UNIVERSITÀ DI PARMA

DIPARTIMENTO DI INGEGNERIA ED ARCHITETTURA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

REALIZZAZIONE DI UN COMPILATORE C++ WEB BROWSER

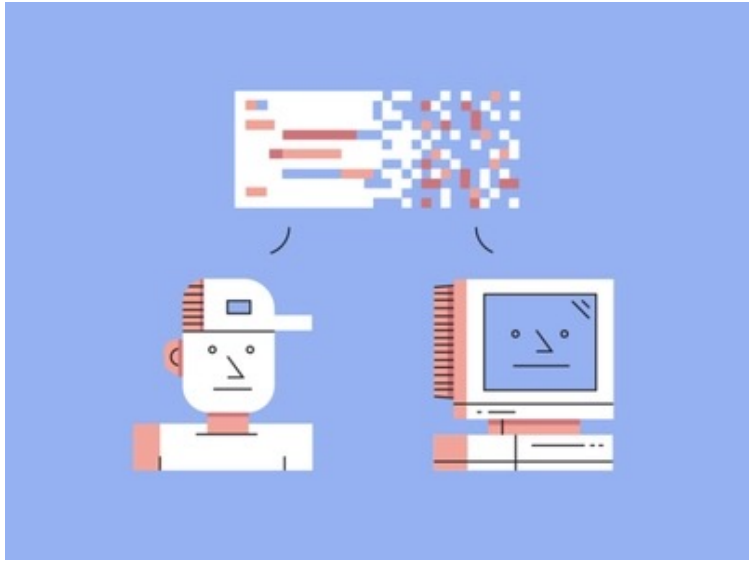
Progetto per il corso ”Paradigmi e linguaggi per analisi di dati” a cura del Professore Michele Tomaiuolo

ANNO ACCADEMICO 2022/2023

Giovanni Schianchi
Giuseppe Ricciardi

INDICE

- CLANG
- WEBASSEMBLY
- EMSCRIPTEN
- WASI
- IL COMPILATORE C++
- DEMO E Q&A



- CLANG
- WEBASSEMBLY
- EMSCRIPTEN
- WASI
- COMPILATORE C++
- DEMO E Q&A

CLANG

INTRODUZIONE

- **Clang** è un compilatore open-source per i linguaggi della famiglia C.
  C, C++, Objective C, CUDA, OPENMP etc. (versione attuale: 15)

- Fa parte del progetto **LLVM**

- *Why? ''*Development of the new front-end was started out of a need for a compiler that allows better diagnostics, better integration with IDEs, a license that is compatible with commercial products, and a nimble compiler that is easy to develop and maintain.''

LLVM

LOW LEVEL VIRTUAL MACHINE*

- Insieme di *compilatori* e *strumenti* che possono essere utilizzati per sviluppare:
  - *front-end* per ogni linguaggio di programmazione
  - back-end per qualsiasi architettura di set d'istruzioni

- Inizio nel 2000, University of Illinois come progetto di ricerca di compilazione dinamica...

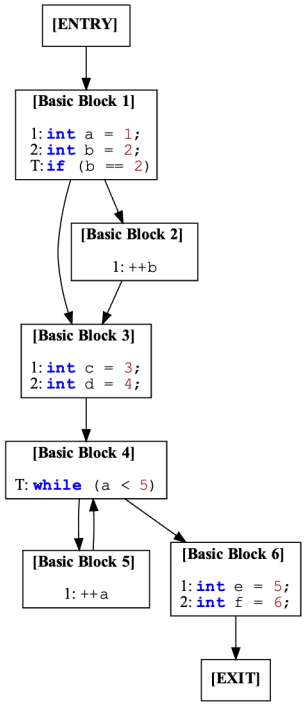
- …oggi, numerosi linguaggi di programmazione sfruttano compilatori LLVM (Haskell, Rust, Fortran, Swift, Ruby etc.)

*LLVM non è più un acronimo dal 2011, il nome indica l'*umbrella project*

LLVM

CARATTERISTICHE PRINCIPALI

- Sfrutta l'**intermediate representation (IR)**, linguaggio di programmazione di basso livello simile ad assembly.
- Garantisce un *formato indipendente* dalla piattaforma.
- L'IR è dotato di *tipizzazione statica*: ogni valore ha un tipo definito e il compilatore controlla la correttezza dei tipi durante la compilazione.
- L'IR rappresenta il codice sorgente come un *grafo dei flussi di controllo.*



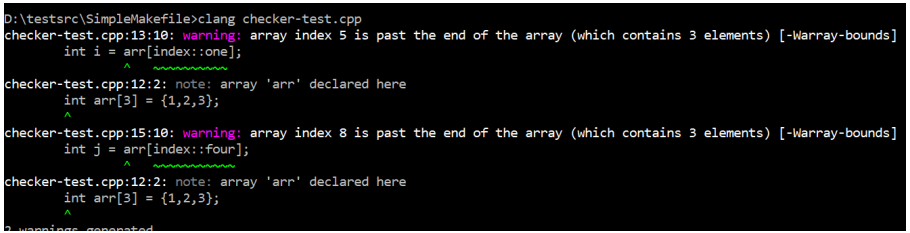
CLANG

DESIGN

- *Library-based architecture*, garantisce al compilatore di collaborare con altri strumenti che interagiscono con il codice sorgente (es. IDE)
  vs. GCC che utilizza il workflow compile-link-debug.

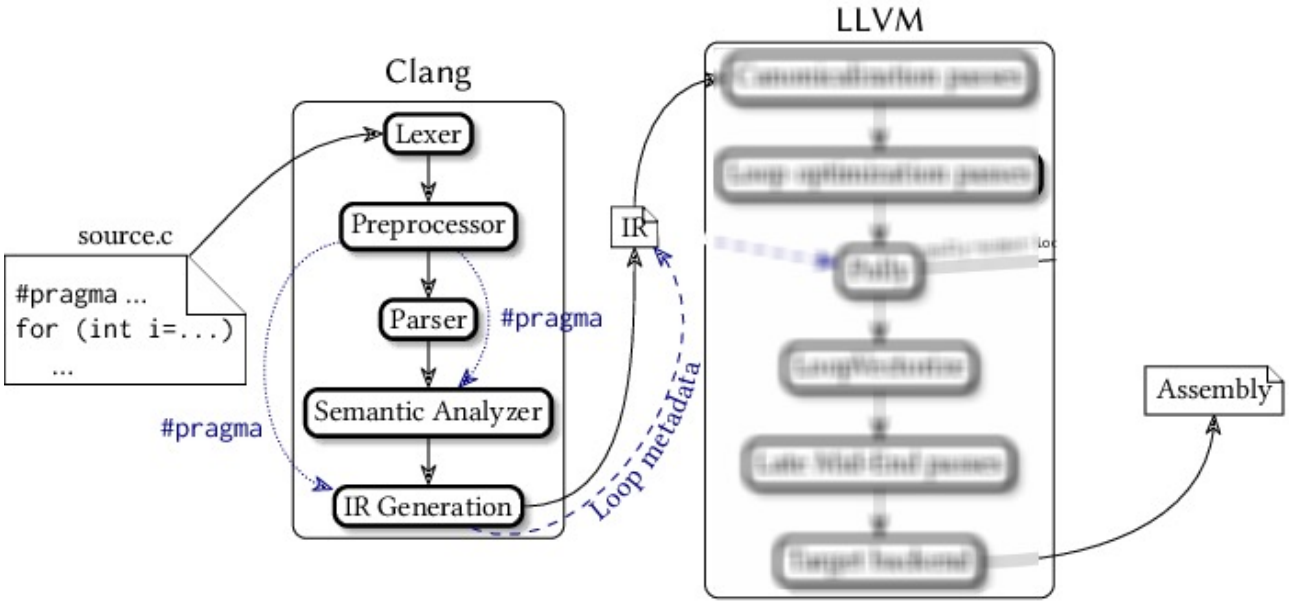
- Clang preserva la forma complessiva del codice originale -> più semplice mappare gli errori!

- Indicizzazione del codice sorgente e maggiore controllo della sintassi grazie all'architettura modulare.



CLANG

ARCHITETTURA



CLANG

ARCHITETTURA

- **Lexer:** suddivide il codice sorgente in token (parole chiavi, identificatori, operatori, costanti, simboli), rivela gli errori lessicali e gestisce le macro.

- **Preprocessore:** si occupa dell'espansione delle macro, include i file, supporta le direttive di precompilazione (''#ifdef'', etc.) e del preprocessore (#pragma). Può essere esteso tramite direttive per personalizzarne il comportamento.

CLANG

LEXER

```c
int factorial(int n) {
  if (n <= 1)
    return 1;
  return n * factorial(n - 1);
}
```

```
> clang -c -Xclang -dump-tokens factorial.c
int                'int'         [StartOfLine]                Loc=<factorial.c:1:1>
identifier         'factorial'   [LeadingSpace]               Loc=<factorial.c:1:5>
l_paren            '('                                        Loc=<factorial.c:1:14>
int                'int'                                      Loc=<factorial.c:1:15>
identifier         'n'           [LeadingSpace]               Loc=<factorial.c:1:19>
r_paren            ')'                                        Loc=<factorial.c:1:20>
l_brace            '{'           [LeadingSpace]               Loc=<factorial.c:1:22>
if                 'if'          [StartOfLine] [LeadingSpace] Loc=<factorial.c:2:3>
l_paren            '('           [LeadingSpace]               Loc=<factorial.c:2:6>
identifier         'n'                                        Loc=<factorial.c:2:7>
lessequal          '<='          [LeadingSpace]               Loc=<factorial.c:2:9>
numeric_constant   '1'           [LeadingSpace]               Loc=<factorial.c:2:12>
r_paren            ')'                                        Loc=<factorial.c:2:13>
...
```

CLANG

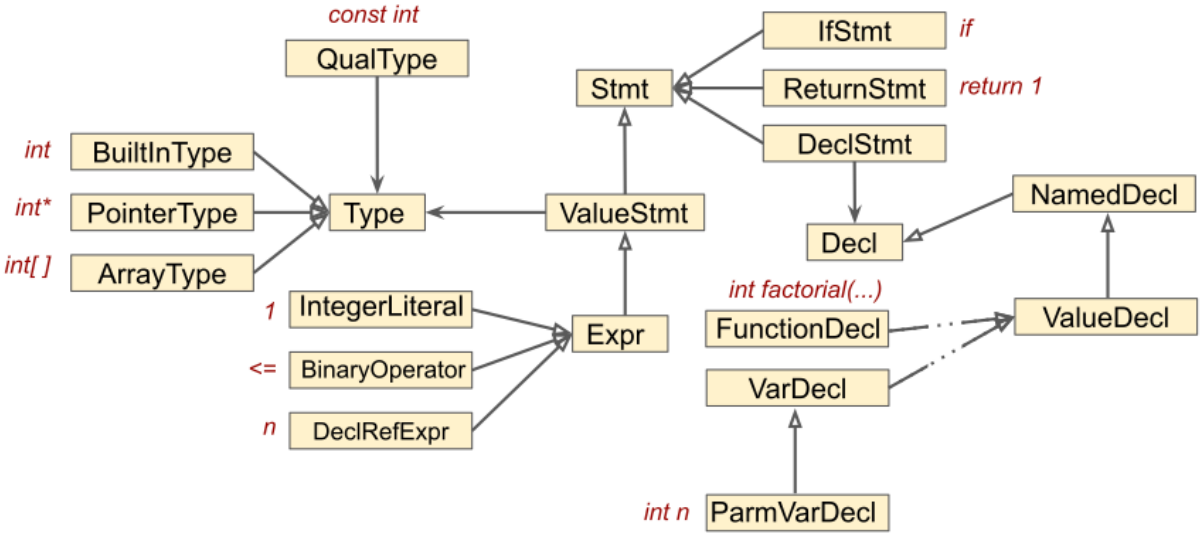
ARCHITETTURA

- **Parser:** converte il flusso di token generato dal lexer in una struttura ad albero chiamata *Abstract Syntax Tree (*AST*).* Esegue l'*analisi sintattica*, ovvero associa gli identificatori alle relative dichiarazioni nel contesto del programma (= gestione dello scope delle variabili).

- **Semantic Analyzer:** è il responsabile della valutazione e dell'applicazione delle regole semantiche del linguaggio di programmazione nel codice sorgente. Verifica la correttezza dei tipi, controlla e rileva eventuali errori semantici, raccoglie informazioni aggiuntive sul codice che possono essere utilizzate nelle fasi successive.

CLANG

AST


const int
QualType
int
BuiltInType
int*
PointerType
int[ ]
ArrayType
Type
Stmt
IfStmt
if
ReturnStmt
return 1
DeclStmt
ValueStmt
Decl
NamedDecl
1
IntegerLiteral
<=
BinaryOperator
n
DeclRefExpr
Expr
int factorial(...)
FunctionDecl
ValueDecl
VarDecl
int n
ParmVarDecl

CLANG

UN CASO D'USO RECENTE

Clang is now used to build Chrome for Windows

By Nico Weber
Mar 5, 2018
#C++ , *#Clang* , *#Products*
10 minute read

As of Chrome 64, Chrome for Windows is compiled with Clang. We now use Clang to build Chrome for all platforms it runs on: macOS, iOS, Linux, Chrome OS, Android, and Windows. Windows is the platform with the second most Chrome users after Android according to statcounter, which made this switch particularly exciting.

CLANG

PRO

- ✅ Modularità ed architettura ben progettata
- ✅ Buone prestazioni
- ✅ Messaggi di errore chiari e accurati
- ✅ Supporto multi-piattaforma

CONTRO

- ❌ Consumo di memoria elevato,
- ❌ Supporto per alcune piattaforme o architetture limitato
- ❌ Documentazione limitata

- CLANG
- **WEBASSEMBLY**
- EMSCRIPTEN
- WASI
- COMPILATORE C++
- DEMO E Q&A

WEBASSEMBLY

- WebAssembly è uno standard web open che definisce un formato di istruzioni binarie per la esecuzione di codice all'interno di pagine web
- Creato per estendere le funzionalità e risolvere i limiti di Javascript

WEBASSEMBLY

Confronto con JavaScript

WebAssembly vs JavaScript critical differences	
WA WebAssembly	JS JavaScript
• **Wasm** uses **small binary files** and ahead-of-time processing, so it works **faster than Javascript.**	• **Javascript's** long **history** of use means that **more devices and browsers support it.**
• **Wasm** executes in a sandbox, giving users **better security than Javascript.**	• **JS** works **quickly for small tasks,** Wasm performs faster for larger, more complex applications.
• **Wasm** uses a compiler, which **improves debugging before deployment.**	• **JS** garbage collection **handles memory usage** while Wasm requires manual management.

WEBASSEMBLY

Caratteristiche

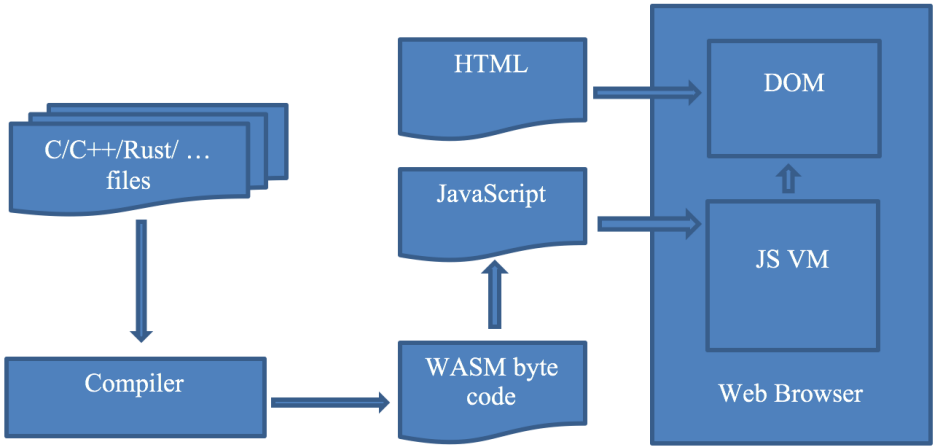
WebAssembly è stato progettato seguendo questi principi :

- Portabile
- Estendibile
- Integrato con gli standard Web
- Basato su tool esistenti

- Wasm è composto da moduli ognuno dei quali è formato da diverse sezioni
- La memoria è lineare e quindi l'indirizzamento è efficace

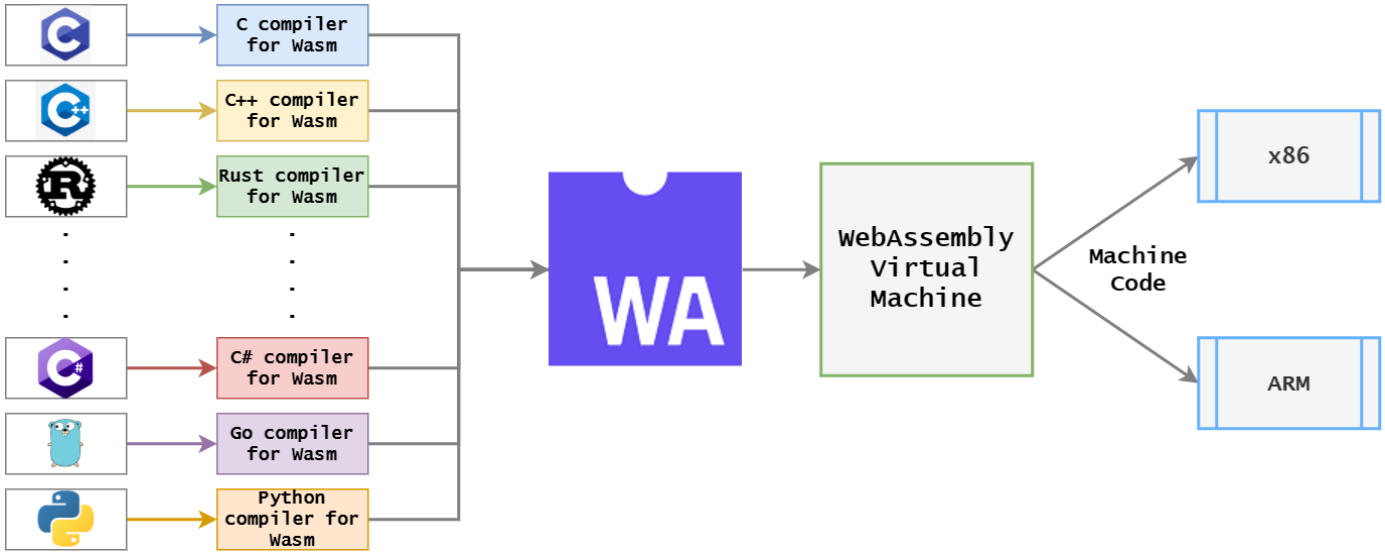
WEBASSEMBLY

Funzionamento



- Il codice viene compilato da WASM che genera del byte code
- Viene avviata una VM su cui gira l'applicazione

WEBASSEMBLY

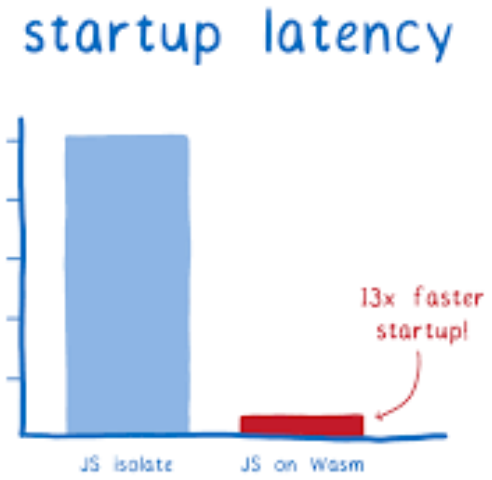


WEBASSEMBLY

Considerazioni

- In conclusione possiamo dire che WASM sarà la tecnologia che prenderà sopravvento nel mondo web

- Anche se mancano ancora alcune funzionalità, di cui alcune verranno aggiunte:
- Supporto al Garbage Collector
- Supporto al multithread
- Gestione delle eccezioni



- CLANG
- WEBASSEMBLY
- **EMSCRIPTEN**
- WASI
- COMPILATORE C++
- DEMO E Q&A

EMSCRIPTEN

INTRODUZIONE

- Strumento open-source che consente la compilazione di codice sorgente in linguaggi come C, C++, Rust, etc. in WebAssembly.
- Sfrutta il LLVRM IR per tradurre il codice sorgente, e lo riutilizza per convertirlo in WebAssembly.
- Fornisce anche un'API JavaScript che permette al codice compilato di interagire con l'ambiente web circostante

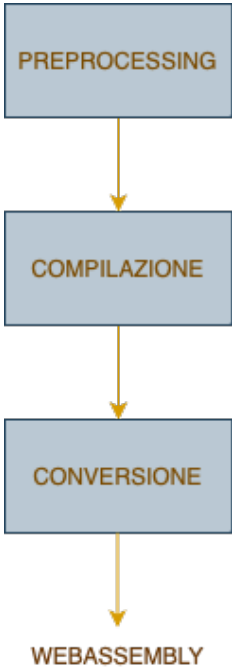
EMSCRIPTEN

WORKFLOW

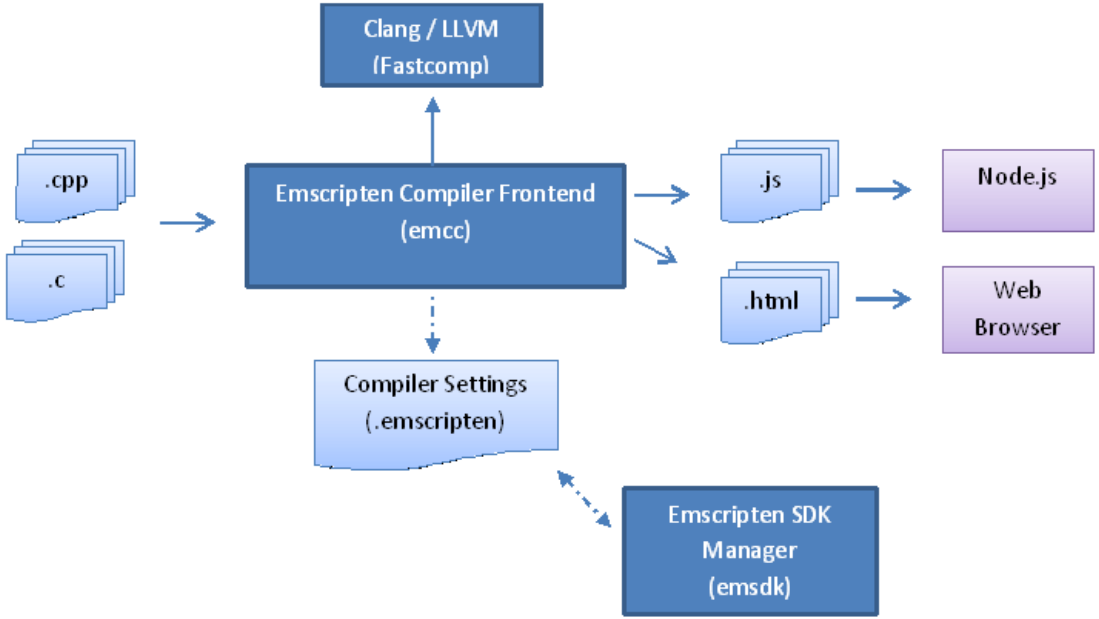
1. *Preprocessing: v*engono applicate le direttive di inclusione, macro e altre trasformazioni necessarie per preparare il codice per la compilazione successiva.

2. *Compilazione*: Emscripten utilizza il frontend di LLVM per compilare il codice C/C++ in LLVM IR. Durante questa fase, vengono applicate ottimizzazioni al codice per migliorare le prestazioni

3. *Conversione in WebAssembly*: l'LLVM IR generato viene quindi convertito in WebAssembly utilizzando il backend di Emscripten.



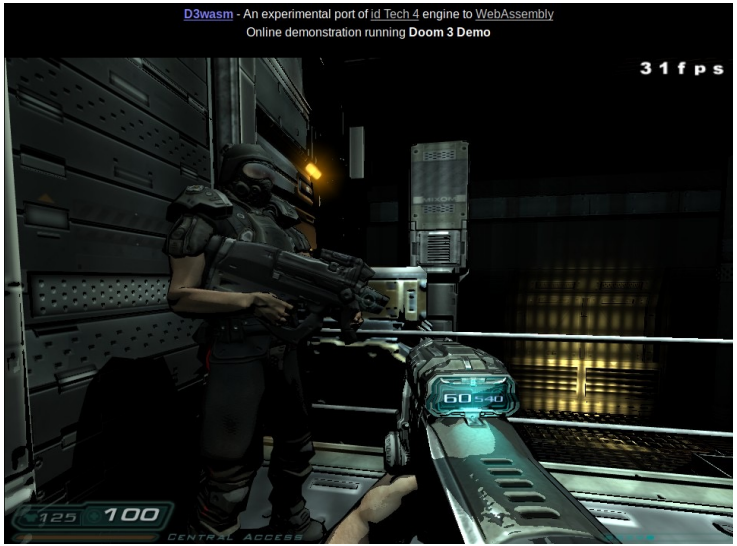
EMSCRIPTEN


Clang / LLVM
(Fastcomp)
.cpp
.c
Emscripten Compiler Frontend
(emcc)
.js
Node.js
.html
Web
Browser
Compiler Settings
(.emscripten)
Emscripten SDK
Manager
(emsdk)

EMSCRIPTEN

ALCUNI ESEMPI

- Porting su webAssembly del noto videogame ''Doom3'': http://wasm.continuation-labs.com/d3demo/
- Porting di Unreal Engine 3, Unreal Engine 4, Unity e molti altri motori grafici.
- Mozilla usa Emscripten per portare diverse librerie e applicazioni all'interno di Firefox. Ad esempio, il progetto "pdf.js" utilizza Emscripten per eseguire l'elaborazione dei file PDF direttamente nel browser.



EMSCRIPTEN

BROWSERFS

- Libreria JavaScript che consente di creare un sistema di file virtuali all'interno di un browser web.
- Emula l'API del file system Node JS e supporta l'archiviazione e il recupero di file da vari back-end.
- CodeSandbox, HyperDev, Theia IDE e molti altri…

```
"backend/AsyncMirror"
"backend/Dropbox"
"backend/Emscripten"
"backend/FolderAdapter"
"backend/HTML5FS"
"backend/InMemory"
"backend/IndexedDB"
"backend/IsoFS"
"backend/LocalStorage"
"backend/MountableFileSystem"
"backend/OverlayFS"
"backend/WorkerFS"
"backend/XmlHttpRequest"
"backend/ZipFS"
```

EMSCRIPTEN

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "r");
    int count = 0;
    char ch;

    if (file == NULL) {
        printf("Impossibile aprire il file.");
        return 1;
    }

    while ((ch = fgetc(file)) != EOF) {
        if (ch == '\n') {
            count++;
        }
    }

    fclose(file);

    printf("Numero di righe nel file: %d\n", count);

    return 0;
}
```

```
emcc count_lines.c -o count_lines.js
```

```html
<!DOCTYPE html>
<html>
<head>
    <title>Esempio di BrowserFS ed Emscripten</title>
    <script src="browserfs.min.js"></script>
    <script src="count_lines.js"></script>
</head>
<body>
    <script>
        BrowserFS.configure({
            fs: "IndexedDB",
            options: {}
        }, function(err) {
            if (err) {
                console.log(err);
                return;
            }

            FS.mkdir('/work');
            FS.mount(BFS.IndexedDB, {}, '/work');
            FS.chdir('/work');

            Module.callMain();
        });
    </script>
</body>
</html>
```

- CLANG
- WEBASSEMBLY
- EMSCRIPTEN
- **WASI**
- COMPILATORE C++
- DEMO E Q&A

WASI

- WebAssembly System Interface
- Nato per standardizzare il modo in cui WebAssembly interagisce con il S.O.
- Con WASI è possibile eseguire applicazioni WebAssembly su ogni piattaforma

WASI

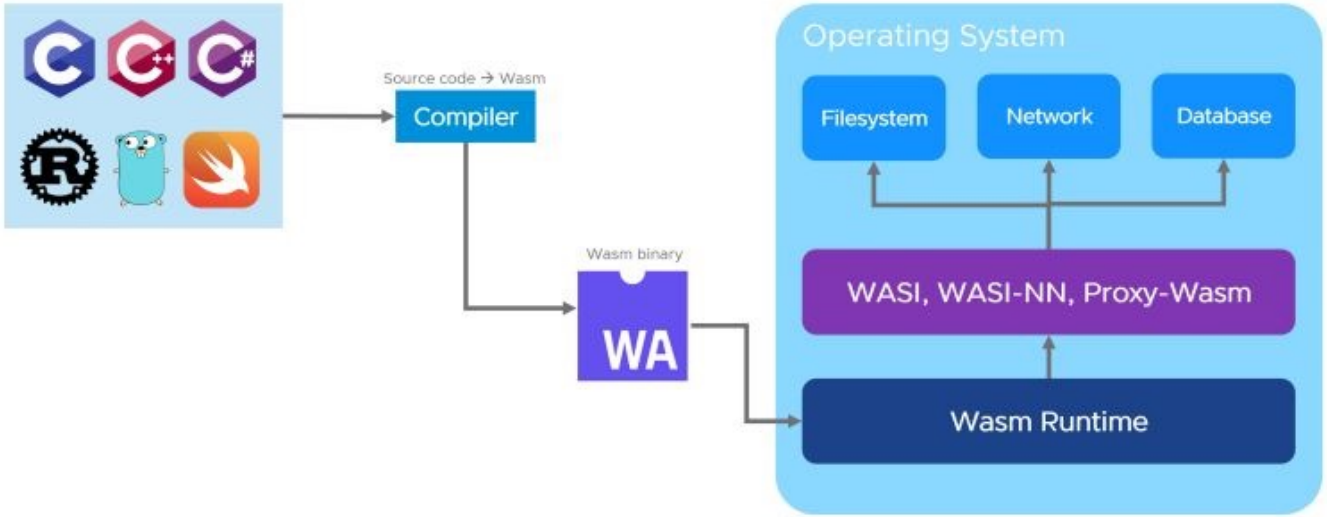
Funzionalità

Offre un set standardizzato di API per i moduli WebAssembly per accedere a risorse di sistema:

- FileSystem
- Networking
- Time
- Random
- Database

WASI

Architettura

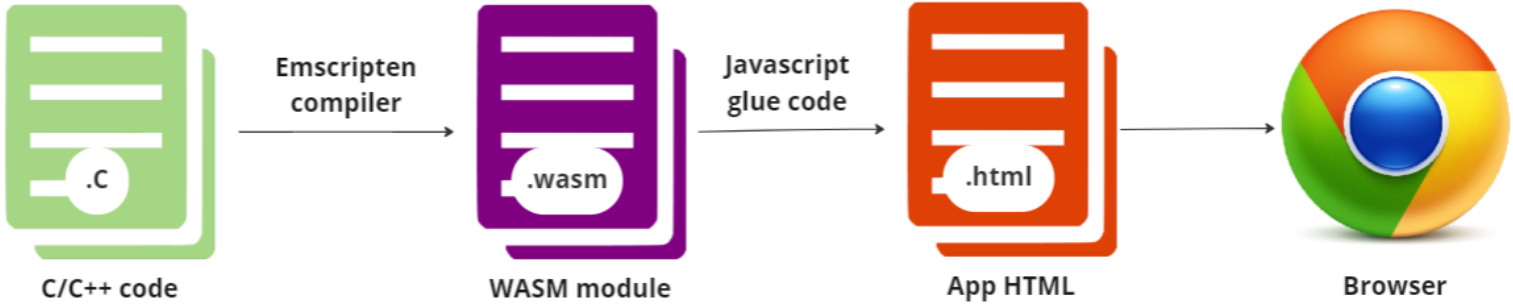


WASI

WASI con Emscripten

- WASI è utilizzato in modo complementare ad Emscripten.
- Consente alle applicazioni in C/C++ di interagire con l'ambiente WebAssembly

WASI


.C
Emscripten compiler
.wasm
Javascript glue code
.html
C/C++ code
WASM module
App HTML
Browser

- CLANG
- WEBASSEMBLY
- EMSCRIPTEN
- WASI
- **COMPILATORE C++**
- DEMO E Q&A

COMPILATORE C++

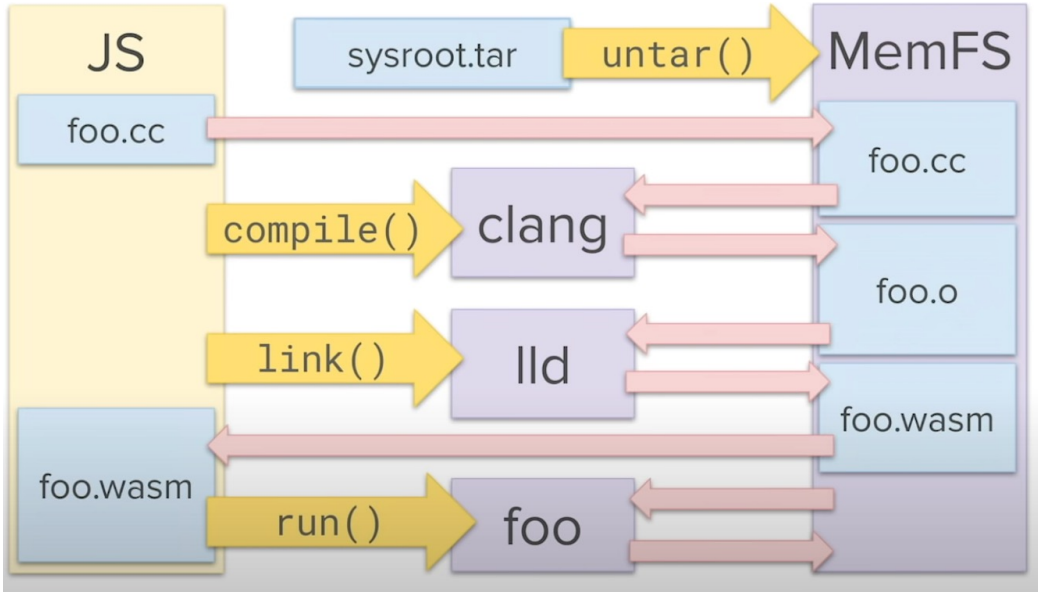
OBIETTIVI

- http://www.ce.unipr.it/cib/
  problemi con iostream
- https://github.com/binji/wasm-clang
  nuovo punto d'inizio
- Implementare la libreria g2d.h

  …facile, no?

COMPILATORE C++

ARCHITETTURA

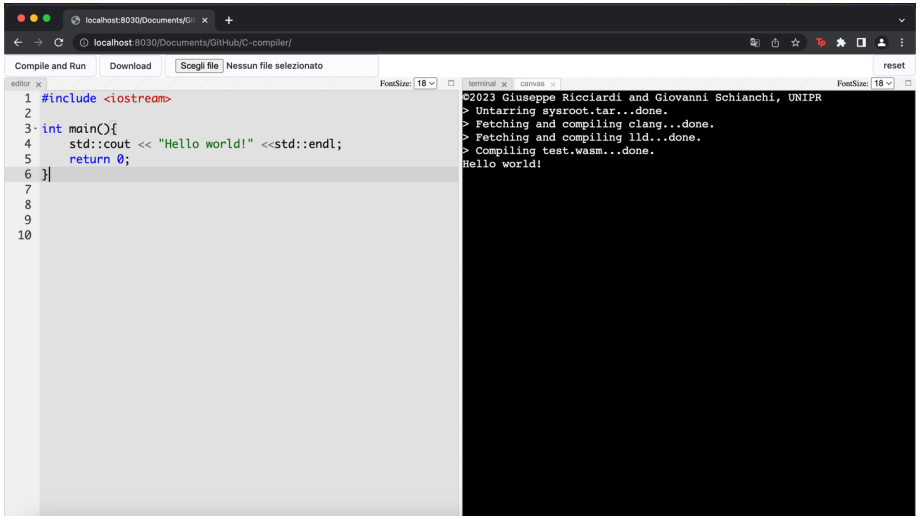


COMPILATORE C++

FRONT-END

- Utilizzo di *Golden Layout*, libreria JS che fornisce un framework per creare layout multi-finestra nelle applicazioni web.

- Ogni Layout è composto da *componenti*: l'*editor* in cui l'utente scrive il proprio codice, il *terminale* che mostra i risultati dell'esecuzione ed il *canvas* per mostrare i risultati di g2d.



COMPILATORE C++

BACK-END

- La classe **Mem**, definisce metodi per leggere e scrivere dati nella memoria del modulo WebAssembly.
- *La classe* **MemFS**, si occupa dei file system in memoria.
- La classe **App** implementa i metodi per eseguire l'applicazione e le funzionalità di interazione con il modulo WebAssembly. (gestione del canvas, l'input/output, le chiamate di sistema WASI etc.)
- La classe **Api** fornisce classi e funzioni di supporto. (classi di errore, ''getInstance'' per ottenere un'istanza di un modulo WebAssembly, etc.)

shared.js

COMPILATORE C++

BACK-END

```
async run(module, ...args) {
  //console.log("Ho iniziato run")
  this.hostLog(`${args.join(' ')}\n`);
  const start = +new Date();
  const app = new App(module, this.memfs, ...args); //crea oggetto App
  const instantiate = +new Date();
  const stillRunning = await app.run(); //chiama run di App
  const end = +new Date();
  this.hostWrite('\n');
  if (this.showTiming) {
    const green = '\x1b[92m';
    const normal = '\x1b[0m';
    let msg = `${green}(${msToSec(start, instantiate)}s`;
    msg += `/${msToSec(instantiate, end)}s)${normal}\n`;
    this.hostWrite(msg);
  }
  return stillRunning ? app : null; //ritorna il risultato del metodo run della classe app
}
```

classe Api

classe App

```
async run() {
  await this.ready;
  //console.log("run di App: " + await this.ready)
  try {
    this.exports._start();
  } catch (exn) {
    let writeStack = true;
    if (exn instanceof ProcExit) {
      if (exn.code === RAF_PROC_EXIT_CODE) {
        console.log('Allowing rAF after exit.');
        return true;
      }
      // Don't allow rAF unless you return the right code.
      console.log(`Disallowing rAF since exit code is ${exn.code}.`);
      this.allowRequestAnimationFrame = false;
      if (exn.code == 0) {
        return false;
      }
      writeStack = false;
    }

    // Write error message.
    let msg = `\x1b[91mError: ${exn.message}`;
    if (writeStack) {
      msg = msg + `\n${exn.stack}`;
    }
    msg += '\x1b[0m\n';
    this.memfs.hostWrite(msg);
    term.write(msg);
    // Propagate error.
    throw exn;
  }
}
```

COMPILATORE C++

BACK-END

```
async compile(options) {
  const input = options.input;
  const contents = options.contents;
  const obj = options.obj;
  const opt = options.opt || '2';

  await this.ready;
  this.memfs.addFile(input, contents);
  const clang = await this.getModule(this.clangFilename);
  return await this.run(clang, 'clang', '-cc1', '-emit-obj',
                        ...this.clangCommonArgs, '-O2', '-o', obj, '-x',
                        'c++', input);
}
```

```
async link(obj, wasm) {
  const stackSize = 1024 * 1024;
  const libdir = 'lib/wasm32-wasi';
  const crt1 = `${libdir}/crt1.o`;
  await this.ready;
  const lld = await this.getModule(this.lldFilename);
  //console.log(this.lldFilename);
  return await this.run(
      lld, 'wasm-ld', '--no-threads',
      '--export-dynamic',  // TODO required?
      '-z', `stack-size=${stackSize}`, `-L${libdir}`, crt1, obj, '-lc',
      '-lc++', '-lc++abi', '-lcanvas','-lg2d' ,'-o', wasm)
}
```

```
async getModule(name) {
  if (this.moduleCache[name]) return this.moduleCache[name];
  const module = await this.hostLogAsync(`Fetching and compiling ${name}`,
                                         this.compileStreaming(name));
  this.moduleCache[name] = module;
  return module;
}
```

COMPILATORE C++

COMPILE

- Compila il codice sorgente C++ in un file oggetto (obj). Aspetta prima che il modulo Clang sia pronto per essere utilizzato (*this.ready*) e quindi aggiunge il file di input a in-memory file system (*this.memfs.addFile*). Successivamente, viene ottenuto il modulo Clang e viene eseguito utilizzando il metodo *run*.

```
async compile(options) {
  const input = options.input;
  const contents = options.contents;
  const obj = options.obj;
  const opt = options.opt || '2';

  await this.ready;
  this.memfs.addFile(input, contents);
  const clang = await this.getModule(this.clangFilename);
  return await this.run(clang, 'clang', '-cc1', '-emit-obj',
                        ...this.clangCommonArgs, '-O2', '-o', obj, '-x',
                        'c++', input);
}
```

COMPILATORE C++

LINK

- Collega il file oggetto (*obj*) in un modulo WebAssembly utilizzando il *linker* LLD. Prende in input il percorso del file oggetto e il percorso del file di output del modulo wasm. La funzione aspetta che venga caricato il modulo LLD. Successivamente, viene eseguito il linker LLD utilizzando il metodo run.

```
async link(obj, wasm) {
  const stackSize = 1024 * 1024;
  const libdir = 'lib/wasm32-wasi';
  const crt1 = `${libdir}/crt1.o`;
  await this.ready;
  const lld = await this.getModule(this.lldFilename);
  //console.log(this.lldFilename);
  return await this.run(
      lld, 'wasm-ld', '--no-threads',
      '--export-dynamic',  // TODO required?
      '-z', `stack-size=${stackSize}`, `-L${libdir}`, crt1, obj, '-lc',
      '-lc++', '-lc++abi', '-lcanvas','-lg2d' ,'-o', wasm)
}
```

COMPILATORE C++

G2D

- L’implementazione si basa su:
  - aggiunta della libreria negli in-memory files system
  - gestione delle chiamate ai metodi della libreria in WebAssembly

- Per l’implementazione in memoria è stato aggiunto un file ’’canvas.h’’ per la definizione dei metodi della libreria -> C++

- La classe App si occupa della gestione dei metodi della libreria (disegno su canvas, caricamento delle immagini, prompt, dialog, etc.) -> Javascript

COMPILATORE C++

G2D

sysroot/canvas.h

```
extern "C"{
    void js_load_element(const char * tag, const char * src);
    void js_fill_rect(double x, double y, double width, double height);
    void js_fill_circle(double x, double y, double r);
```

```
void fill_rect(Point p, Point s) { js_fill_rect(p.x, p.y, s.x, s.y); }
void fill_circle(Point p, int r) { js_fill_circle(p.x, p.y, r); }
std::string load_image(std::string src) { js_load_element("IMG", src.c_str()); return src; }
```

shared.js

```
js_fill_circle(x, y, r) {
    ctx2d.beginPath();
    ctx2d.arc(x, y, r, 0, 2*Math.PI);
    ctx2d.closePath();
    ctx2d.fill();
}
js_fill_rect(...args){if (ctx2d) ctx2d.fillRect(...args)}
js_load_element(tag, src) { loadElement(this.mem.readStr(tag), this.mem.readStr(src)); }
```

EMSCRIPTEN

RISORSE UTILI

- Sito web LLVM:
  https://llvm.org
- Sito web Clang:
  https://clang.llvm.org
- Sito WebAssemby:
  https://webassembly.org
- Sito web di emscripten:
  https://emscripten.org/index.html
- Sito web WASI:
  https://wasi.dev

DEMO TIME!

Puoi provarlo qui

COMPILATORE C++

WHAT'S MISSING?

- Input con <iostream> (cin>>>, scanf…).
- Lettura e scrittura da file (parzialmente implementata).
- Implementazione del timer


Q
&
A

GRAZIE PER L'ATTENZIONE