

UW Madison Math/CS 714

Methods of Computational Mathematics I

Optional activity 1: Introduction to C++¹

Instructor: Yue Sun (yue.sun@wisc.edu)

September 10, 2025

¹Based off slides from Prof. [Chris Rycroft](#) and Rycroft Group alumni [Dan Fortunato](#), [Nick Derr](#), [Nick Boffi](#), and [Michael Emanuel](#).

Why learn C/C++?

Isn't Python the hot language these days? Isn't C/C++ notoriously hard to learn and painful to debug?

Many people would say “yes” to both.

But ...

- C++ is **fast**. In many applications that is crucial.
- C++ is **highly expressive** and lets you **get close to the hardware**.
- C++ provides features for organizing and managing large projects.
- C and C++ have stood the test of time.

Compiled vs. interpreted languages

In a compiled language:

- You write a program in source code, e.g. C++
- You run a special program called a **compiler**, e.g. g++
- This creates an executable program: machine language that can run on your hardware

In an interpreted language:

- You write a program or just one line of source code, e.g. Python
- You run a program called an **interpreter**, e.g. python3
- The interpreter runs each line of code one at a time

The interpreter

The interpreter itself is an executable program that was compiled

Question: What language was the Python interpreter written in?

References

There are several excellent references written by the their creators:

- Brian W. Kernighan and Dennis M. Ritchie, *C Programming Language, 2nd edition*, Pearson (1988).
- Bjarne Stroustrup, *The C++ programming language, 4th edition*, Addison-Wesley (2013).

There are also many excellent online resources:

- <https://cplusplus.com>
- <https://en.cppreference.com/w/>
- <https://godbolt.org>

Anatomy of a C++ program

- A program begins execution in the `main()` function, which is called automatically when the program is run.
- Code from external libraries can be used with the `#include` directive.

```
#include <cstdio> // Allows us to use printf

int main()
{
    printf("Hello world!\n");
    return 0; // Indicates the program exited normally
}
```

```
Hello world!
```

Syntax

C++ is...

- Case-sensitive
- Whitespace-insensitive
- Statically typed

```
#include <iostream>
// this code will compile

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

int main()
{
    double x = 1.2;
    std::cout << cube(x) << std::endl;
    return 0;
}
```

Syntax

C++ is...

- Case-sensitive
- Whitespace-insensitive
- Statically typed

```
#INCLUDE <Iostream>
// this code will throw an error, because keywords are
// incorrectly capitalized

DOUBLE cube(Double x) {
    RETURN x*X*x;
}

INT MAIN()
{
    double X = 1.2;
    STD::COUT << Cube(x) << std::endl;
    ReTurn 0;
}
```


Syntax

C++ is...

- Case-sensitive
- Whitespace-insensitive
- Statically typed

```
#include <iostream>
// this code will compile, despite the weird formatting

double cube(double x){return x * x*x;}
int main( ){double x=1.2;std::cout<<cube(x)<<
std::endl; return 0;

}
```

Syntax

C++ is...

- Case-sensitive
- Whitespace-insensitive
- Statically typed

```
#include <iostream>
// this code will not compile, because it is missing
// type information

cube(x) {
    return x*x*x;
}

main() {
    x = 1.2;
    std::cout << cube(x) << std::endl;
    return 0;
}
```

A note about IO

There are different ways to print strings in C++.

```
#include <cstdio>    // C-style
#include <iostream>  // C++-style

int main()
{
    puts("I am C-style.");
    printf("Me too!\n");
    std::cout << "I am C++-style." << std::endl;
    return 0;
}
```

```
I am C-style.
Me too!
I am C++-style.
```

A note about comments

Use `//` to comment a single line, or `/* ... */` for multiple.

Comments are useful but easy to misuse. Here are some good rules to keep in mind.

- ① **Code** tells **what** you are doing.
Comments tell **why** you are doing it.
- ② Only comment what you cannot express in code.
- ③ If a comment restates code, delete it.

A note about comments

Use `//` to comment a single line, or `/* ... */` for multiple.

Comments are useful but easy to misuse. Here are some good rules to keep in mind.

- ① **Code** tells **what** you are doing.
Comments tell **why** you are doing it.
- ② Only comment what you cannot express in code.
- ③ If a comment restates code, delete it.

```
// Loop from 0 to 10000  
for (int x=0; x<10000; ++x) {  
    ...  
}
```

A note about comments

Use `//` to comment a single line, or `/* ... */` for multiple.

Comments are useful but easy to misuse. Here are some good rules to keep in mind.

- 1 **Code** tells **what** you are doing.
Comments tell **why** you are doing it.
- 2 Only comment what you cannot express in code.
- 3 If a comment restates code, delete it.

```
// Loop from 0 to 10000  
for (int x=0; x<10000; ++x) {  
    ...  
}
```

```
// Calculate primes up to a cutoff  
for (int x=0; x<10000; ++x) {  
    ...  
}
```

Types

Built-in types

What types are built into C++?

Type	Example	Size [†]
bool	true	1 byte
char	'a'	1 byte
short	-2	2 bytes
int	-2	4 bytes
long	-2	8 bytes
float	3.4	4 bytes
double	3.4	8 bytes
void	—	—

[†]These sizes may be different on your computer. The only guarantee is that $1 = \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$.

Types

Strings

Strings are not built into C++, but can be included from an external library.

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

int main()
{
    std::string s1 = "Hello";
    std::string s2 = " world!";
    std::string s3 = s1 + s2;
    std::cout << s3 << std::endl;
    std::cout << s3.size() << std::endl;
    return 0;
}
```

```
Hello world!
12
```


Types

Type modifiers

We can modify some types with the prefixes:

- ① short
- ② long
- ③ unsigned
- ④ signed

#1–4 can prefix int, #3–4 can prefix char, and #2 can prefix double.

Types

Type modifiers

We can also add the prefix **const** to any type. This indicates that the value of the variable cannot be changed; you must initialize a const variable when it is declared.

```
const int x = 1; // OK
x = 2;           // Error
const double y; // Error
y = 3.0;         // Error
```

This may seem unnecessary, but it will prove to be very useful. Using const will help the compiler enforce rules you set and force your functions to adhere to contracts about the data they use.

Types

Typedefs

We can also define our own types using the typedef keyword.

```
typedef unsigned int size_t; // Conveys meaning
typedef float real;          // Easy to change code to double later
```

It may seem trivial, but typedefs can help you

- ... convey meaning in your code
(“What does this type actually represent?”)
- ... allow for easy modifications down the road
(change one line of code instead of hundreds)

Types

References

A **reference** is an alias to an existing variable. Any type can be made into a reference type by adding an & after the type's name.

```
double x;  
double& y = x; // y is a reference to x  
x = 1.0;  
printf("x = %g, y = %g\n", x, y);  
y = 2.0;  
printf("x = %g, y = %g\n", x, y);
```

```
x = 1, y = 1  
x = 2, y = 2
```

A reference must be initialized when it is created. Once a reference is initialized to a variable, it cannot be changed to refer to another variable.

Types

Pointers

Each variable has an address in memory. The & operator can be used to take the address of a variable.

```
int x = 1;  
printf("%p\n",&x);
```

```
0x7ffeef76096c
```

Types

Pointers

A **pointer** is a variable whose value is such an address. Any type can be made into a pointer type by adding an `*` after the type's name.

We can retrieve the value of the variable being pointed to by using the dereference operator, which is a `*` before the pointer's name.

```
double x = 1.0;
double* y = &x; // y is a pointer to x
x = 1.0;
printf("x = %g, y = %g\n", x, *y);
*y = 2.0;
printf("x = %g, y = %g\n", x, *y);
```

```
x = 1, y = 1
x = 2, y = 2
```

Types

Arrays

An **array** is a collection of variables of the same type that is contiguous in memory. To make an array, add `[]` after the variable's name with a size inside the brackets.

```
double a[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};  
double b[] = {1000.0, 2.0, 3.4, 17.0, 50.0}; // Same  
int c[10];  
c[2] = 1;
```

Note that arrays are **zero-indexed**. An array can also be used as a pointer.

```
double a[] = {1000.0, 2.0, 3.4, 17.0, 50.0};  
double* p = a;  
double* q = &a[0]; // Same
```

Scoping

All variables have a **scope**—the extent of code in which they exist. A variable's scope begins when it is declared and ends at the closing curly brace of that level.

```
int a = 1; // a's scope begins. a has global scope.

int main()
{
    double b; // b's scope begins
    b = 2.7;
    bool c = true; // c's scope begins
    if (a < b) {
        bool d = c; // d's scope begins
    } else {        // d's scope ends
        c = false;
    }

    return 0;
} // b and c's scopes end
```


Functions

Declaration and definition

Functions must be **declared** before they are called.

They can be **defined** before or after they are called.

```
#include <stdio>

// f is declared and defined here
double f(double x, double y) {
    return x/y + y/x;
}

int main() {
    double a = -3.2, b = 7.5, c = f(a,b);
    printf("%g\n",c);
    return 0;
}
```

-2.77042

Functions

Declaration and definition

Functions must be **declared** before they are called.

They can be **defined** before or after they are called.

```
#include <stdio>

// f is declared here, but not defined
double f(double x, double y);

int main() {
    double a = -3.2, b = 7.5, c = f(a,b);
    printf("%g\n", c);
    return 0;
}

// f is defined here
double f(double x, double y) {
    return x/y + y/x;
}
```

-2.77042

Functions

Passing arguments

Functions can take and return arguments of any type.
Arguments that are **passed by value** are local to the function.

```
#include <stdio>

void add_two(int x) {
    x += 2;
}

int main() {
    int x = 1;
    add_two(x);
    printf("%d\n", x);    // What will this print?
    return 0;
}
```

Functions

Passing arguments

Functions can take and return arguments of any type.
Arguments that are **passed by value** are local to the function.

```
#include <stdio>

void add_two(int x) {
    x += 2;
}

int main() {
    int x = 1;
    add_two(x);
    printf("%d\n", x);    // What will this print?
    return 0;
}
```

1

Functions

Passing arguments

Functions can take and return arguments of any type.
Arguments that are **passed by reference** can be modified.

```
#include <stdio>

void add_two(int& x) {
    x += 2;
}

int main() {
    int x = 1;
    add_two(x);
    printf("%d\n", x);    // What will this print?
    return 0;
}
```

Functions

Passing arguments

Functions can take and return arguments of any type.
Arguments that are **passed by reference** can be modified.

```
#include <stdio>

void add_two(int& x) {
    x += 2;
}

int main() {
    int x = 1;
    add_two(x);
    printf("%d\n", x);    // What will this print?
    return 0;
}
```

3

Functions

Passing arguments

Functions can take and return arguments of any type.

Arguments that are **passed by pointer** can have their contents modified.

```
#include <stdio>

void add_two(int* x) {
    *x += 2;
}

int main() {
    int x = 1;
    add_two(&x);
    printf("%d\n", x);    // What will this print?
    return 0;
}
```

Functions

Passing arguments

Functions can take and return arguments of any type.

Arguments that are **passed by pointer** can have their contents modified.

```
#include <stdio>

void add_two(int* x) {
    *x += 2;
}

int main() {
    int x = 1;
    add_two(&x);
    printf("%d\n", x);    // What will this print?
    return 0;
}
```

3

Functions

Passing arguments

Passing a **const reference** ensures your functions can only alter what they are supposed to.

```
#include <cstdio>
#include <string>

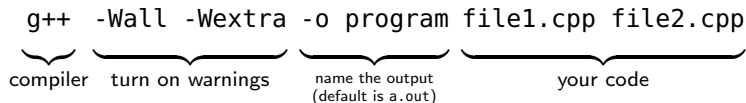
void get_length(const std::string& s, int& len) {
    s = "I am malicious!"; // Will not compile
    len = s.size();
}

int main() {
    int len;
    std::string s = "My length is 16.";
    get_length(s, len);
    printf("%d\n", len);
    return 0;
}
```

Compiling

Basic commands

`g++ -Wall -Wextra -o program file1.cpp file2.cpp`



compiler turn on warnings name the output
(default is a.out) your code

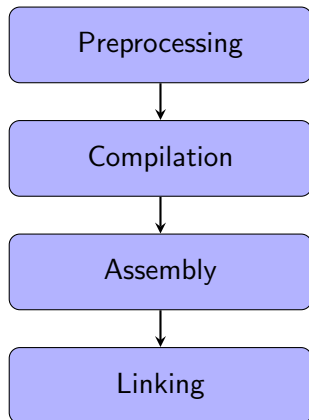
The compiler can generate **warnings**, which allow compilation to continue, and **errors**, which abort the compilation.

Other useful options:

- `-pedantic`: Even more warnings
- `-std=c++11`: Warn if you violate the C++11 standard
(or any standard you choose)
- `-O0` – `-O3`: Optimize to varying degrees, from least to most
- `-g`: Include debug information (will be useful later)

Compiling

The stages of compilation

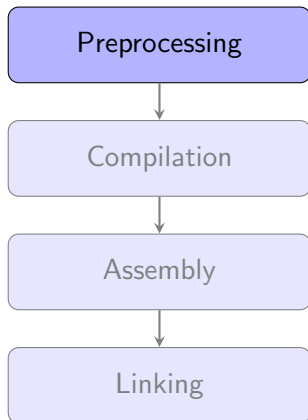


```
#include <stdio>

int main()
{
    // Comment
    printf("Hello world!\n");
    return 0;
}
```

Compiling

The stages of compilation

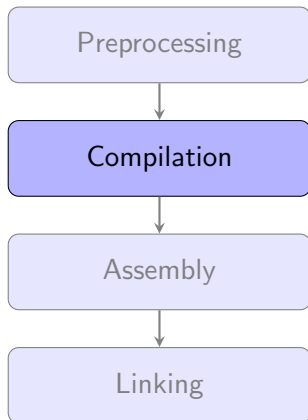


```
...  
int printf(const char * , ...)  
    __attribute__((__format__ (  
        __printf__, 1, 2)));  
...  
  
# 109 "/usr/include/stdio.h" 2 3  
# 2 "hello_world.cpp" 2  
  
int main()  
{  
  
    printf("Hello world!\n");  
    return 0;  
}
```

- Strips comments
- Processes lines starting with #
- Inserts contents of header files

Compiling

The stages of compilation



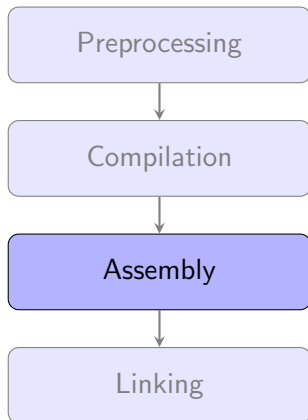
```
...
_main:

    ## @main
    .cfi_startproc
## BB#0:
    pushq    %rbp
Lcfi0:
    .cfi_def_cfa_offset 16
Lcfi1:
    .cfi_offset %rbp, -16
    movq     %rsp, %rbp
Lcfi2:
    .cfi_def_cfa_register %rbp
    subq     $16, %rsp
    leaq     L_.str(%rip), %rdi
    movl     $0, -4(%rbp)
...
```

- Translates to assembly instructions
- Specific to processor architecture

Compiling

The stages of compilation

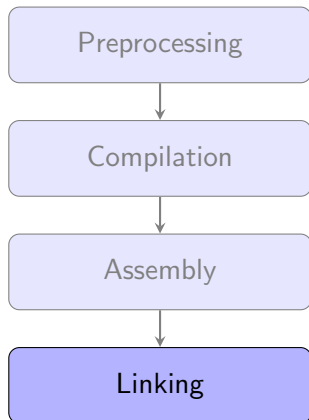


```
cffa edfe 0700 0001 0300 0000 0100 0000
0400 0000 0002 0000 0020 0000 0000 0000
1900 0000 8801 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
9800 0000 0000 0000 2002 0000 0000 0000
9800 0000 0000 0000 0700 0000 0700 0000
0400 0000 0000 0000 5f5f 7465 7874 0000
0000 0000 0000 0000 5f5f 5445 5854 0000
0000 0000 0000 0000 0000 0000 0000 0000
2a00 0000 0000 0000 2002 0000 0400 0000
b802 0000 0200 0000 0004 0080 0000 0000
...
```

- Translates to binary object code
- Instructions for processor

Compiling

The stages of compilation



```
cffa edfe 0700 0001 0300 0080 0200 0000
0f00 0000 b004 0000 8500 2000 0000 0000
1900 0000 4800 0000 5f5f 5041 4745 5a45
524f 0000 0000 0000 0000 0000 0000 0000
0000 0000 0100 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 1900 0000 d801 0000
5f5f 5445 5854 0000 0000 0000 0000 0000
0000 0000 0100 0000 0010 0000 0000 0000
0000 0000 0000 0000 0010 0000 0000 0000
0700 0000 0500 0000 0500 0000 0000 0000
...
```

- Fills in missing pieces from external

(e.g. printf)
- Puts things in the right order
- Produces the final program

Header files

We can implement functionality across multiple files.

- A **header file** exposes the interface to our functionality. It is only required to contain a function's declaration.
 - This is all the caller of a function needs to know in order to call it.
 - Usually named e.g. `file.h` or `file.hh`.
- An **implementation file** implements the functionality. It contains a function's definition.
 - Usually named e.g. `file.cpp` or `file.cc`.

Header files

add.hh

```
// Protect against multiple or circular includes
#ifndef ADD_HH
#define ADD_HH

int add(int x,int y);

#endif
```

add.cc

```
int add(int x,int y)
{
    return x+y;
}
```

main.cc

```
#include <stdio>
#include "add.hh"

int main()
{
    printf("%d\n",add(3, 4));
    return 0;
}
```

g++ add.cc main.cc

Memory management

Stack vs. heap

There are two types of memory in C++. Both are stored in RAM but have different properties.

- **Stack:** Memory reserved by the CPU each time a function is called and freed when a function returns. Holds local variables and bookkeeping data. Each thread has a stack.
- **Heap:** Memory that can be dynamically allocated/freed at runtime by the program. Threads share the heap.

Memory management

Stack vs. heap

There are two types of memory in C++. Both are stored in RAM but have different properties.

- **Stack:** Memory reserved by the CPU each time a function is called and freed when a function returns. Holds local variables and bookkeeping data. Each thread has a stack.
- **Heap:** Memory that can be dynamically allocated/freed at runtime by the program. Threads share the heap.

Stack	Heap
<code>double x[10];</code>	<code>double* x = new double[10];</code>
faster access (likely in cache)	slower access (likely out of cache)
memory has local scope	memory has global scope
compiler can make lots of optimizations	compiler can't make lots of optimizations
limits on size	no limits on size
fixed size known compile time	size can change at runtime
automatically deallocated by CPU	must deallocate yourself (<code>delete</code>)

Memory management

Stack vs. heap

Every new must have a matching delete!

```
#include <iostream>

int main()
{
    const int m = 10;
    double x[m]; // Stack allocated

    double* y = new double; // Heap allocated

    int n;
    std::cout << "Input a size: ";
    std::cin >> n;
    double* z = new double[n]; // Heap
                               allocated

    delete y;
    delete [] z;

    return 0;
}
```

Memory management

Common pitfalls

The compiler will likely not catch memory errors, which may cause your program to crash at runtime.

- ① Segmentation faults
 - Index out of bounds
 - Dereference a null pointer
- ② Uninitialized memory
- ③ Off-by-one errors
- ④ Memory leaks

```
double* x = new double[10];  
x[11] = 1.0; // Segmentation fault
```

```
int* x = NULL;  
*x = 1; // Segmentation fault
```

Memory management

Common pitfalls

The compiler will likely not catch memory errors, which may cause your program to crash at runtime.

- ① Segmentation faults
 - Index out of bounds
 - Dereference a null pointer
- ② Uninitialized memory
- ③ Off-by-one errors
- ④ Memory leaks

```
void scale(double c, double* x, int N) {  
    for (int i=0; i<N; ++i)  
        x *= c;  
}  
...  
double* x = new double[10]; // x is uninitialized. What if it is NaN?  
scale(0, x, 10);           // Is x guaranteed to be zero after this?
```

Memory management

Common pitfalls

The compiler will likely not catch memory errors, which may cause your program to crash at runtime.

- ① Segmentation faults
 - Index out of bounds
 - Dereference a null pointer
- ② Uninitialized memory
- ③ Off-by-one errors
- ④ Memory leaks

```
int m = 4, n = 5, size = m*n;
double* x = new double[size];
for (int i=size-1; i>0; --i) // The loop does not set x[0]
    x[i] = size - i;
```

Memory management

Common pitfalls

The compiler will likely not catch memory errors, which may cause your program to crash at runtime.

- ① Segmentation faults
 - Index out of bounds
 - Dereference a null pointer
- ② Uninitialized memory
- ③ Off-by-one errors
- ④ Memory leaks

```
if (...) {  
    double x[10];  
    double* y = new double[10];  
    ...  
} // x and y go out of scope here  
// The memory for x (on the stack) is freed, but for y (on the heap) is not, since we forgot  
// Since y is now out of scope, we can't free it anymore!
```


Speed test: Ridders' root-finding method

Consider a continuous function f defined on an interval $[a, b]$ and assume that $f(a) < 0$ and $f(b) > 0$.

The **bisection method** is a simple approach to efficiently find a root of f over the interval. Define $c = (a + b)/2$, and compute $f(c)$. There are two cases:

- If $f(c) < 0$ then a root is in $[c, b]$.
- If $f(c) \geq 0$ then a root is in $[a, c]$.

In each case, the length of the interval containing the root is halved.

When the procedure is applied iteratively, the interval rapidly converges around a root of f .

Speed test: Ridders' root-finding method

The bisection method only uses the sign of f at the evaluation points. It does not use the values of f .

Ridders' method improves upon this, by using the function values to converge more rapidly on the root.³

As a speed test, suppose that we wanted to construct a table of inverse cosines. We want to find the root of

$$f(x; \lambda) = \lambda - \cos x$$

for many values of $\lambda \in [-1, 1]$.

³See [AM225 root-finding notes](#) for a complete description of the method.

Speed test: Ridders' root-finding method

The programs `ridders_array.py` and `ridders_array.cc` implement Ridders' method in Python and C++. They compute the inverse cosines using 4×10^6 values of λ .

Before running the program, take a guess at the relative speed difference between the two implementations.

Typical output

The Python version reports times of

```
Time: 40.51 s (total)
Time: 10.1268 microseconds (per value)
```

The C++ version reports times of

```
Time: 0.763 s (total)
Time: 0.190742 microseconds (per value)
```

This represents a factor of 53 speed difference. This will depend on hardware and software versions, although in general C++ is at least an order of magnitude faster.

Comments on the speed comparison

The relative slowness of Python is well-documented and is due to many reasons: interpreted language, dynamic typing, *etc.*⁴

There are many considerations in the language choice:

- Python offers great flexibility
- Many Python library routines (e.g. NumPy) are in compiled code and are much faster
- Extra speed not required for many tasks; need to weigh the time of the programmer against the time of computation

Compiled languages are a good choice for critical code bottlenecks.

⁴See, e.g., <https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/>

Matrix computations and timing

The directory `matrices` contains several examples of matrix operations and timing. The source code files `matrix.cc` and `matrix.hh` implement routines to construct matrices and print them.

A matrix $A \in \mathbb{R}^{m \times n}$ is stored as a one-dimensional array with mn elements. The routines use [row-major ordering](#), meaning that element a_{ij} is stored at position $ni + j$.⁵

⁵Assuming that the matrix rows and columns are indexed from zero.

Matrix computations and timing

The program `mat_timing.cc` measures the time to make random matrices of different sizes m .

For each size, the program tries to generate as many random matrices as possible in a 1 s interval.

Suppose N trials were completed in time $T > 1$ s. Then the average time for one trial is T/N . This gives better timing accuracy for timing trials that can be computed quickly.