Åbo Akademi

Parallel Computing

IT00CD91-3005, Spring 2025

Lecture 2: C/C++ Refresher

by Alireza Olama

C/C++

Compiled Language

Both C and C++ syntax is used in parallel computing.

CUDA programming syntax is like C

Requires **manual memory management** (mostly C)

C++ has direct support for **Object Oriented Programming (OOP)** but in this course we don't use OOP too much

C++ features that we use: Exception handling, `std::vector`, `std::thread`, smart pointers, etc.

cppreference.com Create account Search

Page Discussion Standard revision: Diff View View source History

Planned Maintenance
The site will be in a temporary read-only mode in the next few weeks to facilitate some long-overdue software updates. We apologize for any inconvenience this may cause!

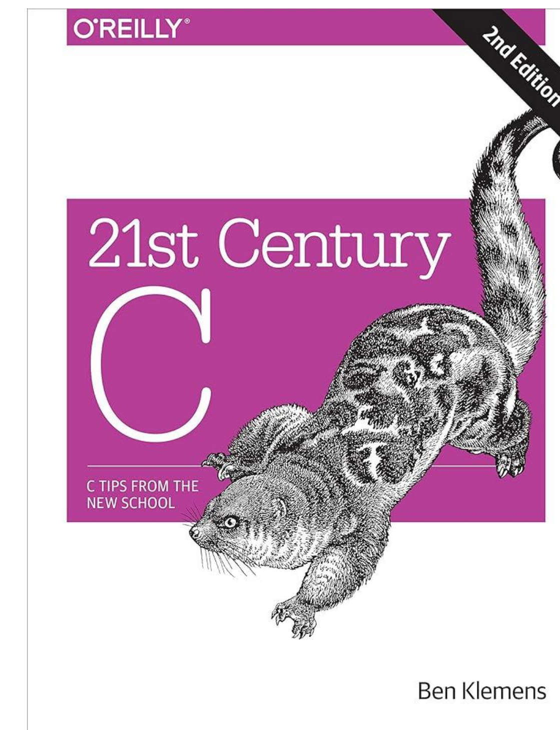
C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

Language
Preprocessor – Comments
ASCII chart
Basic concepts
Keywords
Names (lookup)
Types (fundamental types)
The main function
Modules (C++20)
Contracts (C++26)
Expressions
Value categories
Evaluation order
Operators (precedence)
Conversions – Literals
Constant expressions
Statements
if – switch
for – range-for (C++11)
while – do-while
Declarations – Initialization
Functions – Overloading
Coroutines (C++20)
Classes (unions)
Templates – Exceptions
Freestanding implementations
Standard library (headers)

Diagnostics library
Assertions – System error (C++11)
Exception types – Error numbers
basic_stacktrace (C++23)
Debugging support (C++26)
Memory management library
Allocators – Smart pointers
Memory resources (C++17)
Metaprogramming library (C++11)
Type traits – ratio
integer_sequence (C++14)
General utilities library
Function objects – hash (C++11)
Swap – Type operations (C++11)
Integer comparison (C++20)
pair – tuple (C++11)
optional (C++17)
expected (C++23)
variant (C++17) – any (C++17)
bitset – Bit manipulation (C++20)
Containers library
vector – deque – array (C++11)
list – forward_list (C++11)
inplace_vector (C++26)
hive (C++26)
map – multimap – set – multiset

Strings library
basic_string – char_traits
basic_string_view (C++17)
Text processing library
Primitive numeric conversions (C++17)
Formatting (C++20) – Localization
text_encoding (C++26)
Regular expressions (C++11)
basic_regex – Algorithms
Default regular expression grammar
Null-terminated sequence utilities:
byte – multibyte – wide
Numerics library
Common math functions
Mathematical special functions (C++17)
Mathematical constants (C++20)
Basic linear algebra algorithms (C++26)
Data-parallel types (SIMD) (C++26)
Pseudo-random number generation
Floating-point environment (C++11)
complex – valarray
Date and time library
Calendar (C++20) – Time zone (C++20)
Input/output library
Print functions (C++23)
Stream-based I/O – I/O manipulators



Common Bugs

Segmentation faults

Memory leaks

Use-after-free

Double Free

Race Conditions (multi-threaded programs)

Deadlocks (multi-threaded programs)

Integer overflows

Tools

Compiler

`brew install gcc` on MAC

`sudo apt install build-essential` On Linux

MSVC, Mingw, and **WSL** on Windows

How to install Linux on Windows with WSL

Article • 11/20/2024 • 10 contributors

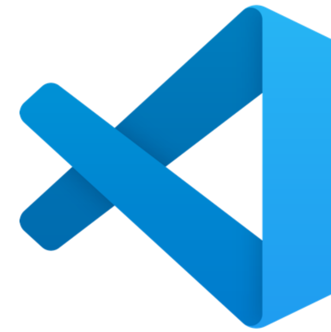
Feedback

In this article

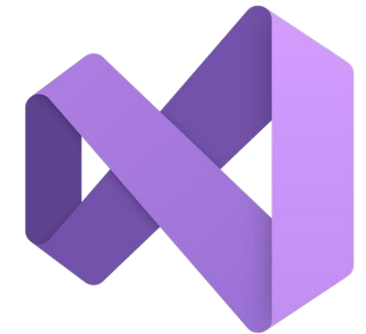
- Prerequisites
- Install WSL command
- Change the default Linux distribution installed
- Set up your Linux user info

Show 6 more

Developers can access the power of both Windows and Linux at the same time on a Windows machine. The Windows Subsystem for Linux (WSL) lets developers install a Linux distribution (such as Ubuntu, OpenSUSE, Kali, Debian, Arch Linux, etc) and use Linux applications, utilities, and Bash command-line tools directly on Windows, unmodified, without the overhead of a traditional virtual machine or dualboot setup.



Visual Studio Code (Vscode)



Visual Studio

CLion

https://learn.microsoft.com/en-us/windows/wsl/install

Try to setup dev tools locally if possible

Hello World & Compilation Basics

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, Parallel World!\n");
    return 0;
}
```

```
gcc hello.c -o hello
```

Compilation on Linux & Mac

Hello World & Compilation Basics

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, Parallel World!\n");
    return 0;
}
```

hello.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, Parallel World!" << std::endl;
    return 0;
}
```

```
gcc hello.c -o hello
```

```
g++ hello.cpp -o hello
```

Compilation on Linux & Mac

Hello World & Compilation Basics

hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, Parallel World!\n");
    return 0;
}
```

```
gcc hello.c -o hello
```

--save-temps

hello.cpp

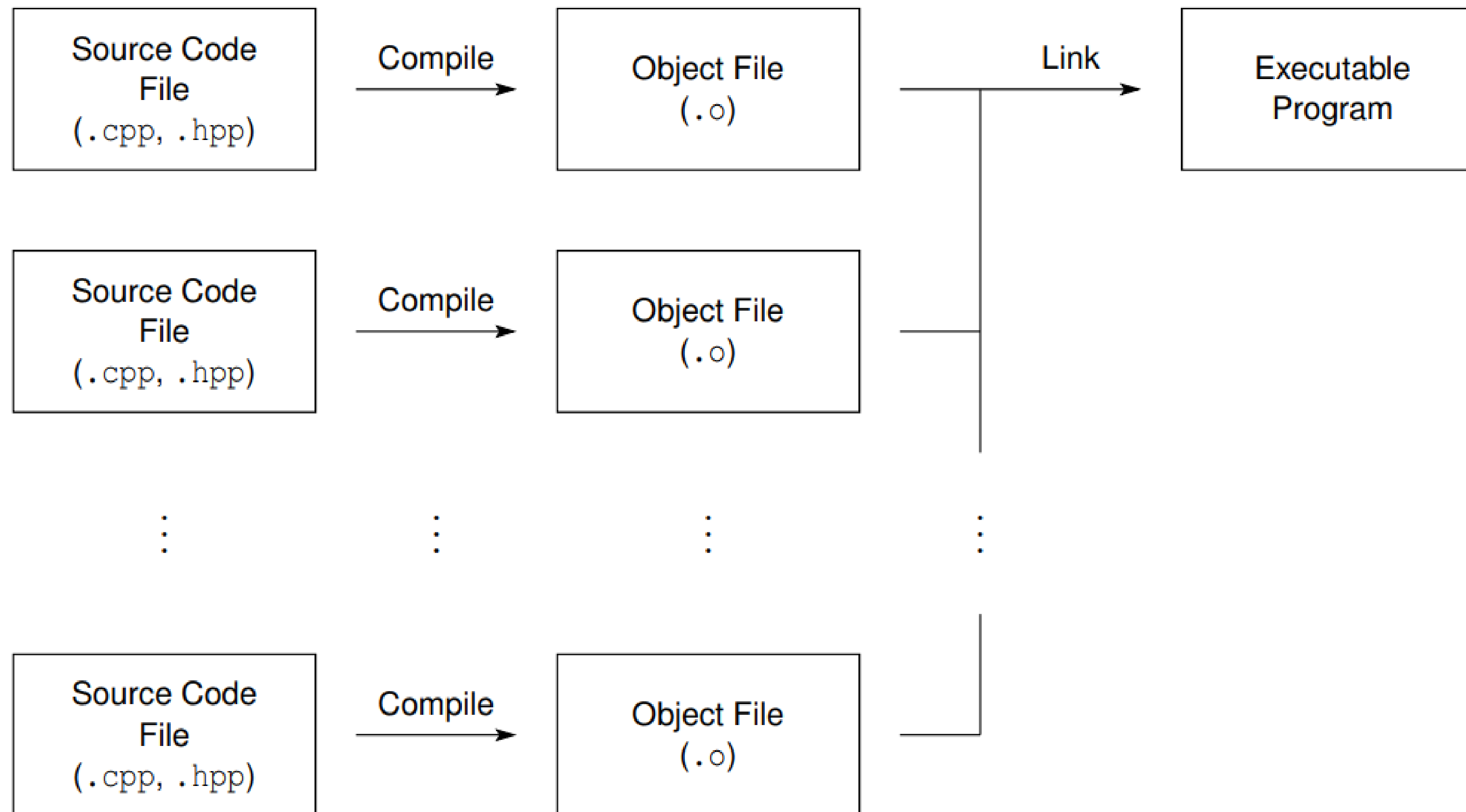
```cpp
#include <iostream>

int main() {
    std::cout << "Hello, Parallel World!" << std::endl;
    return 0;
}
```

```
g++ hello.cpp -o hello
```

--save-temps

Compilation on Linux & Mac

Compilation Basics



Disassemble the object file and show the assembly code

```
Objdump -d object_file.o
```

Preprocessor

The **preprocessor** processes a source file before compilation, it allows to define macros which are abbreviations for longer structs.

Lines which begin with #

Preprocessor

The **preprocessor** processes a source file before compilation, it allows to define macros which are abbreviations for longer structs.

Lines which begin with #

```
#define PI 3.14159        // Defines a constant value
#define SQUARE(x) ((x)*(x))  // Defines a function-like macro

// Usage
int area = PI * SQUARE(5);  // Expands to: 3.14159 * ((5)*(5))
```

Preprocessor

The **preprocessor** processes a source file before compilation, it allows to define macros which are abbreviations for longer structs.

Lines which begin with #

```c
#define PI 3.14159       // Defines a constant value
#define SQUARE(x) ((x)*(x))   // Defines a function-like macro

// Usage
int area = PI * SQUARE(5);   // Expands to: 3.14159 * ((5)*(5))
```

```c
#ifndef CONFIG_H
#define CONFIG_H

// Header file content here
// This prevents the file from being included multiple times

#endif  // CONFIG_H
```

Preprocessor

The **preprocessor** processes a source file before compilation, it allows to define macros which are abbreviations for longer structs.

Lines which begin with #

```c
#define PI 3.14159       // Defines a constant value
#define SQUARE(x) ((x)*(x))  // Defines a function-like macro

// Usage
int area = PI * SQUARE(5);  // Expands to: 3.14159 * ((5)*(5))
```

```c
// 4. #pragma - Compiler-specific instruction
#pragma once  // Ensures the file is included only once
```

```c
#ifndef CONFIG_H
#define CONFIG_H

// Header file content here
// This prevents the file from being included multiple times

#endif  // CONFIG_H
```

Preprocessor

The **preprocessor** processes a source file before compilation, it allows to define macros which are abbreviations for longer structs.

Lines which begin with #

```
#define PI 3.14159       // Defines a constant value
#define SQUARE(x) ((x)*(x))  // Defines a function-like macro

// Usage
int area = PI * SQUARE(5);  // Expands to: 3.14159 * ((5)*(5))
```

```
#ifndef CONFIG_H
#define CONFIG_H

// Header file content here
// This prevents the file from being included multiple times

#endif  // CONFIG_H
```

```
// 4. #pragma – Compiler-specific instruction
#pragma once  // Ensures the file is included only once
```

```
#include <stdio.h>     // Standard library
#include "myheader.h" // Local user-defined header file
```

Preprocessor

The **preprocessor** processes a source file before compilation, it allows to define macros which are abbreviations for longer structs.

Lines which begin with #

```c
#define PI 3.14159       // Defines a constant value
#define SQUARE(x) ((x)*(x))   // Defines a function-like macro

// Usage
int area = PI * SQUARE(5);   // Expands to: 3.14159 * ((5)*(5))
```

```c
#ifndef CONFIG_H
#define CONFIG_H

// Header file content here
// This prevents the file from being included multiple times

#endif  // CONFIG_H
```

```c
// 4. #pragma – Compiler-specific instruction
#pragma once  // Ensures the file is included only once
```

```c
#include <stdio.h>     // Standard library
#include "myheader.h" // Local user-defined header file
```

```c
#define DEBUG

#ifdef DEBUG
    printf("Debug mode is on\n");
#endif
```

Preprocessor

The **preprocessor** processes a source file before compilation, it allows to define macros which are abbreviations for longer structs.

Lines which begin with #

```c
#define PI 3.14159        // Defines a constant value
#define SQUARE(x) ((x)*(x))   // Defines a function-like macro

// Usage
int area = PI * SQUARE(5);  // Expands to: 3.14159 * ((5)*(5))
```

```c
// 4. #pragma — Compiler-specific instruction
#pragma once   // Ensures the file is included only once
```

```c
#include <stdio.h>     // Standard library
#include "myheader.h"  // Local user-defined header file
```

```c
#ifndef CONFIG_H
#define CONFIG_H

// Header file content here
// This prevents the file from being included multiple times

#endif  // CONFIG_H
```

```c
#define DEBUG

#ifdef DEBUG
    printf("Debug mode is on\n");
#endif
```

```c
#define VERSION 2

#if VERSION == 1
    printf("Version 1\n");
#elif VERSION == 2
    printf("Version 2\n");
#else
    printf("Unknown version\n");
#endif
```

Primitive datatypes: Integers, Floats, and Doubles

```c
#include <stdio.h>

int main() {
    int myInt = 42;                    // 4 bytes, stores whole numbers
    short myShort = -32768;         // 2 bytes, smaller range
    long myLong = 1234567890L;     // 4 or 8 bytes, larger range
    long long myLongLong = 9223372036854775807LL;  // Very large range

    unsigned int myUInt = 100;     // Only positive values

    printf("int: %d\n", myInt);
    printf("short: %d\n", myShort);
    printf("long: %ld\n", myLong);
    printf("long long: %lld\n", myLongLong);
    printf("unsigned int: %u\n", myUInt);

    return 0;
}
```

Primitive datatypes: Integers, Floats, and Doubles

```c
#include <stdio.h>

int main() {
    int myInt = 42;                   // 4 bytes, stores whole numbers
    short myShort = -32768;          // 2 bytes, smaller range
    long myLong = 1234567890L;       // 4 or 8 bytes, larger range
    long long myLongLong = 9223372036854775807LL;  // Very large range

    unsigned int myUInt = 100;       // Only positive values

    printf("int: %d\n", myInt);
    printf("short: %d\n", myShort);
    printf("long: %ld\n", myLong);
    printf("long long: %lld\n", myLongLong);
    printf("unsigned int: %u\n", myUInt);

    return 0;
}
```

```c
#include <stdio.h>

int main() {
    float myFloat = 3.14f;           // 4 bytes, single precision
    double myDouble = 3.14159265;   // 8 bytes, double precision
    long double myLongDouble = 3.141592653589793238L; // Higher precision

    printf("float: %f\n", myFloat);
    printf("double: %lf\n", myDouble);
    printf("long double: %Lf\n", myLongDouble);

    return 0;
}
```

Primitive datatypes: Booleans

```
#include <iostream>

int main() {
    bool isReady = true;     // Can be true or false

    std::cout << "Boolean value: " << isReady << std::endl;

    return 0;
}
```

Non-Primitive datatypes: Strings

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[20] = "Hello"; // Static allocation with buffer size 20

    // Concatenate another string (careful with buffer size!)
    strcat(str, " World!");

    printf("%s\n", str);  // Output: "Hello World!"
    return 0;
}
```

C-Style

```cpp
#include <iostream>
#include <string>

int main() {
    std::string greeting = "Hello";
    greeting += " World!"; // Concatenate easily
    std::cout << greeting << std::endl; // Output: "Hello World!"
    return 0;
}
```

C++ Style

Prefer `std::string` over C-style string

Non-Primitive datatypes: Arrays and Vectors

An **Array** in C is a **fixed-size** collection of **similar data items** stored in a **contiguous memory locations**.

It can be used to store the collection of primitive datatypes and derived and user defined types.

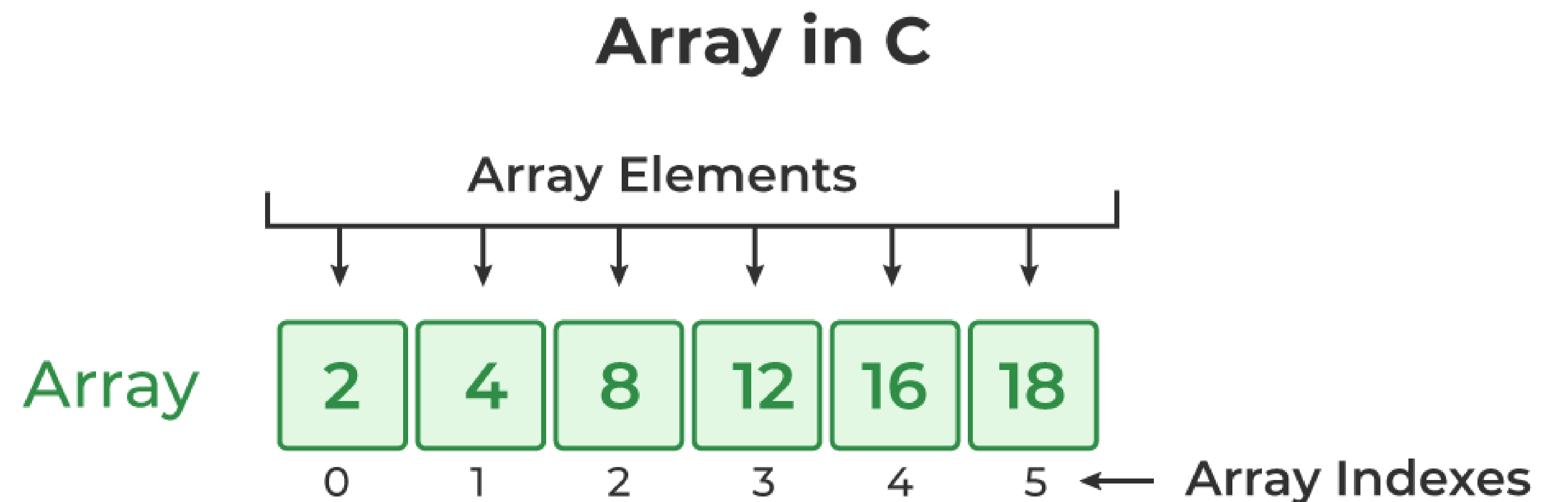


Image: https://www.geeksforgeeks.org/c-arrays/

Non-Primitive datatypes: Arrays and Vectors

```
int arr1[5];            // Uninitialized
int arr2[5] = {0};      // All elements = 0
int arr3[] = {1,2,3};   // Compiler deduces size = 3
```

```
#include <stdio.h>

int main() {
    int arr[3] = {10, 20, 30};
    for(int i = 0; i < 3; i++) {
        printf("Element %d: %d\n", i, arr[i]);
    }
    return 0;
}
```

C-Style

Low-level code, embedded systems, performance-critical sections

Fixed-size that must be defined at **compile time**. Once declared, their size cannot change.

Lack of built-in functions for resizing, inserting, or deleting.

Not **safe**! No built-in bound checking

Minimal overhead due to their simplicity.

Manual memory management.

Non-Primitive datatypes: Arrays and Vectors

```
#include <vector>

std::vector<int> v1;            // Empty vector
std::vector<int> v2(5, 0);      // 5 elements, each initialized to 0
std::vector<int> v3 = {1, 2, 3}; // C++11 list initialization
```

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> scores = {90, 85, 77};
    scores.push_back(95); // Dynamically add an element

    for (int score : scores) {
        std::cout << score << " ";
    }
    // Output: 90 85 77 95
    return 0;
}
```

C++ Style

Dynamic-size array that can grow or shrink as needed during runtime.

Rich interface: `push_back()`, `pop_back()`, `empty()`, `capacity()`, `size()`

Safe! built-in bound checking when using `.at()`

Compatible with `C++` standard library algorithms

Automatic memory management.

Pointers

A **variable** that holds the **memory address** of another variable

```
int x = 5;
int *p = &x;   // Pointer to x
```

```
#include <stdio.h>

int main() {
    int numbers[3] = {10, 20, 30};
    int *p = numbers; // points to the first element of numbers

    printf("%d\n", *(p + 1)); // Same as numbers[1]: 20
    return 0;
}
```

C-Style

Pointers

A **variable** that holds the **memory address** of another variable

```
int x = 5;
int *p = &x;   // Pointer to x
```

Smart Pointers

`std::unique_ptr<T>` : Exclusive ownership

`std::shared_ptr<T>`: shared ownership

Automatic memory management

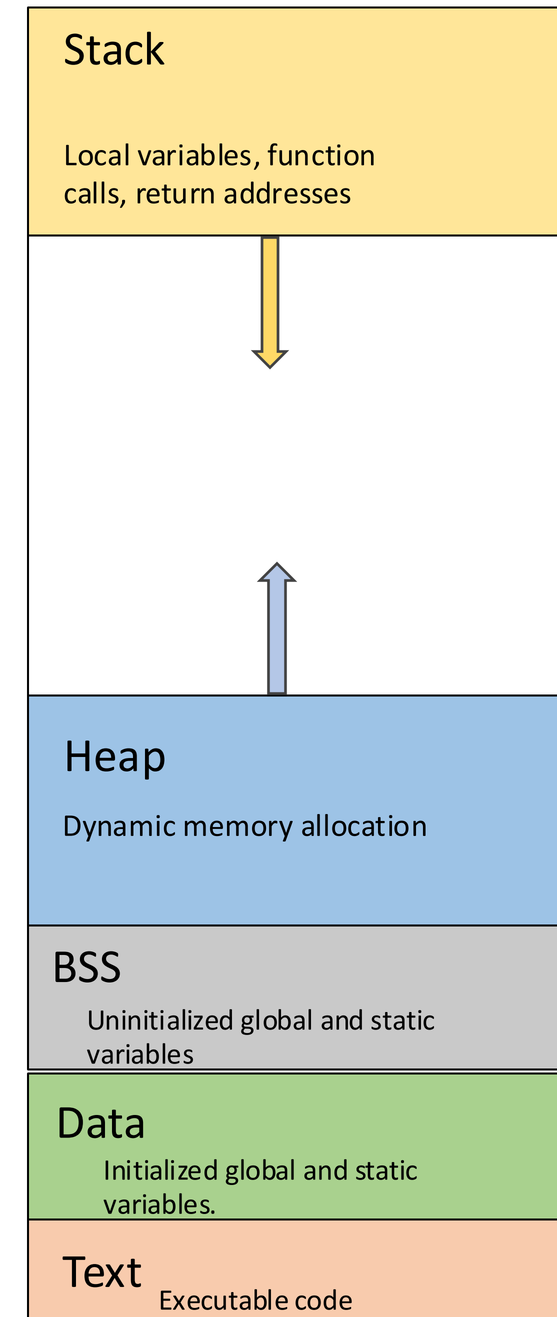
```
#include <stdio.h>

int main() {
    int numbers[3] = {10, 20, 30};
    int *p = numbers; // points to the first element of numbers

    printf("%d\n", *(p + 1)); // Same as numbers[1]: 20
    return 0;
}
```

C-Style

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> ptr = std::make_unique<int>(10);
    std::cout << *ptr << std::endl; // 10
    // No manual delete needed; memory is freed when ptr goes out of scope
    return 0;
}
```

(Modern) C++ Style

C Memory Model & Memory Management



C Memory Model & Memory Management

Stack Memory:

Managed by the OS/Compiler

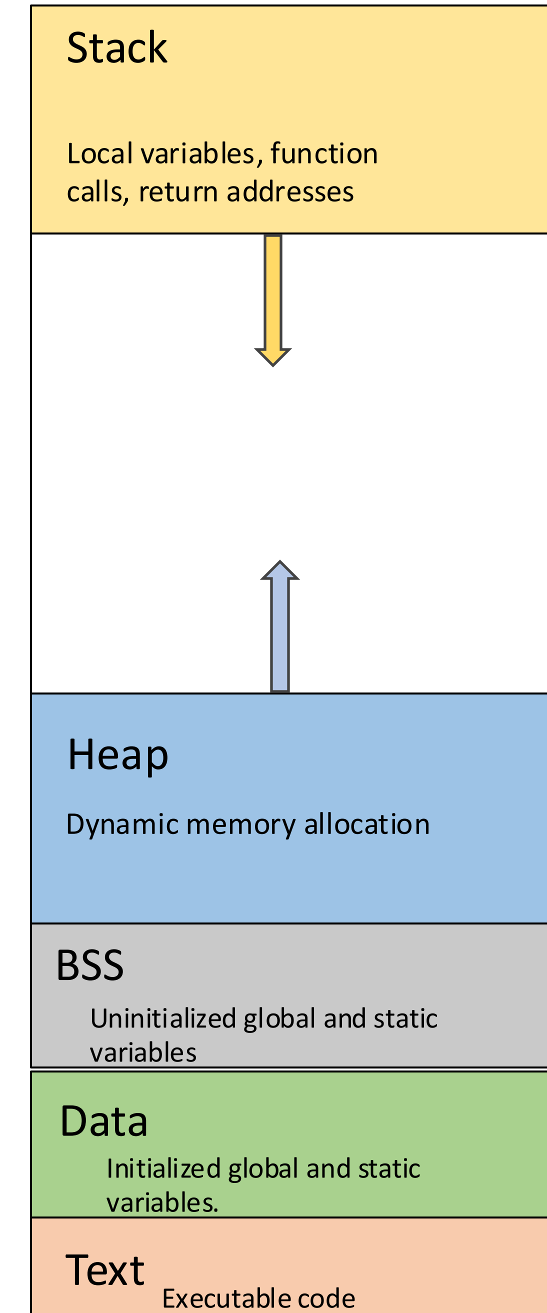
Memory for local variables and function calls

Scope based

Has a fixed, smaller size

Faster than heap

what does `ulimit -a` do on a unix based OS?



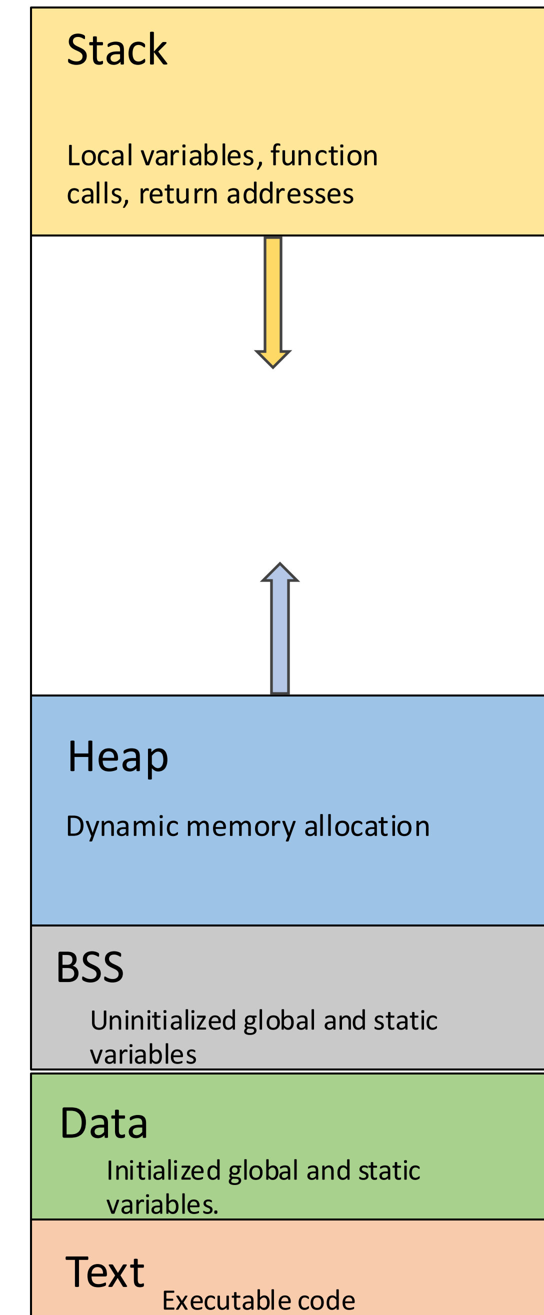
C Memory Model & Memory Management

Stack Memory:

- Managed by the OS/Compiler
- Memory for local variables and function calls
- Scope based
- Has a fixed, smaller size
- Faster than heap
- what does `ulimit -a` do on a unix based OS?

Heap Memory:

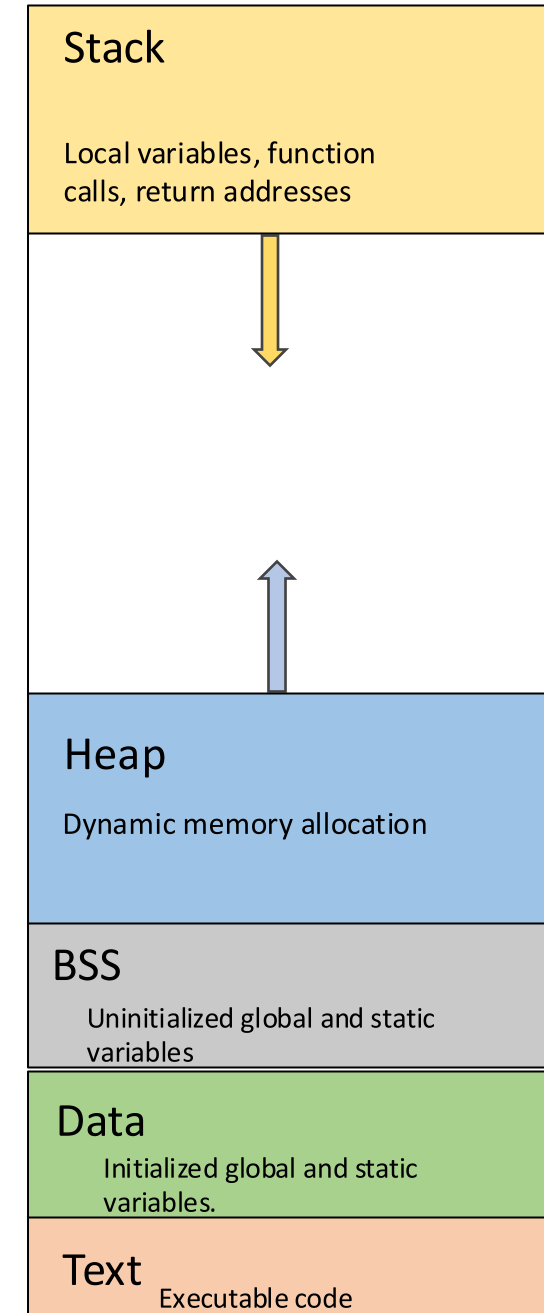
- Managed explicitly by programmers
- Dynamic allocation on runtime
- remains until it is explicitly deallocated
- Much larger than stack
- Slower than stack



C Memory Model & Memory Management

Memory Management becomes even more important when it comes to **GPU programming**

- Memory Data Transfer
- Manage GPU memory
- Manage System Memory



Memory Management

Dynamic memory allocation in C

malloc: allocates a block of memory of a specified size.

calloc: allocates memory for an array of elements and **initializes them to zero**.

realloc: resizes an allocated memory block

free: frees previously allocated memory

Always check if the allocation was successful

Ensure that every allocated block is later freed to avoid memory leaks.

```c
// Allocate memory for an array of 5 integers
int *arr = (int*)malloc(5 * sizeof(int));
if (arr == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    return 1;
}

// Initialize and use the allocated memory
for (int i = 0; i < 5; i++) {
    arr[i] = i * 10;
}

// Print the values
for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}
printf("\n");

// Free the allocated memory
free(arr);
```

Memory Management

Dynamic memory allocation in C++

new: allocates a block of memory of a specified size.

delete: frees a memory block

No casting required

```
// Allocate a single integer on the heap
int *p = new int(42);
std::cout << "Value: " << *p << std::endl;
delete p;  // Free the memory

// Allocate an array of integers on the heap
int *arr = new int[5];
for (int i = 0; i < 5; i++) {
    arr[i] = i * 10;
}

// Display the array values
for (int i = 0; i < 5; i++) {
    std::cout << arr[i] << " ";
}
std::cout << std::endl;

// Free the array memory
delete[] arr;
return 0;
```

Memory Management

Memory Management Best Practices

1. **Consistent Allocation/Deallocation:** Match each `malloc` with `free` and each `new` with `delete`.
2. **Error Checking:** Always verify that memory allocation was successful
3. **Ownership Clarity:** Clearly define which part of your code owns a piece of memory.
4. Once memory is freed, set the pointer to `NULL` (or `nullptr` in `C++`)
5. Use `Valgrind` to detect memory leaks.

Memory Management

MEMORY LEAKS

memory is allocated for an array of integers but never freed before the program ends.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr = (int *)malloc(10 * sizeof(int));
    if (ptr == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    // Use the allocated memory
    for (int i = 0; i < 10; i++) {
        ptr[i] = i;
    }

    // Memory leak: allocated memory is not freed
    return 0;
}
```

Memory Management

MEMORY LEAKS

a function that allocates memory for a string on each call but never frees it, leading to **repeated leaks.**

```c
#include <stdio.h>
#include <stdlib.h>

void leak_memory() {
    char *buffer = (char *)malloc(50 * sizeof(char));
    if (buffer == NULL) {
        return;
    }
    // Do something with buffer
    snprintf(buffer, 50, "This is a leaked string.");
    printf("%s\n", buffer);
    // Memory leak: 'buffer' is not freed
}

int main() {
    for (int i = 0; i < 5; i++) {
        leak_memory();
    }
    return 0;
}
```

Memory Management: Modern C++ Techniques

RAII (Resource Acquisition Is Initialization):

allocate resources (memory, file handles, sockets, etc.) in a constructor and release them in the destructor

```
class ManagedBuffer {
    char* buffer;
    size_t size;
public:
    ManagedBuffer(size_t n) : size(n), buffer(new char[n]) {
        std::cout << "Allocated " << size << " bytes.\n";
    }
    ~ManagedBuffer() {
        delete[] buffer;
        std::cout << "Memory deallocated.\n";
    }
    void fill(const char* data) {
        std::strncpy(buffer, data, size - 1);
        buffer[size - 1] = '\0';
    }
    void display() const {
        std::cout << "Buffer: " << buffer << "\n";
    }
};

int main() {
    ManagedBuffer buf(50);
    buf.fill("Short RAII example.");
    buf.display();
    return 0;
}
```

Memory Management: Modern C++ Techniques

Smart pointers (C++11)

Implement RAII automatically

```cpp
#include <iostream>
#include <memory>

class MyClass {
public:
    MyClass() { std::cout << "MyClass constructed\n"; }
    ~MyClass() { std::cout << "MyClass destructed\n"; }
};

int main() {
    // unique_ptr: Exclusive ownership
    {
        std::unique_ptr<MyClass> uniqueObj = std::make_unique<MyClass>();
        // uniqueObj is automatically deleted when going out of scope.
    }

    // shared_ptr: Shared ownership
    {
        std::shared_ptr<MyClass> sharedObj1 = std::make_shared<MyClass>();
        {
            std::shared_ptr<MyClass> sharedObj2 = sharedObj1;
            // Both sharedObj1 and sharedObj2 point to the same object.
        } // sharedObj2 goes out of scope, but the resource is not freed.
        // Resource is freed when sharedObj1 goes out of scope.
    }

    return 0;
}
```

Structs

Allows to group related data.

```cpp
#include <iostream>

struct Point3D {
    float x, y, z;
};

int main() {
    Point3D sensorPosition = {10.0f, 20.0f, 30.0f};
    std::cout << "Sensor Position: ("
              << sensorPosition.x << ", "
              << sensorPosition.y << ", "
              << sensorPosition.z << ")" << std::endl;
    return 0;
}
```

Functions

Pass by value

```cpp
#include <iostream>

void incrementValue(int x) {
    x++; // Only a copy is incremented
}

int main() {
    int number = 10;
    incrementValue(number);
    std::cout << number << std::endl; // Still 10
    return 0;
}
```

Functions

Pass by reference (C++)

```cpp
#include <iostream>

void incrementReference(int &x) {
    x++; // Modifies the original
}

int main() {
    int number = 10;
    incrementReference(number);
    std::cout << number << std::endl; // 11
    return 0;
}
```

Functions

Inline Functions

```
inline int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(3, 4);
    // The compiler may substitute the function body directly for efficiency
    return 0;
}
```

Functions

Lambda Functions (C++11)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {1, 4, 2, 5, 3};

    // Sort with a custom lambda comparator
    std::sort(numbers.begin(), numbers.end(), [](int a, int b) {
        return a < b; // ascending
    });

    // Print the sorted vector
    for (auto num : numbers) {
        std::cout << num << " ";
    }
    return 0;
}
```

I/O in C++

Reading / Writing files

```cpp
#include <iostream>
#include <fstream>
#include <string>

int main() {
    // Writing to a file
    {
        std::ofstream outFile("example.txt");
        if (outFile.is_open()) {
            outFile << "Hello, file!\n";
            outFile << 123 << std::endl;
        }
        // outFile closes automatically
    }

    // Reading from a file
    {
        std::ifstream inFile("example.txt");
        if (inFile.is_open()) {
            std::string line;
            while(std::getline(inFile, line)) {
                std::cout << line << std::endl;
            }
        }
        // inFile closes automatically
    }
    return 0;
}
```

Did we miss anything?

Cmake - an open-source build system that manages the build process of C/C++ projects across different platforms and compilers using configuration files.

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

# Set C++ standard (optional but common)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Add executable from source files
add_executable(my_app main.cpp)
```

kitware kitware.com contact us
CMake about solutions getting started documentation customize Enter Keyword DOWNLOAD
kitware
CMake
BUILD YOUR WORLD

CMake: A Powerful Software Build System

CMake is the de-facto standard for building C++ code, with over 2 million downloads a month. It's a powerful, comprehensive solution for managing the software build process. Get everything you need to successfully leverage CMake by visiting our resources section.

GETTING STARTED DOCUMENTATION

Visit https://www.cmake.org for tutorials.

Next Lecture: Shared Memory Programming