

# TASK 2

Irem Aslan

December 2024

## 0.1 C++ VARIABLES

### 0.1.1 Sizes of Variables

```
#include <iostream>
#include <limits>
#include <iomanip>

using namespace std;

int main(){

    /* there are no such types as short float, long long float,
    unsigned float, or unsigned double. */

    cout<< setw(10)<<left<<"=====";
    cout<< setw(10)<<right<<"====="<<endl;
    cout<< setw(10)<<left<<"Type";
    cout<< setw(2)<<right<<" ";
    cout<< setw(14)<<right<<"Size"<<endl;
    cout<< setw(10)<<left<<"=====";
    cout<< setw(10)<<right<<"====="<<endl;

    cout<< setw(10)<<left<<"int";
    cout<< setw(2)<<right<<" ";
    cout<< setw(12)<<right<<sizeof(int)<<endl;

    cout<< setw(10)<<left<<"short int";
    cout<< setw(2)<<right<<" ";
    cout<< setw(12)<<right<<sizeof(short int)<<endl;

    cout<< setw(10)<<left<<"long long int";
    cout<< setw(2)<<right<<" ";
    cout<< setw(9)<<right<<sizeof(long long int)<<endl;

    cout<< setw(10)<<left<<"unsigned int";
    cout<< setw(2)<<right<<" ";
    cout<< setw(10)<<right<<sizeof(unsigned int)<<endl;

    cout<< setw(10)<<left<<"float int";
    cout<< setw(2)<<right<<" ";
    cout<< setw(12)<<right<<sizeof(float)<<endl;

    cout<< setw(10)<<left<<"double";
    cout<< setw(2)<<right<<" ";
    cout<< setw(12)<<right<<sizeof(double)<<endl;

    cout<< setw(10)<<left<<"long double";
```

```

cout<< setw(2)<<right<<" ";
cout<< setw(12)<<right<<sizeof(long double)<<endl;

cout<< setw(10)<<left<<"=====";
cout<< setw(10)<<right<<"====="<<endl;

}

```

| Type          | Size |
|---------------|------|
| int           | 4    |
| short int     | 2    |
| long long int | 8    |
| unsigned int  | 4    |
| float int     | 4    |
| double        | 8    |
| long double   | 16   |

## 0.1.2 Sizes of Pointers

```

#include <iostream>
#include <limits>

using namespace std;

int main(){

int a=10;
int *pointerInt= &a;
cout<<"pointer int-->"<<sizeof(pointerInt)<<endl;

short int b=10;
short int *pointerShortInt= &b;
cout<<"pointer short int-->"<<sizeof(pointerShortInt)<<endl;

long long int c=10;
long long int *pointerlonglongInt = &c;
cout<<"pointer long long int-->"<<sizeof(pointerlonglongInt)<<endl;

unsigned int d=10;
unsigned int *pointerunsignedInt= &d;

```

```

cout<<"pointer unsigned int-->"<<sizeof(pointerunsignedInt) <<endl;

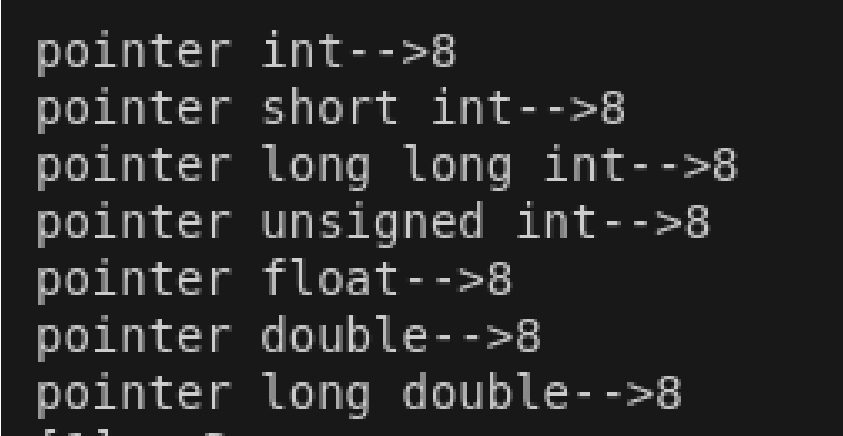
float e=10;
float *pointerFloat= &e;
cout<<"pointer float-->"<<sizeof(pointerFloat)<<endl;

double f=10;
double *pointerDouble=&f;
cout<<"pointer double-->"<<sizeof(pointerDouble)<<endl;

long double g=10;
long double *pointerlongDouble= &g;
cout<<"pointer long double-->"<< sizeof(pointerlongDouble)<<endl;

}

```



```

pointer int-->8
pointer short int-->8
pointer long long int-->8
pointer unsigned int-->8
pointer float-->8
pointer double-->8
pointer long double-->8

```

### 0.1.3 Numeric Limits

```

#include <iostream>
#include <limits>

using namespace std;

int main(){

cout<<"Lower limit of int: "
<< numeric_limits<int>::min()<<endl;
cout<<"Upper limit of int: "
<< numeric_limits<int>::max()<<endl;
cout<<"Lower limit of short int: "
<< numeric_limits<short int>::min()<<endl;

```

```

cout<<"Upper limit of int: "
<< numeric_limits<short int>::max()<<endl;
cout<<"Lower limit of long long int: "
<< numeric_limits<long long int>::min()<<endl;
cout<<"Upper limit of int: "
<< numeric_limits<long long int>::max()<<endl;
cout<<"Lower limit of unsigned int: "
<< numeric_limits<unsigned int>::min()<<endl;
cout<<"Upper limit of int: "
<< numeric_limits<unsigned int>::max()<<endl;
cout<<"Lower limit of float: "
<< numeric_limits<float>::min()<<endl;
cout<<"Upper limit of float: "
<< numeric_limits<float>::max()<<endl;
cout<<"Lower limit of double: "
<< numeric_limits<double>::min()<<endl;
cout<<"Upper limit of double: "
<< numeric_limits<double>::max()<<endl;
cout<<"Lower limit of long double: "
<< numeric_limits<long double>::min()<<endl;
cout<<"Upper limit of long double: "
<< numeric_limits<long double>::max()<<endl;

}

```

```

Lower limit of int: -2147483648
Upper limit of int: 2147483647
Lower limit of short int: -32768
Upper limit of int: 32767
Lower limit of long long int: -9223372036854775808
Upper limit of int: 9223372036854775807
Lower limit of unsigned int: 0
Upper limit of int: 4294967295
Lower limit of float: 1.17549e-38
Upper limit of float: 3.40282e+38
Lower limit of double: 2.22507e-308
Upper limit of double: 1.79769e+308
Lower limit of long double: 3.3621e-4932
Upper limit of long double: 1.18973e+4932

```

Text after it ...

### 0.1.4 Auto Variables

In C++, the auto keyword is used for type inference, meaning that the compiler automatically deduces the type of a variable based on its initializer expression. This allows you to avoid explicitly specifying the type, and it can simplify code, especially when dealing with complex types or iterators.

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

int main(){

    auto a=10;  // 'x' is deduced to be of type 'int'
    auto b=10.5; // 'y' is deduced to be of type 'double'

    cout<<"a: "<<a<<endl;
    cout<<"b: "<<b<<endl;

    return 0;

}
```

If you want to deduce a reference type, you need to use auto& or const auto& for references:

```
auto a= 10; // 'a' is deduced to be of type 'int'
auto& b= a; // 'b' is deduced to be of type 'int&'
```

### 0.1.5 Const and Constexpr Keywords

#### Const Keyword

In C++, the const keyword is used to define constant variables or to indicate that something should not be modified. It can be applied to variables, pointers, and even function parameters to ensure that their values or behaviors cannot be changed after they are initialized or assigned.

const: When a variable is declared as const, its value cannot be changed once it has been initialized.

#### 1.Constant Variables:

When you declare a variable as const, its value cannot be changed after initialization.

```
const int x= 15;
```

```
std::cout<< x << std::endl;
```

## 2.Constant Pointers:

A constant pointer means that the pointer itself cannot point to a different memory address after it is initialized, but the value at the address it points to can still be modified.

```
int a = 5;
int* const ptr = &a;
*ptr=20;
// ptr = &b;           // Error: cannot change the address 'ptr' is pointing to
```

## 3.Pointer to Constant Data:

A pointer to constant data means that the data (value) the pointer points to cannot be modified, but the pointer itself can be changed to point to a different address.

```
const int* ptr= &a;
// *ptr=20;           // Error: cannot change the value 'ptr' is pointing to
ptr = &b;
```

## 4.Constant References:

A constant reference ensures that the value referred to by the reference cannot be modified. This is often used in function parameters to prevent the function from altering the argument passed to it, while still allowing the argument to be passed efficiently (without copying large objects).

```
void print(const int& x){
// x = 20; // Error: cannot modify a const reference
std::cout<< x << std::endl;
}
```

## 5.Constant Functions:

In C++ you can use const for function arguments to indicate that they should not be changed inside the function. This is commonly used for arguments passed by reference to avoid accidental modification of the original data.

```
void display(const std::string& message){
    // message = "Hello"; // Error: cannot modify a const reference
    std::cout<< message << std::endl;
}
```

## Constexpr Keyword

The constexpr keyword in C++ is used to indicate that the value of a variable or function can be evaluated at compile time. It tells the compiler that the variable or function's result is constant and can be computed during the compilation process rather than at runtime. This is useful for optimizing performance and ensuring certain computations are done before the program is even executed.

```
//example 1
constexpr int max_size = 100;

//example 2
constexpr int square(int x) {
    return x * x;
}
```

!!constexpr functions can only use a limited set of operations, such as basic arithmetic and conditional logic.

!!They cannot have dynamic memory allocation (like new or delete), and they cannot call other non-constexpr functions.

### Benefits of using constexpr:

- Performance: By evaluating expressions at compile time, you can avoid the overhead of computation at runtime, which can lead to performance improvements, especially in high-performance applications.
- Code clarity and safety: By enforcing that certain values or computations must be constant and known at compile time, you avoid potential bugs related to dynamic computations and improve the clarity of your code.

## 0.1.6 Fixed-Width Types

These types are part of the C++ Standard Library and are defined in the "stdint" header.

### Fixed-Width Integer Types



They allow you to define integers with an exact number of bits, ensuring consistency regardless of the system or compiler being used.

For example:

`int32_t`: A **signed** integer that is exactly 32 bits (4 bytes) wide

`uint32_t`: An **unsigned** integer that is exactly 32 bits wide

`int64_t`: A **signed** integer that is exactly 64 bits wide

`uint64_t`: An **unsigned** integer that is exactly 64 bits wide

### **Fixed-Width Float Types**

Fixed-width floating-point types specify the exact bit-width of a floating-point value.

For example:

`float32_t`: 32-bit floating-point (equivalent to **float**).

`float64_t`: 64-bit floating-point (equivalent to **double**).