CS100 Recitation 7

Contents

Transition from C to C++

- Using Compilers
- C++ Language Standardization
- VS Code Configuration for C++

Using Compilers

Recall the procedure for compiling a **C** program (e.g., `hello.c`):

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

In this command, `gcc` is the GNU C Compiler executable (e.g., `gcc.exe` on Windows).

The analogous process for compiling a **C++** program involves using **g++**—the dedicated C++ compiler—and source files with the **.cpp** extension. For example:

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

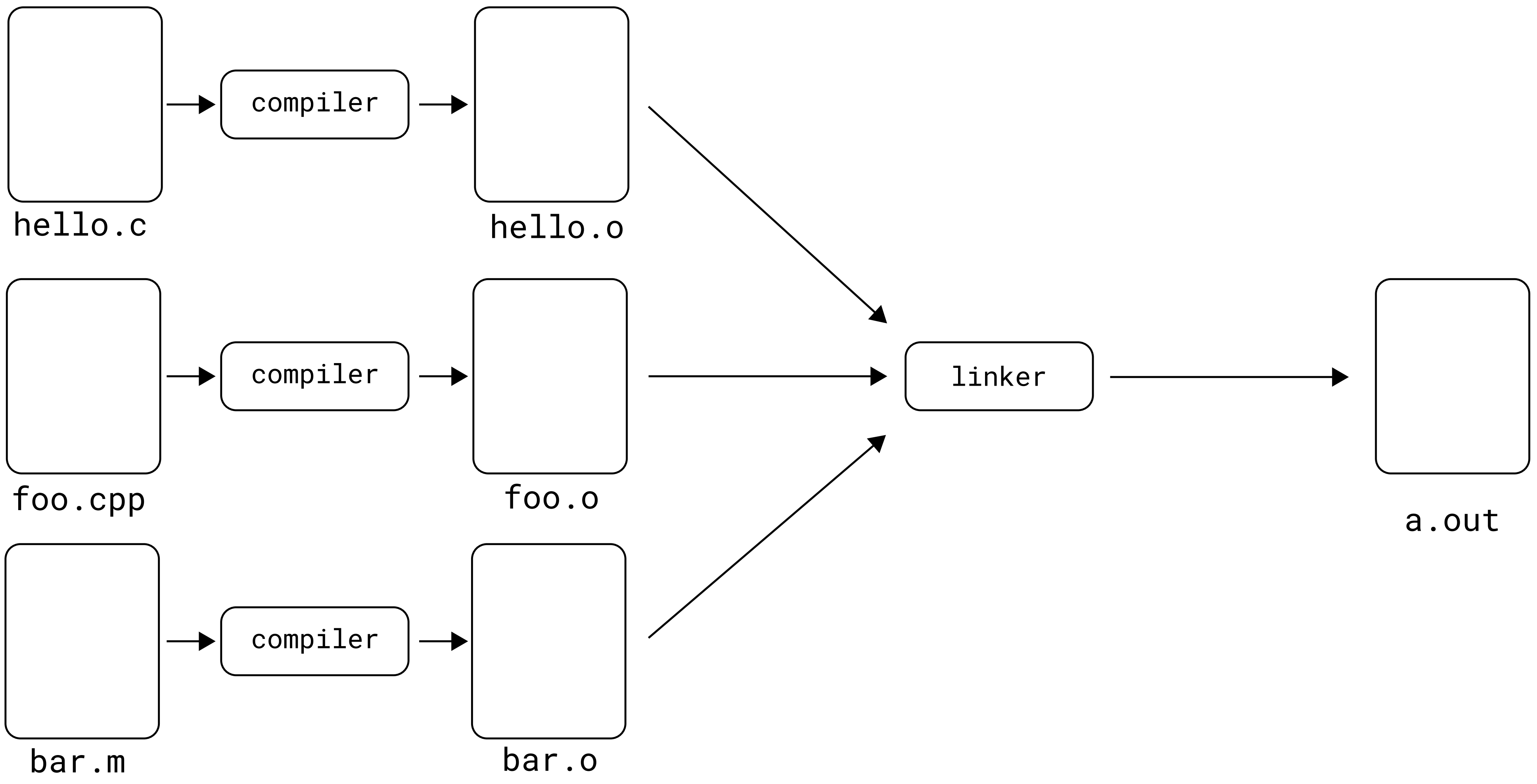
Note: On Windows, the C++ compiler is typically named `g++.exe`.

The GNU Compiler Collection (GCC)

GCC (***GNU Compiler Collection***) is an **integrated suite** of compilers that supports multiple programming languages, including *C*, *C++*, *Objective-C*, and *Java*, among others. When compiling C programs specifically, **GCC** is often synonymous with the **GNU C Compiler**.

- The behavior of the `gcc` command is **context-sensitive**; it selects the appropriate compilation backend based on the **input file type**.
- You may explicitly specify the language using the `-x language` option.

Differences Between GCC and G++



Differences Between GCC and G++ (cont.)

G++ is a compiler dedicated to processing C++ programs. Unlike GCC when used for C++, g++ directly compiles C++ source code without converting it to an intermediate C form. The primary distinctions between **GCC** and **G++** ***in the context of C++ compilation*** are:

- **G++** automatically links the **C++ standard library** and, by default, treats .c and .h files as C++ source files (unless overridden using the -x option).
- When using **GCC** for compiling C++ code, the C++ standard library is not linked automatically. It is necessary to include linker flags (e.g., -lstdc++ and -shared-libgcc) explicitly.

In effect, executing g++ hello.cpp -o hello is equivalent to running:

```
gcc -x c++ hello.cpp -o hello -lstdc++ -shared-libgcc
```

Exercise 1: Compiling C and C++ Programs

File 1: `c_program.c`

```
#include <stdio.h>

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

File 2: `cpp_program.cpp`

```
#include <iostream>

int main() {
  std::cout << "Hello world\n";
  return 0;
}
```

Compile these files using `gcc` *for the C program and* `g++` *for the C++ program.*

C++ Language Standardization

- For this course, we adhere to the **C++17** standard.
- To enforce C++17 during compilation, include the `-std=c++17` flag. For example:

```
g++ hello.cpp -o hello -std=c++17
```

Configuring VS Code for C++ Development

- **Compilation Flag:**
  In the `settings.json` file under `code-runner.executorMap`, ensure that the `"cpp"` entry includes the `-std=c++17` flag.
- **IntelliSense Configuration:**
  In `c_cpp_properties.json`, set the `"cppStandard"` field to `c++17`.
- **Debugging Setup:**
  The recommended approach is to remove the existing `tasks.json` and `launch.json` files. VS Code will regenerate these configuration files automatically when debugging a C++ program.
  Ensure that the debugger is configured to use `g++.exe` for C++ debugging.

Introduction to C++

- Overview of C++
- Historical Evolution of C++ Standards
- "Hello World" Example in C++

Overview of C++: A Federation of Languages

- C++ originated as an extension of the C programming language, primarily introducing **classes**.
- Over time, C++ has evolved into a **multiparadigm programming language**, incorporating: procedural programming, object-oriented programming, functional programming, generic programming, meta-programming.

C Blocks Preprocessor Built-in data type Arrays Pointers etc.	**Template** Generic programming TMP
Object Oriented C++ Classes Encapsulation Inheritance Virtual functions	**STL** Containers Iterators Algorithms etc.

- Due to its diverse features, C++ is often categorized into four sublanguages: **C**, **Object-Oriented C++**, **Template C++**, **Standard Template Library (STL)**

First Sub-Language: Procedural C

- **C** is a procedural programming language that focuses on the use of functions.
- At its core, C++ retains many elements of C, including:
  - Code blocks
  - Pointers
  - Manual memory management
  - Other low-level features

```
// A C++ program using C-style features

#include <cstdio>

int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

int main() {
    int num = 5;
    printf("Factorial of %d is %d\n",
          num, factorial(num));
    return 0;
}
```

Second Sub-Language: Object-Oriented C++

> This part of C++ is **C with classes**.

- Object-oriented programming (OOP) focuses on organizing code into **classes** and **objects** to encapsulate data and behavior.
- Key concepts:
  - **Encapsulation:** Bundling data and methods within a class.
  - **Inheritance:** Deriving new classes from existing ones.
  - **Polymorphism:** Using the same interface for different data types.

```cpp
#include <iostream>
class Rectangle {
  int width;
  int height;
public:
  Rectangle(int w, int h) :
        width(w), height(h) {}

  int area() {
    return width * height;
  }
};

int main() {
  Rectangle rect(5, 3);
  std::cout << "Area: " <<
        rect.area() << std::endl;
  return 0;
}
```

Third Sub-Language: Generic Programming with Templates

> This is the **generic programming** part of C++.

Templates enable the creation of functions and classes that can operate on different types without duplication.

- **Function Templates:** Generic functions that work with any data type.
- **Class Templates:** Generic classes that can store or operate on different types.

```cpp
#include <iostream>

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << "Sum of integers: "
          << add(3, 4) << std::endl;
    std::cout << "Sum of doubles: "
          << add(3.5, 4.5) << std::endl;
    return 0;
}
```

Fourth Sub-Language: The Standard Template Library

- The **Standard Template Library (STL)** provides a collection of **generic containers** and **algorithms**.
- Key components of the STL:
  - **Containers:** Predefined classes to store collections of data (e.g., `std::vector`).
  - **Algorithms:** Generic functions to operate on containers (e.g., `std::sort`).
  - **Iterators:** Objects to iterate over containers.

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

int main() {
  // Initialize a container
  std::vector<int> nums =
        {1, 4, 3, 9, 2};

  // Sort the vector
  std::sort(nums.begin(), nums.end());

  // Print the sorted vector
  for (int num : nums)
    std::cout << num << " ";
  std::cout << "\n";

  return 0;
}
```

C++ as a Federation of 4 Sub-Languages

- C++ is not a single, unified language but a **federation** of four sublanguages, each with its own rules and conventions.
- **Switching between sublanguages** requires a change in approach and mindset.

The C++ Standardization Process

- C++ Standardization Process
- Key Features in Major C++ Standards

Introduction to C++ Standardization

- The C++ programming language is standardized and continuously evolved by the **ISO C++ Standards Committee**.
- Major standards include **C++98**, **C++11**, and **C++17**, each marking significant language enhancements and innovations.
- Our primary reference remains CppReference.

Key Features Introduced in C++98

Released in 1998, C++98 marked the first standardized version of C++ and enhanced the C language with advanced programming constructs.

- **Classes:** Introduction of object-oriented programming principles such as encapsulation, inheritance, and polymorphism.
- **Templates:** Facilitation of generic programming for functions and classes.
- **Exception Handling:** Mechanisms (`try`, `catch`, and `throw`) for error management.
- **Standard Template Library (STL):** Provision of reusable containers, algorithms, and iterators.
- **Namespaces:** Tools to organize code and prevent naming conflicts.

Key Features Introduced in C++11

Released in 2011, C++11 introduced transformative features that enhanced performance, readability, and memory management.

- **Range-Based For Loops:** Simplified syntax for iterating over containers.
- **Move Semantics:** Optimization of resource transfers to reduce unnecessary copying.
- **Smart Pointers:** Introduction of `unique_ptr`, `shared_ptr`, and `weak_ptr` for safe dynamic memory management.
- **Lambda Expressions:** Support for inline, anonymous functions for concise coding.
- **Auto Keyword:** Automatic type deduction to improve code clarity and reduce verbosity.
- **Hash-Based Containers:** Implementation of efficient associative containers using hash tables.

Key Features Introduced in C++14

Released in 2014, C++14 refined many of the features introduced in C++11 to further enhance code readability, performance, and flexibility.

Return Type Deduction:

Functions can now automatically deduce their return types:

```
auto multiply(int a, int b) { return a * b; }
```

Generic Lambdas:

Enable lambda expressions to use the `auto` keyword for parameter types:

```
auto add = [](auto a, auto b) { return a + b; };
```

Binary Literals:

Facilitate clearer representation of binaries:

```
int binary = 0b101010;  // 42
```

Enhanced `constexpr`:

`constexpr` functions can include more complex expressions, such as recursion:

```
constexpr int square(int x) {
  return x * x;
}
```

Key Features Introduced in C++17

Released in 2017, C++17 introduced new features that streamline code, enhance functionality, and optimize performance.

- **Nested Namespaces:**

  Allow the definition of namespaces within other namespaces:

  ```
  namespace Outer::Inner {
    int value = 10;
  }
  ```

- **Class Template Argument Deduction (CTAD):**

  Automatically deduce template arguments from constructor arguments:

  ```
  std::vector vec = {1, 2, 3};  // Deduces type as std::vector<int>
  ```

More Features in C++17

- **Variable Declarations in `if` and `switch`:**

  Permit declaration and initialization of variables within `if` or `switch` statements:

  ```
  if (int x = 10; x > 5) {
    std::cout << "x is greater than 5" << std::endl;
  }
  ```

- **Structured Bindings:**

  Allow decomposition of structured objects into individual variables:

  ```
  std::tuple<int, double> data(10, 3.14);
  auto [x, y] = data;  // x = 10, y = 3.14
  ```

Timeline of Major C++ Standards

Major Standards: C++98 → C++11 → C++14/17 → C++20 → C++23 → C++26…

- **C++98:** First ISO standard, establishing foundational language features.
- **C++11:** Major update that introduced modern C++ concepts.
- **C++14/17:** Incremental enhancements and refinements.
- **C++20:** Achieved nearly all features envisioned by Bjarne Stroustrup in *The Design and Evolution of C++* (1994) (*"D&E Complete"*).
- **C++23:** Polishes and extends C++20 features.

Hello World! - `std::cout << "Hello World!"`

- C++ "Hello World" Program
- Basic Input and Output (I/O)
- Introduction to `<iostream>`

"Hello World" Example in C++

```cpp
#include <iostream>

int main() {
  std::cout << "Hello World!\n";
  return 0;
}
```

Alternatively, you can write:

```cpp
#include <iostream>

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

Basic Input and Output in C++

```cpp
#include <iostream>
int main() {
  int a, b;
  std::cin >> a >> b;
  std::cout << a + b << std::endl;
  return 0;
}
```

- `std::cin` and `std::cout` are abstractions for the standard input and output streams, respectively.
- The `<<` and `>>` operators are used for data transfer with these streams:
  - `std::cin >> x` extracts data from the input stream and stores it in `x`.
  - `std::cout << x` inserts data into the output stream, sending the value of `x` to the console.

Basic I/O: `std::cin` and `std::cout`

- **`std::cin >> x`**: This expression reads input from the standard input stream and stores the value in the variable `x`.
  - The variable `x` can be of any supported type, such as `int`, `float`, `char`, or `std::string`.
  - **C++ automatically detects the type of `x`** and selects the appropriate input method, so you do not need to use format specifiers like `"%d"` or `"%c"` as in C.
  - **C++ accesses `x` by reference**, so there is no need to use the address-of operator (`&`) to pass `x` to `std::cin`.
  - After executing `std::cin >> x`, the `std::cin` object is returned, enabling you to chain input operations, as in `std::cin >> x >> y >> z`.
- Similarly, **output** is performed using `std::cout`: `std::cout << x << y << z`.

Basic I/O: `std::endl`

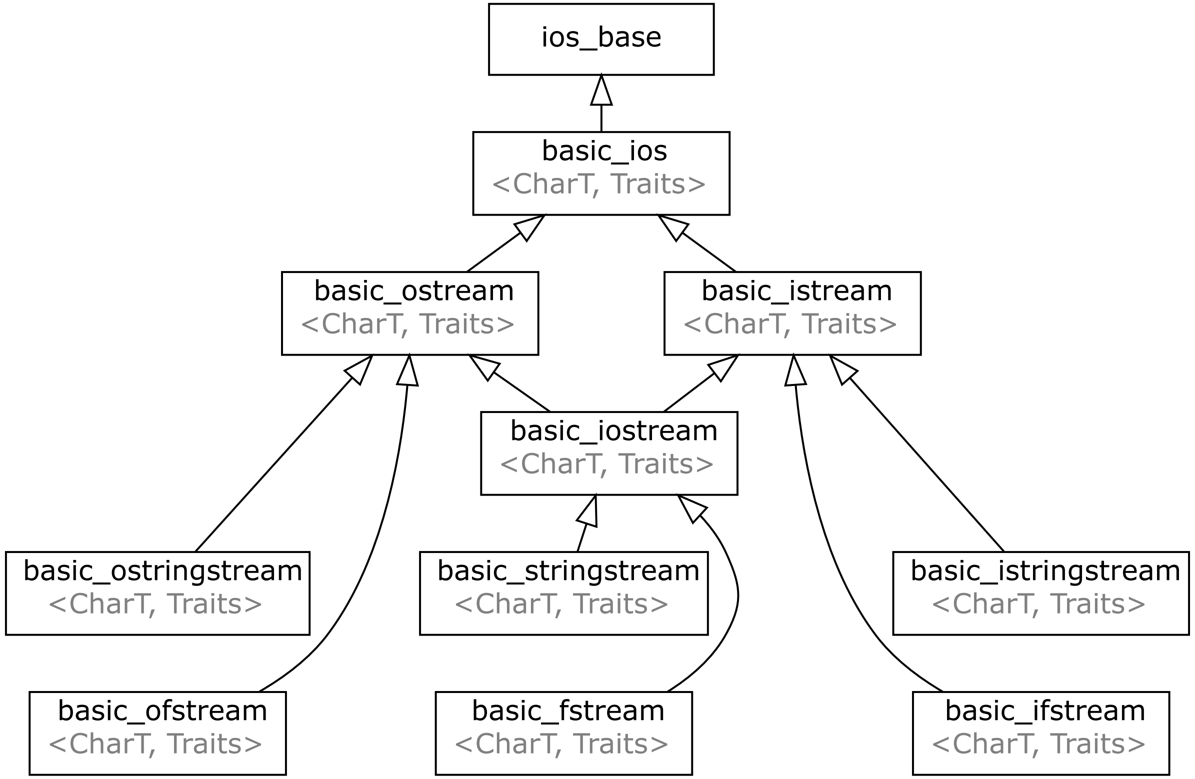
- **`std::endl`** is a manipulator used with `std::cout`.
- It outputs a newline (`\n`) and flushes the output buffer.
  - **Flushing** means clearing the output buffer and writing its contents to the screen.
  - You can also manually flush the stream using `std::cout << std::flush`. Additionally, `std::cout` may automatically flush the stream in certain cases, similar to how `stdout` works in C.

`<iostream>`: `std::cout` and `std::cin`

- `<iostream> - cppreference` is a standard library header, part of the **Input/Output Library**. Once `<iostream>` is included, **standard stream objects** such as `std::cin`, `std::cout`, and others are made available.
- `std::cout` and `std::cin` are global **objects** of the `std::ostream` and `std::istream` classes, respectively. They are associated with the standard C input and output streams, `stdin` and `stdout`.
    - `std::cin` reads from the standard C input stream `stdin`.
    - `std::cout` writes to the standard C output stream `stdout`.

`<iostream>`: Stream-based I/O

C++ provides an **[object-oriented, stream-based I/O library](#)** [(the inheritance diagram is shown below)](#) and the standard C-style I/O functions.



`<iostream>`: Operators `<<` and `>>`

Recall that operators `<<` and `>>` are originally bit-shift operators. In C++, we can customize operators function ourselves, which is called operator overloading.

- `<<`, `>>` are operators (not functions!) overloaded for the class `std::ostream` and `std::istream` respectively. Their return values are themselves, i.e., `*this`.
- You can view the two operators as if they are the functions with the following signatures:

  ```
  std::ostream& operator<<(typename value);
  ```

  ```
  std::istream& operator>>(typename value);
  ```

`<iostream>`: Manipulator `std::endl`

- Manipulators are helper ***functions*** that make it possible to control input/output streams using operator `<<` or `>>`.
- `std::endl` - CppReference is implemented as aa function that take a reference to a stream as its only argument.

  ```
  std::ostream& endl(std::ostream& os);
  ```

  Operator `<<` of class `std::ostream` has a special overload that accept pointers to this function:

  ```
  std::ostream& operator<<(std::ostream& (*func) (std::ostream&) );
  ```
- Thus we can write `std::cout << std::endl;` contiguously.

`<iostream>`: Manipulator `std::flush`

```
std::ostream& flush(std::ostream& os);
```

- `std::flush` - CppReference flushes the output buffer of the stream `os`.
- It is used to immediately output data, particularly useful when you need accurate output during debugging, to avoid losing buffered data in case of a runtime error.

Note: There are many other manipulators. See CppReference - Input/Output Manipulators for more details.

Compatibility between C and C++

- **Using the C Standard Library in C++**
- **Enhancements to the Type System**

Note: More information will be introduced in **Lecture 13**.

Using the C Standard Library in C++

The C++ standard library has everything from the C standard library, **but not exactly the same as in C**.

- The C++ version of a C standard library file `<xxx.h>` is `<cxxx>`, with all the names also introduced into `namespace std`.

Example of C-style I/O in C++

```cpp
#include <cstdio>
int main() {
  int a, b;
  std::scanf("%d%d", &a, &b);
  std::printf("%d\n", a + b);
}
```

Example: Differences in the `strchr` Function Between C and C++

C Version:

```c
#include <string.h>
int main(void) {
  const char* str = "Hello, World!";
  char* result = strchr(str, 'o');
  // Accepts const char*, but returns char*
}
```

In C, `strchr` accepts a `const char*` but returns a `char*`, which can be problematic because modifying the string through the returned pointer can lead to undefined behavior.

C++ Version:

```cpp
#include <cstring>
int main() {
  const char* str = "Hello, World!";
  const char* result = std::strchr(str, 'o');
    // Correct return type
}
```

In C++, `std::strchr` is overloaded by two versions:

- `const char* strchr(const char*, int)`
- `char* strchr(char*, int)`

Differences Between C and C++ Standard Libraries

- C exhibits several inconsistencies, primarily due to historical reasons and the need for backward compatibility. For example, the `strchr` function accepts a `const char *` but returns a `char *`, and certain entities that should be functions are implemented as macros.
- C lacks features like function overloading, which are available in C++, making some designs more complex and less flexible.
- C++ offers more advanced compile-time computation capabilities compared to C. For instance, starting with C++23, certain mathematical functions in the `<cmath>` header can be evaluated at compile time.

Use the `<cxxx>` headers (e.g., `<cstring>`, `<cmath>`) when working with the C standard library in C++ to ensure proper C++ compatibility.

Enhancements to the C++ Type System

- Improved Type Checking and Safety
- Explicit Type Conversion
- Automatic Type Deduction

Type System Enhancements in C++

- **Boolean Type (`bool`)**: In C++, `bool`, `true`, and `false` are built-in types and values, eliminating the need for `#include <stdbool.h>`. Unlike C, where `true` and `false` were often defined as `1` and `0`, in C++ they are of type `bool`.
  - This behavior has been standardized since **C23** as well.
- **Logical and Comparison Operators**: In C++, the result of logical (`&&`, `||`, `!`) and comparison (`<`, `<=`, `>`, `>=`, `==`, `!=`) operators is of type `bool`, not `int`. This makes logical operations clearer and type-safe.

```cpp
bool result = (5 > 3); // result is a boolean value
```

C++ Enhancements Over C: String Literals

> In C, string literals are **immutable** and are typically stored in **read-only memory**. Attempting to modify a string literal results in **undefined behavior**.

```
char* p = "Hello!";
p[1] = 'M'; // Undefined behavior: modifying a string literal is not allowed.

char a[] = "Hello!";
a[1] = 'M'; // Valid: 'a' is an array, not a string literal.
```

In C++, the type of a string literal (e.g., `"hello"`) is `const char[N + 1]`.

```
const char* str = "hello";
str[0] = 'H';
```

```
main.cpp:3:9: error: assignment of read-only location '* str'
```

C++ Enhancements Over C

- **Character Literals**: In C++, character literals (e.g., `'a'`) are of type `char`, not `int`, unlike in older C versions where they were promoted to `int`.
- **Compile-Time Constants**: In C++, `const` variables initialized with literals are treated as **compile-time constants** and can be used for array sizes.

```
const int maxn = 1000;
int arr[maxn]; // VLA in C
```

- **Function Declarations**: In C++, `int fun()` explicitly means the function takes **no arguments**, unlike older C standards where it could imply an unknown number of arguments.

```
int fun();  // fun takes no arguments
```

  - This change is also in **C23**, clarifying function signatures.

Improved Type Checking and Safety in C++

In C:

```c
const int x = 42;
const int* pci = &x;
int* pi = pci; // Warning
++*pi;         // Undefined behavior
char* pc = pi; // Warning
void* pv = pi;
char* pc2 = pv; // No warning
int y = pc;    // Warning
```

- Implicit type conversions, even between different pointer types, are allowed but result in warnings.
- `void *` casts are unrestricted.

In C++:

```cpp
const int x = 42;
const int* pci = &x;
int* pi = pci; // Error
++*pi;         // Error
char* pc = pi; // Error
void* pv = pi;
char* pc2 = pv; // Error
int y = pc;    // Error
```

- Dangerous implicit type conversions are compile-time errors.
- Unsafe casts require explicit casting (e.g., `static_cast`).

Explicit Type Conversion Methods in C++

- `static_cast< target-type >( expression )`
- `const_cast< target-type >( expression )`
- `reinterpret_cast< target-type >( expression )`
- `dynamic_cast< target-type >( expression )`
- `(target-type) expression`
- `target-type (expression-list)` / `target-type {expression-list}`

Using `static_cast` for Explicit Type Conversion

. It is a compile-time cast. It does things like implicit conversions between types (such as `int` to `double`, or pointer to `void*`), and it can also call explicit conversion functions.

```
int a = 5;
double b = static_cast<double>(a);  // Converts int to double
```

Using `const_cast` to Adjust Constness

`const_cast` is used to add or remove the `const` qualifier from a variable. It does not change the actual type, only the const-ness of the variable.

```
const int x = 10;
int* y = const_cast<int *>(&x);  // Removes const from pointer
```

However, **modifying a `const` variable** through a non-`const` access path (possibly created by `const_cast`) results in **undefined behavior**.

```
const int cival = 42;
const int* cref = &cival;
int* ref = const_cast<int*>(cref);  // Removes const from pointer
++ref;  // Undefined behavior
```

Using `reinterpret_cast` for Low-Level Type Casting

- Converts between types by reinterpreting the underlying bit pattern.
- Primarily used for low-level casting, especially between unrelated types. It should be used carefully, as it can result in undefined behavior, particularly when casting between different pointer types.

```cpp
int a = 65;

// Reinterpret int as a char pointer
char* p = reinterpret_cast<char*>(&a);
```

Using `dynamic_cast` for Safe Downcasting

Safely converts pointers and references to classes up, down, and sideways along the inheritance hierarchy.

```
#include <iostream>

class Base {};
class Derived : public Base {};

int main() {
  Base* basePtr = new Derived();
  Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);

  if (derivedPtr)
    std::cout << "Cast succeeded" << std::endl;
  delete basePtr;
  return 0;
}
```

C-Style Casts: `(target-type) Expression`

The **C-style** cast is a general casting mechanism that allows conversion between different types. While it is more flexible than C++-specific casts, it is less safe and may lead to unexpected behavior if not used carefully. The compiler interprets a C-style cast in the following order:

1. `const_cast<target-type>(expression)`
2. `static_cast<target-type>(expression)`
3. `static_cast` followed by `const_cast`
4. `reinterpret_cast`
5. `reinterpret_cast` followed by `const_cast`

For more details, please refer to [CppReference - Explicit Type Conversion](#).

`target-type ()/{} expression-list`

This syntax is used for **constructing objects** or **performing casts**, typically in **initialization** contexts. It offers an alternative to other casting methods and can serve two purposes:

- **Single expression**: If there is exactly one expression in the `expression-list`, the cast behaves like a C-style cast.

  ```
  double d = 3.14;
  int i = int(d);  // Equivalent to int i = (int)d;
  ```

- **Multiple expressions**: If there are multiple expressions, the syntax will be used to construct a class instance, using the provided expressions for initialization.

The distinction between parentheses `()` and curly braces `{}` will be covered later.

Automatic Type Deduction in C++

- Using the `auto` Keyword
- Class Template Argument Deduction (CTAD)
- The `decltype` Specifier

Using the `auto` Keyword for Type Deduction

The `auto` keyword allows the compiler to **deduce the type** of a variable based on its initializer.

- **Basic Usage:**

  ```
  auto x = 42;      // type is deduced as `int`
  auto y = 3.14;    // type is deduced as `double`
  auto z = x + y; // type is deduced as `double` (result of `x + y`)
  ```

  Note: You cannot use `auto` without an initializer, as the type cannot be deduced.

  ```
  auto m; // Error: initializer required.
  ```

Advanced Usage of the `auto` Keyword

- **Working with References and Pointers:**

```
auto &r = x;          // `r` is a reference to `int`
const auto &rc = r; // `rc` is a const reference to `int`
auto *p = &rc;        // `p` is a pointer to `const int`
```

- **Return Type Deduction (since C++14)**

  You can also use `auto` to deduce the return type of a function:

```
auto sum(int x, int y) {
  return x + y;  // return type is deduced as `int`
}
```

Class Template Argument Deduction (CTAD)

Class Template Argument Deduction (CTAD) allows the compiler to automatically deduce the template type based on the constructor arguments, as long as sufficient information is provided.

Examples:

```
std::vector v1{2, 3, 5, 7}; // deduced as vector<int>
std::vector v2{3.14, 6.28}; // deduced as vector<double>
std::vector v3(10, 42);     // deduced as vector<int> from the value 42 (int)
std::vector v4(10);         // Error; Insufficient information
```

Exercise 2: Figure out the Deducted Type and Output

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

int main(){
  std::vector vec(5, 42);
  for(auto i : vec)
    std::cout << i << std::endl;
}
```

- What the deducted types of `vec` and `i`?
- What will be the output?

The `decltype` Specifier for Type Deduction

`decltype(expr)` will deduce (***yield***) the type of the expression `expr` **without evaluating it**.

```
auto fun(int a, int b) { // The return type is deduced to be `int`.
  std::cout << "fun() is called." << std::endl;
  return a + b;
}
int x = 10, y = 15;
decltype(fun(x, y)) z; // Same as `int z;`.
                       // Unlike `auto`, no initializer is required here.
                       // The type is deduced from the return type of `fun`.
```

- `decltype(fun(x, y))` only deduces the return type of `fun` without actually calling it. Therefore, **no output is produced**.

CS100 Recitation 7