Hello, World!

Hello, World!

```
#include <iostream>

int main( int argc, char * * argv)
{
  std::cout
   << "Hello, world!"
   << std::endl;
  return 0;
}
```

- Program entry point
  - Heritage of the C language
    - No classes or namespaces
  - Global function "main"
    - argc, argv (traditional names)
    - Command-line arguments
      - Split to pieces
    - Archaic data types
      - Pointer to pointer to char
      - Logically: array of strings
      - "Forgotten" const modifier
    - Modern C++ interface would be

```
std::size_t argc,
const char * const * argv
```

- or

```
std::size_t argc,
const std::string_view * argv
```

- or

```
const std::string_view * arg_begin,
const std::string_view * arg_end
```

- or

```
std::ranges::input_range auto arg
```

Hello, World!

```
#include <iostream>

int main( int argc, char * * argv)
{
  std::cout
   << "Hello, world!"
   << std::endl;
  return 0;
}
```

- std - standard library namespace
  - C++ uses :: instead of .
    - for non-objects on the left
  - cout - standard output
    - global variable
  - << - stream output
    - "misused" overloaded operator
    - not std::<< although inside std
      - Argument Dependent Lookup
  - endl - line delimiter
    - global function (archaic trick)
    - "\n" works too
      - But endl works in all 8/16/32-bit encodings
- iostream – standard header file
  - no .h or .hpp suffix!
    - non-standard header files shall have a suffix
  - cout, <<, endl declared inside
    - enclosed in namespace std { }

Hello, World!

- More than one module
  - Module interface described in a file
- .hpp - "header" file
  - The defining and all the using modules shall "include" the file
  - Text-based inclusion
  - Compiler config specifies folders

```
// world.hpp
#ifndef WORLD_HPP_
#define WORLD_HPP_

void world(); // a declaration

#endif
```

```
// main.cpp
#include "world.hpp"

int main( int argc, char * * argv)
{
  world();    // a use
  return 0;
}
```

```
// world.cpp
#include "world.hpp"
#include <iostream>

void world()  // the definition
{
  std::cout
    << "Hello, world!"
    << std::endl;
}
```

Hello, World! – in C++20 modules

- C++20 modules
  - export, module, import clauses
    - The suffix .ixx of *module interface files* is Microsoft-specific
- Standard library in new form
  - Only Microsoft implements the standard **import std.core** clause (in 2023)
  - Others do it differently: **import <iostream>**

```
// world.ixx

export module world;

export void world(); // a declaration
```

```
// main.cpp

import world;

int main( int argc, char * * argv)
{
  world();    // a use
  return 0;
}
```

```
// world.cpp

module world;

import std.core;

void world()  // the definition
{
  std::cout
    << "Hello, world!"
    << std::endl;
}
```

Hello, World!

- .hpp - "header" files contain
  - Definitions of types, classes, constants, inline functions and inline static variables
  - Declarations of non-inline functions and non-inline static variables
- .cpp – "source" files contain
  - Definitions of non-inline functions and non-inline static variables

```
// world.hpp
#ifndef WORLD_HPP_
#define WORLD_HPP_

#include <vector>
#include <string>

// a type definition:
using t_arg = std::vector<std::string>;

// a function declaration:
void world(const t_arg & arg);

#endif
```

```
// main.cpp
#include "world.hpp"

// a function definition:
int main( int argc, char * * argv)
{
  // a local variable definition:
  t_arg arg(argv + 1, argv + argc);
  // statements:
  world(arg);
  return 0;
}
```

```
// world.cpp
#include "world.hpp"
#include <iostream>

// a function definition:
void world( const t_arg & arg)
{
    std::cout << "Hello";
    // another local variable:
    for ( auto && x : arg )
        std::cout << "," << x;
    std::cout << std::endl;
}
```

Hello, World!

- .hpp - "header" files contain
  - Definitions of types, classes, constants, inline functions and inline static variables
    - Functions and static variables defined inside class/struct are implicitly inline
- Header-only modules
  - Not compiled alone
  - Not a module in binary sense

```cpp
// main.cpp
#include "world.hpp"

// a function definition:
int main( int argc, char * * argv)
{
  // a local variable definition:
  t_arg arg(argv + 1, argv + argc);
  // statements:
  world(arg);
  return 0;
}
```

```cpp
// world.hpp
#ifndef WORLD_HPP_
#define WORLD_HPP_

#include <vector>
#include <string>

// a type definition:
using t_arg = std::vector<std::string>;

// a function definition:
inline void world(const t_arg & arg)
{
    std::cout << "Hello";
    // another local variable:
    for ( auto && x : arg )
        std::cout << "," << x;
    std::cout << std::endl;
}

#endif
```

Declarations and definitions

- Declarations of functions
- Inline, implicitly inline, and non-inline definitions of functions
  - pure virtual functions have no definitions (called abstract in other languages)

```
// m.hpp
#ifndef M_HPP_
#define M_HPP_

void f1();                // global function declaration
inline void f2() {}       // inline global function definition
class C {                 // class definition
    void f3();               // member function declaration
    void f4();               // member function declaration
    virtual void f5();       // virtual member function declaration
    virtual void f6()=0;     // pure virtual member function declaration
    static void f7();        // static member function declaration
    void f8() {}             // implicitly inline member function definition
};
inline void C::f3() {}    // inline member function definition
inline void C::f7() {}    // inline static member function definition

#endif
```

```
// m.cpp
#include "m.hpp"

void f1() {}              // non-inline global function definition
void C::f4() {}           // non-inline member function definition
void C::f5() {}           // non-inline virtual member function definition
```

Declarations and definitions

- Declarations/definitions of variables
  - non-static member and local variables have only definitions
- Beware: (non-static member and local) variables of number and (raw) pointer types are NOT implicitly initialized!

```
// m.hpp
#ifndef M_HPP_
#define M_HPP_

extern int v1;        // global variable declaration
inline int v2;        // inline global variable definition
inline int v3=3;      // inline global variable definition
inline int v4(4);     // inline global variable definition
inline int v5{5};     // inline global variable definition
class C {             // class definition
    int v6;                    // member variable definition (not initialized)
    int v7=7;                  // member variable definition
    static int v9;             // static member variable declaration
    static int v10;            // static member variable declaration
    inline static int v11;     // inline static member variable definition
    inline static int v12=12;  // inline static member variable definition
};
inline int C::v9=9;  // inline static member variable definition

#endif
```

```
// m.cpp
#include "m.hpp"

int v1;               // non-inline global variable definition
int C::v10;           // non-inline static member variable definition
int & f() { static int v11=0; return v11; }  // static local variable definition
```

Declarations and definitions

- Declarations and definitions of class/struct

```
class B;                                    // class declaration

class A {                                   // class definition
public:
  virtual B * get_B() {                     // virtual member function definition
    return nullptr;                         // declaration of B is sufficient here
  }
  static std::unique_ptr<B> create_B();     // static member function declaration
};

class B : public A {                        // definition of A required here
private:
  virtual B * get_B() override { return this; }
};

inline std::unique_ptr<B> A::create_B() {// static member function definition
  return std::make_unique<B>();             // definition of B required here
}
```

Hello, World!

- t_arg
  - A type alias for "vector of string"
  - using does not create a new type
    - there are also other, unrelated uses of the keyword "using"
- vector is a container
  - Contains objects, not references!
  - string is similar to vector<char>

```
// world.hpp
#ifndef WORLD_HPP_
#define WORLD_HPP_

#include <vector>
#include <string>

// a type-alias definition:
using t_arg = std::vector<std::string>;

// a function declaration:
void world(const t_arg & arg);

#endif
```

```
// main.cpp
#include "world.hpp"

// a function definition:
int main( int argc, char * * argv)
{
  // a local variable definition:
  t_arg arg(argv + 1, argv + argc);
  // statements:
  world(arg);
  return 0;
}
```

- The definition of a variable
  - always creates the corresponding object (not a reference to it)
    - no "new" required/possible here
  - it always calls a constructor
    - constructor arguments may be specified in ()
    - in this case, there are several layers of tricks and implicit casts
  - for local variables, the object is always destroyed when the scope is exited

Hello, World!

- & declares a reference
  - equivalent to passing by reference when used in a function argument
- const
  - prevents accidental modification of the actual argument
  - in this case, passing by reference is done solely for speed
  - const is required in these cases, otherwise the function could not be called with some kinds of arguments

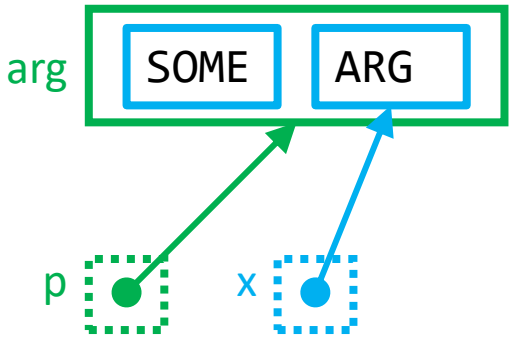
- The for (:) statement
  - iterates through the container p
  - declares the variable x
  - its type is a reference to string
    - "auto" means "let the compiler determine the type"
    - not a "reference to anything" - no run-time flexibility or cost
  - x refers to one of the objects stored inside the container arg
    - without the &&, it would be a copy

```
// world.hpp
#ifndef WORLD_HPP_
#define WORLD_HPP_

#include <vector>
#include <string>

// a type definition:
using t_arg = std::vector<std::string>;

// a function declaration:
void world(const t_arg & p);

#endif
```

```
// world.cpp
#include "world.hpp"
#include <iostream>

// a function definition:
void world( const t_arg & p)
{
    std::cout << "Hello";
    // another local variable:
    for ( auto && x : p )
        std::cout << "," << x;
    std::cout << std::endl;
}
```

Hello, World!

- &, && declare references
  - The variable behaves as if it was the object referred to
  - x refers to one of the objects stored inside the container arg
- Logical point of view
  - vector contains strings
  - string contains chars

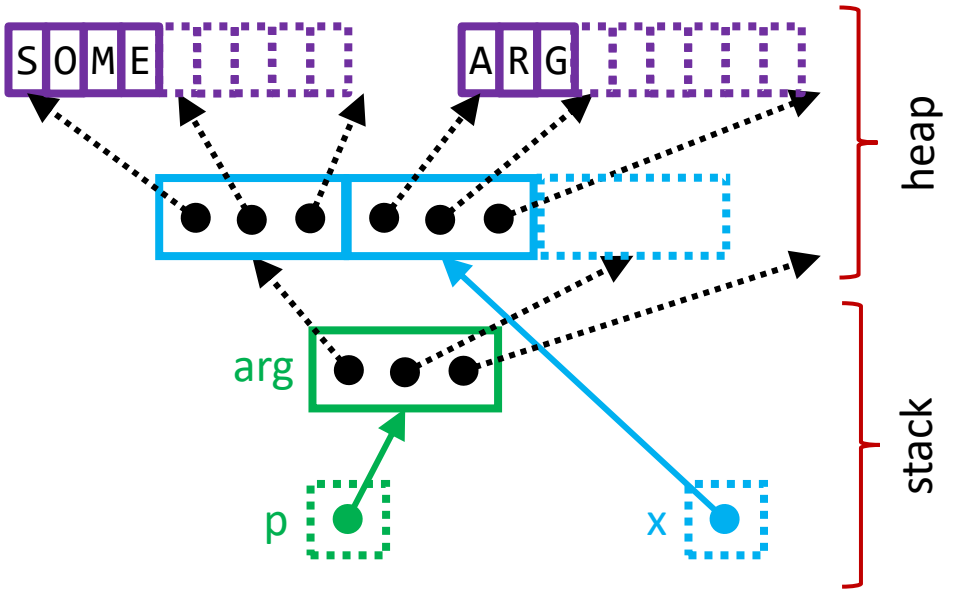



```
// main.cpp
#include "world.hpp"

// a function definition:
int main( int argc, char * * argv)
{
  // a local variable definition:
  t_arg arg(argv + 1, argv + argc);
  // statements:
  world(arg);
  return 0;
}
```

```
// world.cpp
#include "world.hpp"
#include <iostream>

// a function definition:
void world( const t_arg & p)
{
    std::cout << "Hello";
    // another local variable:
    for ( auto && x : p )
        std::cout << "," << x;
    std::cout << std::endl;
}
```

Hello, World!

- &, && declare references
  - The variable behaves as if it was the object referred to
  - x refers to one of the objects stored inside the container arg
- Physical point of view
  - vector/string dynamically allocates place for the strings/chars




```
// main.cpp
#include "world.hpp"

// a function definition:
int main( int argc, char * * argv)
{
  // a local variable definition:
  t_arg arg(argv + 1, argv + argc);
  // statements:
  world(arg);
  return 0;
}
```

```
// world.cpp
#include "world.hpp"
#include <iostream>

// a function definition:
void world( const t_arg & p)
{
    std::cout << "Hello";
    // another local variable:
    for ( auto && x : p )
        std::cout << "," << x;
    std::cout << std::endl;
}
```