**Constants/Literals**

Constants refer to fixed values that do not change during the execution of a program

They are called literals

Literal constants do not have memory locations

Constants can be of any of the basic data types

Examples

|  |  |
| --- | --- |
| 123 | Decimal Integer |
| 12.34 | Floating point integer |
| 037 | Octal integer |
| 0x2 | Hexadecimal integer |
| “C++” | String Constant |
| ‘A’ | Character constant |
| L’ab’ | Wide character constant |

# Defining Constants

Two simple way

1. #define preprocessor
2. const keyword

## #define preprocessor

Defines a substitute text for a name

Creates symbolic constants, called a macro

Example: #define LENGTH 10

## const keyword

You can use const prefix to declare constants with a specific type

const type variable = value;

Example: cons tint LENGTH = 10;

## constexpr

Check in C++ 11 topics

## enum

Check in file enum C CPP

# Difference between #define and const

* #define is a preprocessor directive while const variables are actual variables
* Advantage of const over #define is type checking
* Disadvantage is extra space for variable which is immaterial due to optimizations done by compilers
* In general const are better options if we have choice
* There are situations where #define cannot be replaced by const
* #deifne can take parameters
* #define can also be used to replace some text in a program with another text

**NOTE:**

1. In C++ we can use const in a constant expression, in C not

const int size = 10;

char name[size]; // C++ ✓ C 🗶

1. Character literals have different type in C and C++

* In C character literal is treated as an int type
* In C++ character literal is treated as char type

# Raw String Literal (C++ 11)

From C++11 we can use raw strings in which escape characters (\n, \t or \” …) are not processed. The syntax of raw string is that the literal starts with R”(and ends in)”.

string str1 = “Geeks\nFor\nGeeks\n”;

string str2 = R“Geeks\nFor\nGeeks\n”;

std::cout << str1 << std::endl;

std::cout << str2 << std::endl;

Output:

Geeks.

For.

Geeks.

Geeks.\nFor.\nGeeks.\n

# User defined literals (C++11)

Why to use UDL? Consider example

Long double weight = 2.3; // pounds?, Kilogram?, Grams?

With UDL, we attach units to the values which has following advantages

The code becomes readable

**Conversion computations are done at compile time**

weight = 2.3kg;

ratio = 2.3kg/1.21b;

To calculate above ratio it is necessary to convert them to same units.

UDLs help us to overcome unit translation cost

We can define user defined literals for user defined types and new form of literals for built-in types

Value of UDLs is substituted with the actual value defined in the code by the compiler at compile time

UDLs do not save much of coding time but more and more calculation can be shifted to compile time for faster execution

Example

"hello"s // string

4.3i // imaginary

101000111101001b // binary

53h // hours

234093270497230409328432840923849 // extended-precision

UDLs are treated as a call to a literal operator

Only suffix form is supported

The name of the literal operator is **operator “”** followed by the suffix

Example

#include<iostream>

#include<iomanip>

using namespace std;

long double operator"" \_kg( long double x ) {return x\*1000;}

long double operator"" \_g( long double x ) {return x;}

long double operator"" \_mg( long double x ) {return x / 1000;}

int main() {

long double weight = 3.6\_kg;

cout << weight << endl;

cout << setprecision(8) << ( weight + 2.3\_mg ) << endl;

cout << ( 32.3\_kg / 2.0\_g ) << endl;

cout << ( 32.3\_mg \*2.0\_g ) << endl;

return 0;

}

Output

3600

3600.0023

16150

0.0646

#include <iostream>

#include <complex>

using namespace std;

constexpr complex <double> operator"" \_i(long double d) {

return complex <double> {0.0, static\_cast<double>(d)};

}

int main() {

complex <double> z = 3.0 + 4.0\_i;

complex <double> y = 2.3 + 5.0\_i;

std::cout << "z+y = " << (z+y) << std::endl;

std::cout << "z\*y = " << (z\*y) << std::endl;

std::cout << "abs(z) = " << abs(z) << std::endl;

return 0;

}

Output:

z+y = (5.3,9)

z\*y = (-13.1,24.2)

abs(z) = 5

* constexpr is used to enable compile time evaluation

# Restriction

UDL can only work with the following parameters:

char const\*

unsigned long long

long double

char const\*, std::size\_t

wchar\_t const\*, std::size\_t

char16\_t const\*, std::size\_t

char32\_t const\*, std::size\_t

# Type difference of character literals in C and C++

In C, a character literal is treated as int type

In C++, a character literal is treated as char type

It is required to support function overloading

#include <stdio.h>

int main() {

printf("sizeof('V') = %zu sizeof(char) = %zu", sizeof('V'), sizeof(char));

return 0;

}

Output:

sizeof('V') = 4 sizeof(char) = 1 //In C

sizeof('V') = 1 sizeof(char) = 1 //In C++

# END