Discussion-1: Focus on C/C++: Starting with C++ and Security

<u>Discussion Topic:</u>

In this module, we will be introduced to working with the C++ programming language. What are some of the issues that you had installing and running the Eclipse IDE for C/C++ developers? Working with the C++ language, discuss the different data types available. What are some of the differences, if any, between data types in C++ versus Java? What tips can be utilized to identify possible vulnerabilities using C++ data types? Be sure to provide an appropriate source code example to illustrate your points.

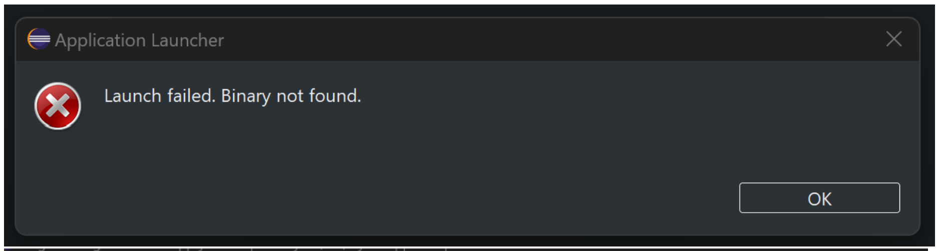
<u>My Post:</u>

Hello Class,

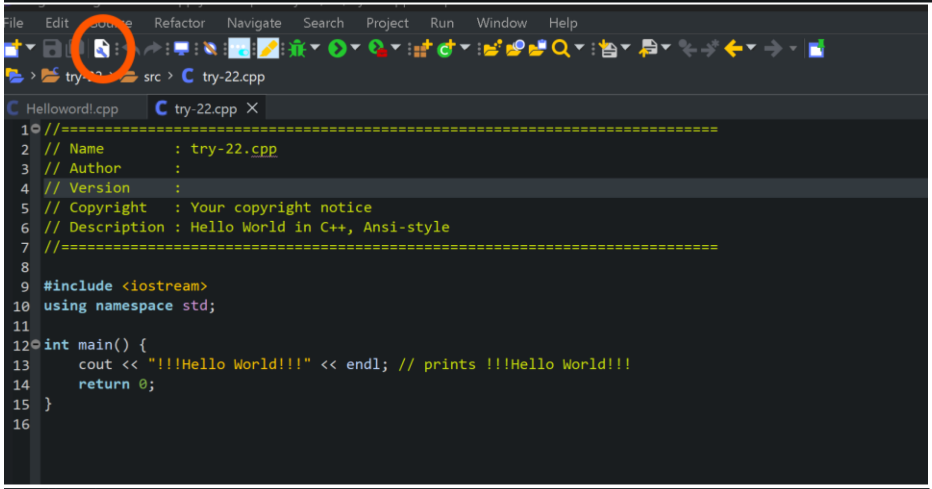
I use Microsoft <u>Visual Studio 2022</u> as my IDE for C++. I have tried different IDEs, and Visual Studio 2022 is the one that gives me the least grief when importing libraries. However, I decided to try Eclipse for C++. Having Eclipse for Java already installed on my Windows system I looked for a tutorial on how to install C++ in already existing Eclipse for Java installations and found the following video: <u>How to Setup Eclipse IDE for C/C++ Development in Easy steps</u>. I had no problem with my installation. However, the first time you try to run your program, do not forget to build it, as I did (forgot), if not you will get a 'Launch failed. Binary not found' error, see Pictures. Visual Studio 2022 built your binary file automatically for you… And if you listen to the video tutorial, it tells you to do it.

Pictures:

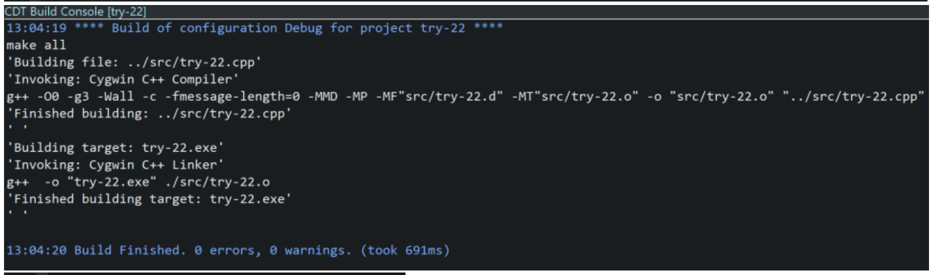
Error Launch failed. Binary not found



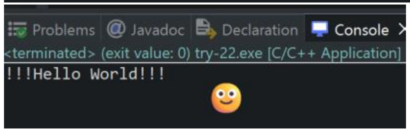
Launch failed. Binary not found.



```
//============================================================================
// Name        : try-22.cpp
// Author      :
// Version     :
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//============================================================================

#include <iostream>
using namespace std;

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



```
13:04:19 **** Build of configuration Debug for project try-22 ****
make all
'Building file: ../src/try-22.cpp'
'Invoking: Cygwin C++ Compiler'
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"src/try-22.d" -MT"src/try-22.o" -o "src/try-22.o" "../src/try-22.cpp"
'Finished building: ../src/try-22.cpp'
' '
'Building target: try-22.exe'
'Invoking: Cygwin C++ Linker'
g++  -o "try-22.exe" ./src/try-22.o
'Finished building target: try-22.exe'
' '

13:04:20 Build Finished. 0 errors, 0 warnings. (took 691ms)
```



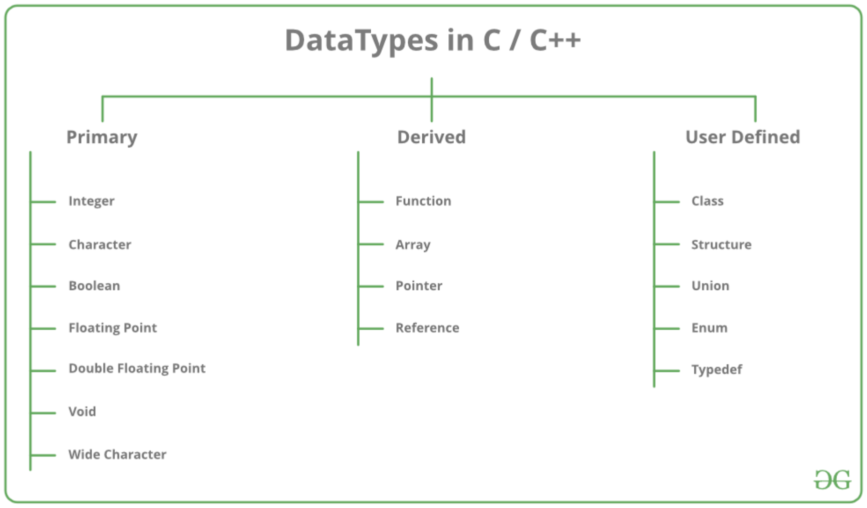
```
<terminated> (exit value: 0) try-22.exe [C/C++ Application]
!!!Hello World!!!
```

For this class, I probably will use both IDEs, depending on what I am trying to accomplish.

C++ Data Types

In C++, there are broadly three types of data: Primitive (or Built-in) Data Types, Derived Data Types, and User-defined Data Types, see Figure 1:

Figure1
C/C++ Data Types

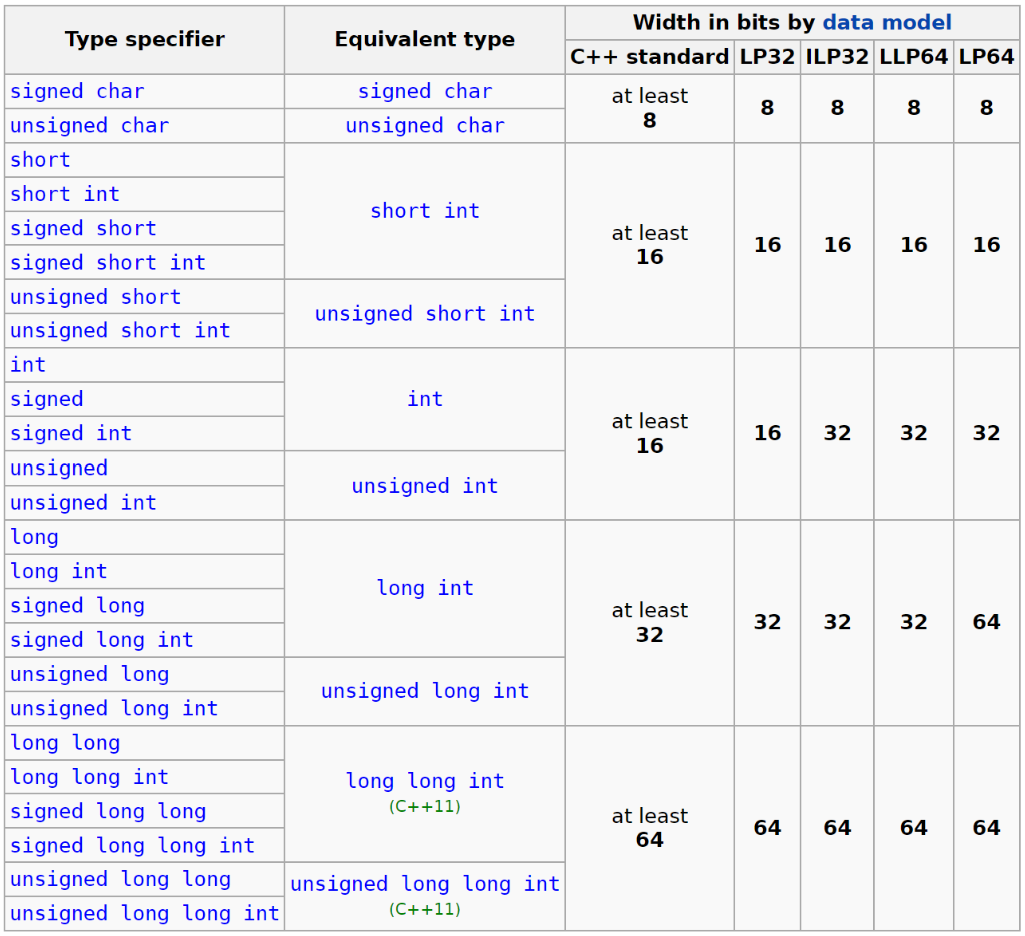


Note: From "C++ Data Types" by Argarwal (2023)

- The Primitive types also refer as fundamental types are the basic data types that are predefined in C++, see below for a list of them:

- **Integer (int)**: Stores whole numbers, usually require 4 bytes of memory, and have a range from -2,147,483,648 to 2,147,483,647 (Argarwal, 2023)

  Table 1
  Integers



Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
`signed char`	`signed char`	at least 8	8	8	8	8
`unsigned char`	`unsigned char`					
`short`	`short int`	at least 16	16	16	16	16
`short int`						
`signed short`						
`signed short int`						
`unsigned short`	`unsigned short int`					
`unsigned short int`						
`int`	`int`	at least 16	16	32	32	32
`signed`						
`signed int`						
`unsigned`	`unsigned int`					
`unsigned int`						
`long`	`long int`	at least 32	32	32	32	64
`long int`						
`signed long`						
`signed long int`						
`unsigned long`	`unsigned long int`					
`unsigned long int`						
`long long`	`long long int` (C++11)	at least 64	64	64	64	64
`long long int`						
`signed long long`						
`signed long long int`						
`unsigned long long`	`unsigned long long int` (C++11)					
`unsigned long long int`						

Note: From “Fundamental Types” by C++ Reference (n.d.)

- **Character (char)**: Stores character, require, 1 byte, and has a range of -128 to 127 (or 0 to 255 for unsigned char). Note they are technically positive integers storing characters as integers using their underlying ASCII codes
  Wide Character (wchar_t) stores wide characters, usually requiring 2 or 4 bytes depending on the platform.
- **Boolean (bool)**: Stores logical values, true or false, usually requires 1 byte. They are technically positive integers, where O is true all other positive integers is false.
- **Floating Point (float)**: Stores single-precision floating-point numbers, usually requiring 4 bytes.
- Double (double): Stores double-precision floating-point numbers, requiring 8 bytes
  long double — extended precision floating-point type. Does not necessarily map to types mandated by IEEE-754. (C++ Reference , n.d., p.1)
- **Void (void)**: type with an empty set of values or absence of value, often used in functions that do not return any value.
  “ type with an empty set of values. It is an incomplete type that cannot be completed (consequently, objects of type void are disallowed). There are no arrays of void,

nor references to void. However, pointers to void and functions returning
type void (*procedures* in other languages) are permitted." (C++ Reference , n.d., p.1)

- Derived data types are derived from primitive data types, and they include:

  - Array: collection of elements of the same type.
  - Pointer: memory addresses of variables.
  - Function: block of code that performs a specific task.
  - Reference: alias to another variable.

  (Argarwal, 2023)

- User-defined Data Types, as the name says are defined by the user

- Class: stores variables and functions into a single unit.
- Structure (struct): Similar to classes but with public default access.
- Union: Stores different data types in the same memory location.
- Enumeration (enum): Sets of names associated with an integer constant, uses extensively in the video game industry.
- Typedef: Allown to create or assign new names for existing data types.

Main Differences Between Models: C++ vs Java

Before listing the differences between the C++ and Java data types, I would like to discuss the main differences and similarities between the two languages.

Both languages are Object Oriented Programming (OOP) languages. However, However, C++ is platform-dependent, while Java is platform-independent due to Java's Virtual Machine (JVM) component that compiles Java to bytecode using an interpreter, which can run on any system with a JVM, this is referred to as "Write Once, Run Anywhere" capability (Eck, 2015). Additionally, C++ supports both procedural and object-oriented programming, whereas Java strictly OPP, making C++ more suitable for programming operating systems. Furthermore, Java, being strictly object-oriented, lacks support for global variables or free functions (as C++ does). Java encapsulates everything within classes.

Data Types Differences Between C++ and Java

Below is a list of data type differences between C++ and Java.

- Both languages have primitive data types; however, C++ supports both signed and unsigned types, making C++ primitive data types more flexible but also less secure when unsigned. Java primitive data types are signed by default (Eck, 2015).
- Java primitive data types have fixed sizes and are platform-independent, whereas C++ primitive data types storage sizes are platform-dependent.
- C++ provides consistency between primitive and object types. Java also distinguishes between primitives and objects; however, it has "wrapper classes" for primitive types like Integer for int, allowing them to have object-like behavior (Sruthy, 2024).
- Java does not support structure or union data types, whereas C++ does.
- C++ supports pointers and reference types, whereas Java has very limited support. This is by choice and for security reasons.

- C++ variables have a global scope as well as namespace scope. On the other hand, Java variables have no global scope; however, they can have package scope.
- Java supports documentation comments 'Javadocs', whereas C++ does not.

Possible Vulnerabilities When Using C++ Data Types

If not handled right C++ data types may create security issues. Thus, it is crucial to understand and identify C++ data type vulnerabilities. Below is a list of the most common vulnerabilities that may arise with handling C++ data types and types on how to mitigate them.

1. **Buffer Overflows**
   Buffer overflow occurs when the data written in memory surpasses a buffer-allocated capacity. This can lead to program crashes or malicious code execution.
   To prevent buffer overflow, it is important to check bounds before coding, and use library functions resistant to buffer overflow like "fgets." Refrain from using "scanf," "strcpy," "printf," "get," and "strcaf," which are prone to buffer overflows (GuardRails, 2023). See example below.

```
#include <iostream>
#include <cstring>

int main() {
    char buffer[10];
    std::cout << "Enter a string: ";
    fgets(buffer, sizeof(buffer), stdin);  // Avoids buffer overflow by limiting input
    std::cout << "You entered: " << buffer << std::endl;
    return 0;
}
```

2. **Integer Overflow and Underflow**
   Integer overflow occurs when a trying to store in an integer variable exceeds the maximum value an integer can hold (Snyk Security Research Team, 2022). Integer underflow occurs when a value becomes less than the minimum value an integer can hold.

To prevent integer overflow and underflow always validate the values before performing arithmetic operations, see code example below:

```
#include <iostream>
#include <climits>

int add(int a, int b) {
    if (a > 0 && b > INT_MAX - a) {
        std::cerr << "Integer overflow detected!" << std::endl;
        return -1;
    }
    return a + b;
}
```

```cpp
int main() {
    int x = INT_MAX - 1;
    int y = 2;
    std::cout << "Result: " << add(x, y) << std::endl;
    return 0;
}
```

Note that before adding a and b, in the add function, the condition of adding them would cause an overflow is checked, here the sum is greater than the integer max value allowed.

3. **Incorrect Type Conversion**
   Incorrect type conversions often happen when converting a signed integer to an unsigned integer leading to unintentional data loss. This can be avoided by Not-implicitly converting data type, for example, avoid casting double to float or a signed to an unsigned type. See code example below for example:

```cpp
#include <iostream>

void checkLength(int len) {
    if (len < 0) {
        std::cerr << "Negative length detected!" << std::endl;
        return;
    }
    std::cout << "Length is: " << len << std::endl;
}

int main() {
    // Incorrect implicit conversion
    unsigned int value = -5;  // unsigned int are always non-negative
    checkLength(static_cast<int>(value));  // Properly handle conversion
    return 0;
}
```

4. **Pointer initialization**
   Null pointers are a well-known vulnerability that can lead to program crashing and potential nefarious exploitation. Always check pointers for ‘`nullptr`’ before dereferencing them. See code below for example.

```cpp
#include <iostream>

void processPointer(int* ptr) {
    if (ptr == nullptr) {
        std::cerr << "Pointer is null!" << std::endl;
        return;
    }
    std::cout << "Pointer value: " << *ptr << std::endl;
}

int main() {
    int* p = nullptr;
    processPointer(p);  // Properly handle null pointers
    return 0;
}
```

5. **Void Pointer**
   In C and C++ '`void *`' is often used to represent a polymorphic data structures, aka generic types (Adam, & Kell, 2020). It is '`nullptr`' without a type, similarly, it can lead to program crashing and potential nefarious exploitation. Always cast it to a type before using, see code below for an example: the following is in C code.

```
struct list {
    void *p;
    struct list *next;
};

int f(struct list *l) {
    int max = -1;
    while (l) {
        int i = *((int*)l->p); // Cast void* to int*
        max = i > max ? i : max;
        l = l->next;
    }
    return max;
}
```

Note that the `void *`' is used to store a generic pointer in the linked list and it is cast to '`int*`' before being used.

To summarize, understanding C++ data types, including their vulnerabilities, is essential for writing secure C++ code. This can be done by managing data types properly, performing bounds checks, validating type conversions, and using pointers cautiously by checking for null pointers and always casting generic pointers before use.

-Alex

References:

Adam, J. & Kell, S. (2020). Type checking beyond type checkers, via slice & run. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis (TAPAS 2020). Association for Computing Machinery,* 23–29. Retrieved from: https://dl-acm-org.csuglobal.idm.oclc.org/doi/10.1145/3427764.3428324

Argarwal, H. (2023, September 23). C++ data types. GoogsforGeeks. https://www.geeksforgeeks.org/cpp-data-types/

C++ Reference (n.d.). *Fundamental types*. cppreference.com. https://en.cppreference.com/w/cpp/language/types

Eck, D. J. (2015). Chapter 1 Overview: The mental landscape. *Introduction to programming using Java* (7th ed.). CC BY-NC-SA 3.0. http://math.hws.edu/javanotes/

GuardRails, (2023, April 27). The Top C++ security vulnerabilities and how to mitigate them. Security Boulevard. https://securityboulevard.com/2023/04/the-top-c-security-vulnerabilities-and-how-to-mitigate-them/

Snyk Security Research Team (2022, August 16). Snyk.https://snyk.io/blog/top-5-c-security-risks/

Sruthy, (2024, March 7). C++ vs Java: Top 30 differences between C++ and Java with examples. Software Testing Help. https://www.softwaretestinghelp.com/cpp-vs-java/