements[0] = sun;
```

Namespaces

```
namespace Physics
{
  const double e = 1.602e-19; // As
}
namespace Math
{
  const double e = 2.71828;
}
```

```
std::cout << Physics::e << ' '
          << Math::e << '\n';

using Math::e;
std::cout << Physics::e << ' '
          << e << '\n';
```