Data Structures and Algorithms

Chapter 0: C++ Basic Syntax

Basic Structure

```
1  #include <iostream>
2
3  int main()
4  {
5   std::cout << "Hello world!";
6   return 0;
7  }
```

Line 1 is called a preprocessor directive that indicates that you intend to use the functionality of the iostream library. This **input/output library** (io library) is part of the C++ standard library that allows us to get input from the keyboard and output data to the console.

Line 3 is a function called ***main***. The main function is the entry point of a C++ program. It is where the execution of the program begins.

Line 4 and 7 indicates the ***function body*** of *main.* All statements between the opening and closing curly brace are part of the function *main.*

Line 5 is a statement that will execute inside the *main* function. The ***std::cout*** (character output) followed by the << operator is used to display an output on the console. In this case, that output is "Hello world!".

Line 6 is a ***return*** statement. For the **main(),** in order to let the operating system know whether or not an executable program ran successfully, the program sends a value back to the system when it has finished running. A status code of `0` means the program executed as expected.

Comments

In programming, comments are used to explain code and make it more readable. They can also be used to prevent the execution of certain code when testing alternative code. Comments are used exclusively by programmers; the compiler ignores them which means it won't affect the program execution. There are two types of comments in C++: single-line comments and multi-line comments.

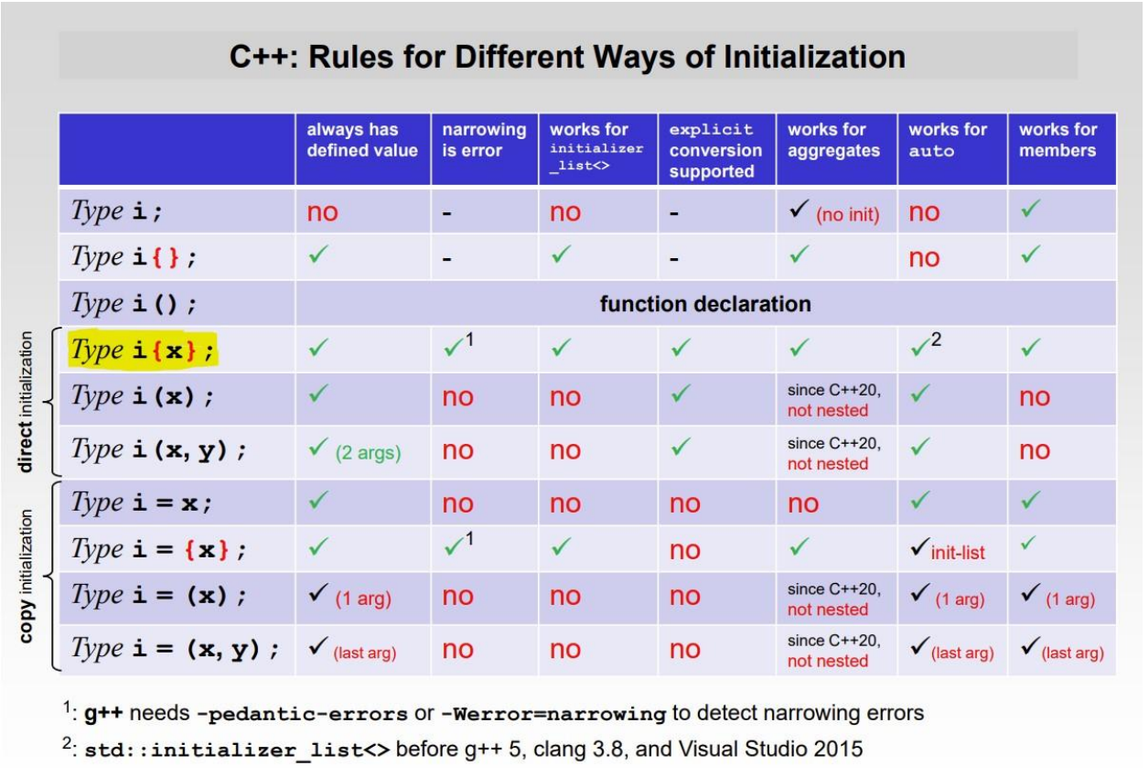
```
1  std::cout << "Hello world!"; // This is a single-line comment
2  /* This is a multi-line comment
3     that can span multiple lines.
4     This line will be ignored by the compiler*
```

Variable initialization

List initialization, also known as uniform initialization or brace initialization, is the current method for initializing objects in C++ (since C++11). It uses curly brackets. **It is recommended to explicitly initialize your variables** (Like in Line 1 and Line 3)**.**

```
1  int length{ 10 };    // direct list initialization of value 10 into variable length
2  int x{};    // value initialization will initialize the variable x with value 0
3  int a{ 0 };    // explicit initialization to value 0
```

Note: *There are different ways to initialize a variable in C++.* ***We will use*** **list initialization** (highlighted yellow in the following image) ***throughout the semester.***



C++: Rules for Different Ways of Initialization

		always has defined value	narrowing is error	works for `initializer_list<>`	`explicit` conversion supported	works for aggregates	works for `auto`	works for members
	Type `i;`	no	-	no	-	✓ (no init)	no	✓
	Type `i{};`	✓	-	✓	-	✓	no	✓
	Type `i();`	function declaration						
direct initialization	*Type* `i{x};`	✓	✓[1]	✓	✓	✓	✓[2]	✓
direct initialization	*Type* `i(x);`	✓	no	no	✓	since C++20, not nested	✓	no
direct initialization	*Type* `i(x, y);`	✓ (2 args)	no	no	✓	since C++20, not nested	✓	no
copy initialization	*Type* `i = x;`	✓	no	no	no	no	✓	✓
copy initialization	*Type* `i = {x};`	✓	✓[1]	✓	no	✓	✓ init-list	✓
copy initialization	*Type* `i = (x);`	✓ (1 arg)	no	no	no	since C++20, not nested	✓ (1 arg)	✓ (1 arg)
copy initialization	*Type* `i = (x, y);`	✓ (last arg)	no	no	no	since C++20, not nested	✓ (last arg)	✓ (last arg)

[1]: **g++** needs **`-pedantic-errors`** or **`-Werror=narrowing`** to detect narrowing errors

[2]: **`std::initializer_list<>`** before g++ 5, clang 3.8, and Visual Studio 2015

std::cout, along with the **insertion operator** << allows us to display data to the console. *cout* stands for "character output".

```
std::cout << "Hello world!"; // print Hello world! to console
```

std::cin reads input from the keyboard using the **extraction operator** >>. *cin* stands for "character input". To use the input, it must be stored in a variable.

```
#include <iostream>  // for std::cout and std::cin

int main()
{
    std::cout << "Enter a number: "; // ask user for a number

    int num{}; // define variable num to hold user input
    std::cin >> num; // get number from keyboard and store it in variable num

    std::cout << "You entered " << num << '\n';
    return 0;
}
```

Escape sequence or newline character '\n'

Escape sequences are characters in C++ that have special meaning. An escape sequence starts with a **'\' (backslash),** followed by a letter or number. **'\n'** is one example of an escape character and it is used to print a newline.

```
#include <iostream>

int main()
{
      std::cout << "First Line " << '\n'; // standalone \n between single quotes
      std::cout << "Second Line\nThird Line\n";// \n can be embedded in double quotes

      return 0;
}
```

Operators

In C++, there are several basic operators that you can use for various operations. Here is a breakdown of some of the most commonly used operators in C++:

1. **Arithmetic** Operators:
    - Addition (+): Adds two operands together.

- Subtraction (-): Subtracts the right operand from the left operand.
- Multiplication (*): Multiplies two operands together.
- Division (/): Divides the left operand by the right operand.
- Modulo (%): Returns the remainder of the division between the left operand and the right operand. Example:

```
int a{ 5 }; int b{ 3 };
int sum{ a + b };
int difference{ a - b };
int product{ a * b };
int quotient{ a / b };
int remainder{ a % b };
```

*Note that when performing **integer division** using the division operator /, the result is the quotient of the division. **The fractional part of the division is discarded**, and the result is an integer value. If the dividend is not exactly divisible by the divisor, the division operator returns only the quotient, and the remainder is discarded.*

2. **Assignment** Operators:
   - Assignment (=): Assigns the value on the right to the variable on the left.
   - Addition assignment (+=): Adds the value on the right to the variable on the left and assigns the result to the variable on the left.
   - Subtraction assignment (-=): Subtracts the value on the right from the variable on the left and assigns the result to the variable on the left.
   - Multiplication assignment (*=): Multiplies the variable on the left by the value on the right and assigns the result to the variable on the left.
   - Division assignment (/=): Divides the variable on the left by the value on the right and assigns the result to the variable on the left. Example:

```
int a{ 0 };   // variable initialization
a = 1;      // variable assignment
a += 35;     // equivalent to a = a + 35
a -= 10;    // equivalent to a = a - 10
a *= 100;   // equivalent to a = a * 100
a /= 50;   // equivalent to a = a / 50
```

3. **Relational** Operators:
   - Equal to (==): Checks if the left operand is equal to the right operand.
   - Not equal to (!=): Checks if the left operand is not equal to the right operand.
   - Greater than (>): Checks if the left operand is greater than the right operand.
   - Less than (<): Checks if the left operand is less than the right operand.
   - Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.
   - Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.

   Example:

```
int a{ 5 };
int b{ 3 };
bool isEqual{ a == b };      // isEqual will be false
bool isNotEqual{ a != b };  // isNotEqual will be true
bool isGreater{ a > b };    // isGreater will be true
 bool isLess{ a < b };          // isLess will be false
bool isGreaterOrEqual{ a >= b };  // isGreaterOrEqual will be true
bool isLessOrEqual{ a >= b };     // isLessOrEqual will be false
```

4. **Logical** Operators:
   - Logical AND (&&): Returns true if both operands are true.
   - Logical OR (||): Returns true if either of the operands is true.
   - Logical NOT (!): Returns the opposite of the operand.

Example:

```
bool a{ true };
bool b{ false };
bool aAndB{ a && b };      // aAndB will be false
bool aOrB{ a || b };      // aOrB will be true
bool notA{ !a };          // notA will be false
```

These are just a few examples of the basic operators in C++. There are also other operators like bitwise operators, ternary operator, comma operator, and more, which you can explore further.

Function

A **function** is a reusable sequence of statements designed to do a specific task.

```
returnType functionName(dataType1 parameter1, dataType2 parameter2, ..) // This is the function header
{
      // This is the function body
}
```

returnType: This specifies the type of value that the function will return. If the function does not return a value, the return type should be **void**.

functionName: This is the name of the function, which you can choose. It should be a descriptive name that indicates what the function does.

dataType1 parameter1, dataType2 parameter2, ...: These are the parameters that the function can accept. Parameters are optional and can be of any valid data type in C++. They allow you to pass values into the function for it to work with.

function body: This is the block of code that defines what the function should do. It contains the statements and expressions that make up the function's logic.

Here's an example of a function that returns an integer value

```
int getValueFromUser()
{
      std::cout << "Enter an integer: ";   int input{};
        std::cin >> input;

      return input; // return the value back to the caller
}
```

To call a function in C++, you simply write the function's name followed by parentheses ()

```
int x{ getValueFromUser() }; // call the function getValueFromUser and initialize the variable x with the returned value
```

When calling a function in C++, there are a few requirements to keep in mind. Here are the requirements for calling a function:

1. The function must be declared or defined before it is called. This means that the function prototype or definition should appear before the point of function call in the code. If the function is defined in a separate source file, you can use a forward declaration to declare the function before calling it.
2. The function name must be spelled correctly. The function name is case-sensitive, so make sure to use the correct capitalization and spelling.
3. The function must have the correct number and type of arguments. When calling a function, you need to provide the correct number of arguments and ensure that the types of the arguments match the function's parameter types. If the function has default arguments, you can omit those arguments when calling the function.
4. The function must have the correct return type. If the function returns a value, make sure to assign the return value to a variable or use it in an expression. If the function has a void return type, you can simply call the function without assigning the return value.
5. If the function is a member function of a class, you need to call the function on an instance of that class or on a pointer/reference to an instance of that class.

Example:

```cpp
#include <iostream>

// Function definition
int add(int a,int b) {
    return a + b;
}

int main() {
    std::cout << "10 plus 15 equals " << add(10, 15) << '\n';
   return 0;
}
```

Forward declarations:

A forward declaration allows you to declare a function without providing its definition. This is useful in situations where a function needs to be referenced before it is defined, such as when functions are mutually dependent on each other. *Note the difference between **declaration** and **definition**.*

The same example as above but using forward declaration:

```cpp
#include <iostream>

// Function declaration int
add(int a, int b);

int main()
{
     std::cout << "10 plus 15 equals " << add(10, 15) << '\n';
}

// Function definition int
add(int a, int b) {
    return a + b;

}
```

It is important to note that forward declarations are not always necessary. They are typically used in header files to declare functions that are defined in separate source files. By forward declaring the functions, you can include the header file in multiple source files without causing conflicts. (*Header files will be discussed later in the semester, but you may study about them in advance*).

However, whether you're using headers or not, forward declarations can be used to define functions in any order, maximizing code organization and reader understanding. This is particularly useful when you have related functions that you want to cluster together or when you have functions that call each other. Forward declarations allow you to resolve circular dependencies and define functions in a logical order.

Void functions

In C++, **void functions**, also known as non-value returning functions, are a type of function that **does not return a value**. The keyword 'void' is used to specify that a function doesn't have a return value

```cpp
#include <iostream>

// This function does not return a value, so no return statement is needed
void printHello()
{
    std::cout << "Hello\n";
}
int main()
{
    printHello(); // function printHello() is called, no value is returned
    return 0;
}
```

It's important to note that void functions can't be used in expressions that require a value

```cpp
#include <iostream>

// void means the function does not return a value to the caller
void printHello()
{
   std::cout << "Hello" << '\n';
}
int main()
{
    printHello(); // function printHello() is called, no value is returned
    std::cout << printHello(); // compile error
    return 0;
}
```

Conditional statements

A conditional statement determines whether or not one or more related statements should be executed.

The most basic kind of conditional statement in C++ is the ***if statement.***

```cpp
if (condition) {
      // code to be executed if the condition is true
}
```

Optional else statement:

```cpp
if (condition) {
    // code to be executed if the condition is true
} else
{
    // code to be executed if the condition is false
}
```

If else with multiple conditional statements

```cpp
if (condition1) {
    // code to be executed if condition1 is true
}
else if (condition2) {
    // code to be executed if condition1 is false and condition2 is true
} else
{
    // code to be executed if both condition1 and condition2 are false
}
```

Example:

```cpp
#include <iostream>
int main()
{
   std::cout << "Enter an integer:";
   int num{ 0 };
   std::cin >> num;
   if (num < 0) {
      std::cout << num << " is negative\n";
   } else if (num <= 100) { // only executes if x >= 0
      std::cout << num << " is between 0 and 100\n";
   } else {// only executes if x > 100
      std::cout << num << " is greater than 100\n";
   }
   return 0;
}
```

Switch statement

It is standard practice to compare a variable or expression's equality against a set of different values, hence C++ has an alternative conditional statement called a **switch statement** that is tailored specifically for this use.

Switch statement with return

```
#include <iostream>

float calculate(float num1, float num2, char op)
{
    switch (op)
    {
        case '+':
          return num1 + num2;
        case '-':
          return num1 - num2;
        case '*':
          return num1 * num2;
        case '/':
            if (num2 != 0) { // num2 is not equal to zero
              return num1 * num2;
            } else {
              std::cout << "Error! Division by zero is not allowed.\n";
              return 0;
            }
        default: // op is not equal to any of the previous cases
            std::cout << "Error! Unknown operator.\n";
            return 0;
     }
}
int main()
{
      float num1{ 0 };  float
num2{ 0 };
      std::cout << "Enter first number: ";
      std::cin >> num1;
      std::cout << "Enter second number: ";
      std::cin >> num2;
      std::cout << "Enter operator";

      char op{};
      std::cin >> op;
      float result{ calculate(num1, num2, op) };
      std::cout << num1 << ' ' << op << ' ' << num2 << " is equal to " << result << '\n';

      return 0;
}
```

Note that using return inside a switch statement will exit the entire function, not just the switch statement This is different from break, which only exits the switch statement.

Switch statement with break

```
#include <iostream>

int main()
{
  int a{ 2 };
  switch (a)
  {
      case 1:
          std::cout << "a is 1\n";
          break;
      case 2:
          std::cout << "a is 2\n";
          break;
      default: // a is not equal to any of the previous cases
          std::cout << "x is not 1 or 2\n";
          break;
  }
      std::cout << "This is a statement after the switch.\n";
  return 0;
}
```

Loops

In C++, loops are used to **execute a block of code repeatedly until a particular condition is satisfied**.
Loops help to avoid redundancy in code by reducing the need to write the same set of lines multiple times. They are mainly used to enhance the efficiency and sophistication of programs by executing the same function multiple times. In C++, there are three types of loops: **for loop**, **while loop**, and **do-while loop.**

For loop

The for loop is an entry-controlled loop that allows us to **execute a block of code a specific number of time**s. It consists of three parts: initialization, test condition, and update expression. Here's the syntax:

```
for (initialization; test condition; update expression) {
    // code to be executed
}
```

Example:

```
#include <iostream>

int main()
{
      for (int i{ 0 }; i < 5; ++i) {
            std::cout << "Hello World\n";
       }
       return 0;
}
```

While loop

The while loop is a pre-test loop that repeatedly executes a block of code as long as a given condition is true. It checks the condition before entering the loop body. Here's the syntax:

```
while (condition) {
      // code to be executed }
```

Example:

```
#include <iostream>

int main()
{
   int i{ 0 };
   while (i < 5) {
       std::cout << "Hello World\n";
       ++i;
   }
   return 0;
}
```

Note that if the condition initially evaluates to false, the associated statement will not execute at all. On the other hand, if the expression always evaluates to true, the while loop will execute forever. This is called an **infinite loop.**

Do-While loop

The do-while loop is a post-test loop that executes a block of code at least once, and then repeatedly executes it as long as a given condition is true. It checks the condition after executing the loop body. Here's the syntax:

```
do {
    // code to be executed } while (condition);
```

Example:

```
#include <iostream>

int main()
{
   int i{ 0 };
   do {
       std::cout << "Hello World\n";
       ++i;
   } while (i < 5);
   return 0;
}
```

Note: Do-while loops aren't common in real-world applications. It is difficult to see the loop condition when it is near the bottom, which can lead to mistakes**. Many developers recommend avoiding do-while loops unless you really need to run a statement at least once.**

An example of practical use for do-while loop is when prompting a user for an input until it's a valid input.

```cpp
#include <iostream>

int main()
{
  int choice{ 0 };
  do {
      std::cout << "Menu:\n";
      std::cout << "1. Option 1\n";

      std::cout << "2. Option 2\n";

      std::cout << "3. Exit\n";
      std::cout << "Enter the number of your choice: ";
      std::cin >> choice;

       // Perform actions based on the choice
      switch (choice) {
          case 1:
            std::cout << "Option 1 was selected\n";
            // some code block for option 1
            break;
          case 2:
            std::cout << "Option 2 was selected\n";
            // some code block for option 2
      break;
          case 3:
            // Exit the program
            break;
          default: // choice is not equal to any of the previous cases

            std::cout << "Invalid choice. Please try again.\n";
            break;
      }
    } while (choice != 3);
   return 0;
}
```

Struct and Classes

In C++, structs and classes are two distinct ways to define custom data types. While both can be used to represent data structures, they differ in terms of their purpose, syntax, and behavior.

Similarities of Struct and Class:

1. **Data Members and Member Functions**:
   Both struct and class can contain data members (variables) and member functions (methods). This means you can define functions inside both struct and class to operate on their data.
2. **Inheritance**:
   Both struct and class can be used as base classes, allowing other struct or class types to inherit from them. This supports the object-oriented programming principle of inheritance, enabling code reuse and the creation of hierarchies. In addition, bases of a struct are inherited publicly by default, whereas bases of a class are inherited privately by default.
3. **Access Specifiers**:
   Both struct and class can use the access specifiers public, private, and protected to control the visibility and accessibility of their members. This allows for encapsulation, ensuring that internal details are hidden, and only exposed interfaces are accessible.

Differences between Struct and Class:

1. **Default Access Modifier:**
   - struct: By default, all members of a struct are public. This means that if you do not specify an access modifier, the data members and member functions are accessible from outside the struct.

- class: By default, all members of a class are private. This means that if you do not specify an access modifier, the data members and member functions are not accessible from outside the class.

2. **Intended Use and Common Practices:**
   - struct: Typically used for simple data structures that primarily contain public data members and few, if any, member functions. struct is often chosen when the primary purpose is to group related data together without needing the encapsulation provided by class.
   - class: Used for more complex structures where encapsulation, data hiding, and member functions are needed. class is generally preferred when you need to implement behavior (methods) and enforce access control.

Example Struct:

```cpp
#include <iostream>

struct Point {
    int x;
    int y;

    // Constructor
    Point(int xCoord, int yCoord) : x(xCoord), y(yCoord) {}

    // Member function to display the point
    void display() const {
        std::cout << "Point(" << x << ", " << y << ")" << '\n';
    }
};

int main() {
    // Create a Point object
    Point p(10, 20);

    // Access and modify public members directly
    p.x = 15;
    p.y = 25;

    // Display the point
    p.display();

    return 0;
}
```

Example Class:

```cpp
#include <iostream>

class Rectangle {
private:
    int width;
    int height;

public:
    // Constructor
    Rectangle(int w, int h) : width(w), height(h) {}

    // Public member function to set the width
    void setWidth(int w) {
        if (w > 0) {
            width = w;
        }
    }

    // Public member function to set the height
    void setHeight(int h) {
        if (h > 0) {
            height = h;
        }
    }

    // Public member function to get the area
    int getArea() const {
        return width * height;
    }

    // Public member function to display the rectangle dimensions
    void display() const {
        std::cout << "Rectangle(width: " << width << ", height: " << height << ")" << '\n';
    }
```

```
};

int main() {
    // Create a Rectangle object
    Rectangle rect(10, 20);

    // Modify dimensions using public member functions
    rect.setWidth(15);
    rect.setHeight(25);

    // Display the rectangle
    rect.display();

    // Display the area of the rectangle
    std::cout << "Area: " << rect.getArea() << '\n';

    return 0;
}
```

Namespaces and directives

The concept of namespaces in C++ is used to organize code into logical groups and prevent naming collisions. Namespaces provide a way to group related classes, functions, and variables together under a unique identifier.

- The scope resolution operator :: is used to access identifiers defined within a namespace
- Explicit namespace prefixes should be used to access identifiers defined in a namespace
- A ***using directive*** statement can be used to access names in a namespace without using a namespace prefix
- User-defined namespaces are namespaces defined by the programmer
- Namespaces provided by C++ (such as the global namespace) or libraries (such as namespace std) are not considered user-defined namespaces
- Using a using directive at the top of the program is generally discouraged

Using a directive like the **using namespace std** is generally considered **bad practice** in C++.

Why?

```
#include <iostream> // imports the declaration of std::cout into the global scope

using namespace std; // makes std::cout accessible as "cout"

int cout() // defines our own "cout" function in the global namespace
{
    return 5;
}

int main()
{
    cout << "Hello, world!"; // Compile error!  Which cout do we want here?  The one in the
std namespace or the one we defined above?

    return 0;
}
```

- Name conflicts: The std namespace contains **a large number** of names, including common ones like cout, cin, and string. By using the using namespace std directive, all the names in the std namespace are brought into the global namespace. **This increases the chance of name conflicts if you have your own functions or variables with the same names**. For example, if you have a function named count and you use using namespace std, it will conflict with std::count function.
- Readability: **Explicitly qualifying names with the std:: prefix makes the code more readable** and self-explanatory. It helps to clearly identify the origin of a particular function or object. This is especially important in larger codebases or when collaborating with other developers.

- Maintenance: If you use using namespace std in a header file, it can potentially pollute the global namespace for all files that include that header. **This can lead to subtle bugs and make it harder to maintain and debug the code.**

Instead of using using namespace std, it is recommended to use the std:: prefix to explicitly qualify names from the std namespace. For example, instead of using cout, use std::cout. This makes the code more readable and avoids potential name conflicts.