Chapter 3
Types, Constants, Variables
(1 week)

References and Pointers, declaration, scope, initialisation,

array : declaration, initialisation, namespace, dynamic allocation

Constants are variables whose values cannot be changed after they are initialized. There are a couple of ways to define constants in C++:

1. Using const Keyword:

```
const int MY_CONSTANT = 42;
```

2. Using #define Preprocessor Directive:

```
#define MY_CONSTANT 42
```

The #define directive is a preprocessor directive that performs simple textual replacement.
It is generally recommended to use const for defining constants in C++.

C++ supports a rich set of data types, and they can be broadly categorized into the following:

1. Fundamental / Primitive Types:

int: Integer type.
float: Single-precision floating-point type.
double: Double-precision floating-point type.
char: Character type.
bool: Boolean type (true or false).

C++ is a statically typed language, so you need to declare the type of a variable before using it.

Syntax:

type variableName = value;

Examples:

```
int myInteger = 42;
float myFloat = 3.14;
double myDouble = 2.71828;
char myChar = 'A';
bool myBool = true;
```

2. Derived Types:

Array: A collection of elements of the same data type.

Pointer: A variable that stores the memory address of another variable.

Reference: An alias for a variable.

Function: A block of code that performs a specific task.

3. User-Defined Types:

Struct: A user-defined data type that groups related variables under a single name.

Class: Similar to a struct but with additional features, such as encapsulation and inheritance.

Union: A special data type that allows storing different data types in the same memory location.

4. Enumeration Types:

enum: A user-defined type consisting of named constants.

5. Void Type:

void: Represents the absence of a type. Used in functions that do not return a value or for generic pointers.

6. Standard Template Library (STL) Types:

Various container classes like vectors, lists, sets, maps, etc.

Algorithms provided by the STL.

Pointers and References

Pointers:

- In C++, a pointer is a variable that stores the memory address of another variable.
- Pointers are used to work with memory directly and enable dynamic memory allocation and manipulation.
- They play a crucial role in tasks like managing arrays, implementing data structures, and interacting with functions that work with memory.

1. Declaration:

To declare a pointer, you use the asterisk * symbol:

```
int* myPointer;  // Declaration of a pointer to an integer
```

This declares a pointer named myPointer that can point to an integer.

2. Initialization:

Pointers should be initialized with the address of a variable:

```
int myVariable = 10;
int* myPointer = &myVariable; // Initialize the pointer with the address of
                              // myVariable
```

3. Dereferencing:

To access the value at the memory location a pointer is pointing to, you use the dereference operator *:

```
int myValue = *myPointer;  // Access the value myPointer is pointing to
```

Two operations allow to retrieve the address of an object and access the pointed object (value). They are respectively called **indirection** and **dereference.**

These operators are **&** and ***** respectively.

***ptr and var = value of the variable .**

ptr and &var = addresse of the variable.

Exeample:

```
#include<iostream>
using namespace std;

main()
{int x=10;
int *px=&x;
cout<<"x= " <<x<<"\n";
cout<<" Using the pointer x= " <<*px<<"\n";
 x*=2;
cout<< " Using the pointer x= " <<*px<<"\n";
*px*=3;
cout<<"x= " <<x<<"\n";
return 0;}
```

Reference

- In C++, a reference is an alias or alternative name for an existing variable.
- It provides an alternative syntax for accessing the same memory location as the original variable.
- References are declared using the & symbol.

1. Declaration:

To declare a reference, you use the & symbol after the data type:

```
int originalVariable = 42;
int &myReference = originalVariable;
// Declaration of a reference
```

In this example, myReference is a reference to originalVariable.

2. Initialization: References must be initialized when they are declared, and once initialized, they cannot be re-assigned to refer to another variable.

They act as an alias to the variable they are referencing.

Example:

```
#include<iostream>
using namespace std;
main()
{int i=0; int &ri=i;
cout<< " using the variable, i= " <<i<<"\n";
cout<<" using the reference, ri= " <<ri<<"\n";
 ri+=2;
cout<< " using the variable, i= " <<i<<"\n";
cout<< " using the reference ri= " <<ri<<"\n";
return 0;}
```

Scope of variables, pointers and references

- In C++, the scope of variables, pointers, and references refers to the region of the program where they can be accessed or modified.
- The scope is determined by the location of their declaration in the code.

1. Variables:

Local Variables: Variables declared within a block of code, such as within a function, have local scope. They are only accessible within that block.

Example:

```
void myFunction()
{    int localVar = 10;  // localVar has local scope
}
```

Global Variables: Variables declared outside of any function, class, or block have global scope. They can be accessed from anywhere in the program.

Example:

```
int globalVar = 20;  // globalVar has global scope

int main()
{    // Access globalVar here}
```

2. Pointers:

The scope of pointers is similar to the variables they point to. The pointer variable itself has its own scope, but the memory it points to might have a different scope.

```
void myFunction()
 {    int localVar = 10;
      int* ptr = &localVar;  // ptr has local scope
     // Access localVar through ptr}
```

If a pointer points to dynamically allocated memory (using new), its scope is often determined by the block of code in which it was created.

```
int* dynamicPtr = new int;  // dynamicPtr has dynamic memory with
//scope defined by the    programmer
```

3. References:

References are similar to pointers in terms of scope. A reference variable itself has scope, but it is an alias for another variable, and its scope is tied to the variable it refers to.

```
void myFunction()
 {    int localVar = 10;
       int& ref = localVar;  // ref has local scope, but it refers to localVar
                                  // Access localVar through ref}
```

References are commonly used as function parameters to allow modifying the original variable directly, and their scope is limited to the function in which they are declared.