DATA TYPES IN C++ – PART 1

Built in data types :

1. Fundamental Types (Integer and Floating Point Number)
2. Compound Types (Arrays, Strings, pointers and structure)

To Store an information

1. Where the information is stored
2. What value is kept there
3. What kind of information is stored.

Ex – int age; age = 21;

Rules for naming a variable:

1. The only character you can use in names are alphabetic characters, numeric digits, and the underscore(_) character.
2. The first character in name cannot be a numeric digit.
3. You cannot use c++ keyword for a name.
4. Case sensitive & spelling sensitive (Uppercase and lowercase characters aren't the name).
5. Names begins with two underscore characters or with an underscore characters are reserved for use by the implementation(Compiler and resource it uses).

6. C++ place no limits on the length for a name (Some platforms might have their own length limits).
7. __itisavariable or _Itisavariable doesn’t produce an error, instead it leads to an undefined behavior.
8. All characters in a name are significant.

Last point(8th) differentiates C++ from ANSI C (C99), which guarantee only that the first 63 characters are significant (Two variable are termed as same if 63 characters are identical even 64th differ).

Integers Types :

Integers are number with no fractional part.

Ex – 23,45,67,999999999 etc

There are a lot of integers, assuming that you consider an infinite number to be a lot, so no finite amount of computer memory can represent an impossible integers.

So, a language can represent a subset of all integers.

We call **width** when the amount of memory used by an integers.

The more memory uses, the more wider it is.

Types Of Integers(In order of increasing width) :

1. **Char**
2. **Short**
3. **Int**
4. **Long**
5. **Long Long (With C++ 11)**

Each comes in both signed and unsigned versions. That gives you a choice of ten different integer types.

Signed : It can represent both positive and negative numbers.

Unsigned : It can represent only positive numbers(Can't represent any negative number).

Computer memory consists of units called bits.

1. A short is at least 16 bit wide.
2. An int integer is at least as big as short.
3. A long integer is at least 32 bits wide and at least as big as int.
4. A long long integer is at least 64 bits wide and at least as big as long.

Bits And Bytes :

The fundamental unit of computer memory is the *bit*.

It is like an electronic switch that you can set either off or on. Off represents the value 0, and on represents the value 1.

A single bit of memory can be set to 2 different combinations. An 8-bit memory can be set to 256 different combinations.

So, an 8-bit can represent either the values from 0 to 255.

Or, it can represent value in the range of -128 to 127.

This means a 64-bit can have 18,446,744,073,709,551,616 different values.

One thing is that ***unsigned long*** can't hold the Earth's current population or the numbers of stars in the galaxy, but long long can.

A byte usually means an 8-bit unit of memory. Byte that describes the amount of memory in a computer.

1 kilobytes(kb) equal to 1024 bytes and a megabyte(mb) equal to 1024 kb.

Byte is implementation dependent. So a 2 byte int could be 16 bit in one system and 32 bit on another.

Declaration :

Declaration is same as int.

```
short cricketScore;
int roomTemperature;
long myPosition;
long long populationOfEarth;
```

Actually *short* is a short form of *short int* and *long* is short for *long int*.

But use modern technique.

We can use *sizeof* operator to know the size in our pc or laptop.

```
cout << “int is  ” << sizeof(int) <<” bytes.\n”;
```

climit header file contain information about integer (Again it came from C. In older C it is limit.h).

Symbolic Constant	**Represents**
CHAR_BIT	Numbers of bits in a *char*
CHAR_MAX	Maximum *char* value
CHAR_MIN	Minimum *char* value
SCHAR_MAX	Maximum *signed char* value
SCHAR_MIN	Minimum *signed char* value
UCHAR_MAX	Maximum *unsigned char* value
SHRT_MAX	Maximum *short* value
SHRT_MIN	Minimum *short* value
USHRT_MAX	Maximum *unsigned short* value
INT_MAX	Maximum *int* value
INT_MIN	Minimum *int* value
UNIT_MAX	Maximum *unsigned int* value
LONG_MAX	Maximum *long* value
LONG_MIN	Minimum *long* value
ULONG_MAX	Maximum *unsigned long* value
LLONG_MAX	Maximum *long long* value
LLONG_MIN	Minimum *long long* value
ULLONG_MAX	Maximum *unsigned long long* value

Initialization :

Initialization combines assignment with declaration.

Ex – int largestInt = INT_MAX;

This statement declares the *largestInt* variable and set it to be the largest possible type *int* value.

You can also use literal constants.

You can initialize a variable to another variable.

You can even initialize a variable to an expression.

Previous initialization technique comes from C. C++ has another initialization syntax that is not shared with C.

int age (21); // Initialize age and set age value to 21.

If you know what the initial value of a variable should be, initialize it.

short age ; // any value it could be

age = 21; // ahh! Ok

Initialization with C++11:

There's another format of initialization that's used with arrays and structures but in C++98 can also be used with single-valued variables.

int age = {21} ; //set age to 21

Using a braced initializer for a single-valued variable hasn't been particularly common.

C++11 standard is extending in some ways.

1. It can be used with or without = sign.

int age{21}; //set age to 21

int friendAge={22};

2. The braces can be left empty, in which case the variable is initialized to 0.

int likes = {}; // set likes to 0

int comments{}; // set comments to 0

3. It provides better protection against type conversion errors.

Integer Literals

C++ let you write integers in three different number base (Just like C).

1. **Base 10** (The public favorite)
2. **Base 8** (The old Unix favorite)
3. **Base 16** (The hardware hacker's favorite)

C++ uses the first digit or two to identify the base of a number constant.

- If the first digit is in the range of 1 to 9, the number is base 10 (decimal).Ex – 93, 25, 784, 4568,etc.
- If the first digit is 0 and the second digit is in the range of 1 to 7, the number is base 8 (octal).Ex – 042, 03567, 0746,etc.
- If the first two characters are **0x** or **0X**, the number is base 16 (hexadeciamal). Ex – 0x42, 0x4A5, 0XA4B7C4D3,etc.

For hexadecimal values, the characters a-f and A-F represent the hexadecimal digits corresponding to the values 10-15.

See the program – IntegerLiterals.cpp

If you want to display a value in hexadecimal or octal form, you can use some special features of *cout.*

Just like the *iostream* header files provides the *endl* manipulator to give *cout* the message to start new line. Similarly, it provides the *dec* , *oct* and *hex* maniupulators to give *cout* the message to display integers in decimal, octal and hexadecimal formats.

```
cout << hex;
```

This code doesn't print anything onscreen. Instead, it changes the way *cout* display integers.

You should know that identifier *hex* is a part of the *std* namespace and the program uses the namespace and that's why the program can't use *hex* as a name of a variable.

However, if you omit the using of namespace in your program you can use hex as the name of a variable.

- C++ stores integer constants as type *int* unless there is no reason to do otherwise. .Two such reasons are if you use a special suffix to indicate a particular type or if a value is too large to be an int.

An l or L suffix on an integer means the integer is a type *long* constant, a u or U suffix indicates an unsigned long constant, ul indicates a type *unsigned long* constant. C++11 provides ll and LL suffix for type long long, and ull, Ull, uLL, and ULL for unsigned long long.

See the programs – IntegerLiterals2.cpp

char **Type :** Characters and small integers

It is designed to store characters, such as letters and numeric digits.

It is large enough to represent entire range of basic symbols – all the letters, digits, punctuation and the like-for the target computer system.

In practice, many system support fewer than 128 kinds of characters, so a single byte can represent the whole range.

The most common symbol set in the United States is the ASCII character set. A numeric code represent each character in the set.

Ex – 65 is the code for character A, and 77 is for M.

- ✓ C++ implementation uses whatever code is native to its host system. Ex – EBCDIC on an IBM mainframe.

Neither EBCDIC nor ASCII serve international needs that well. So, C++ supports a *wide* character type (*wchar_t*) that can hold larger range of values which are used by International Unicode character set.

See the programs – MoreChar.cpp

cout.put() : A member function

It is an example of C++ OOP concept, the member function.

A class defines how to represent data and how to manipulate it.

A member function belongs to a class and describes method for manipulating class data.

To use a class member function with an object such as *cout,* you use a period to combine the object name (cout) with the function name (put()). The period is called the membership operator.

The notation *cout.put()* means to use the class member function *put()* with the class object *cout*.

cout << '!';

//print the ASCII code for the ! character rather than simply display !.

cout.put('!');

//print the character as we desired.

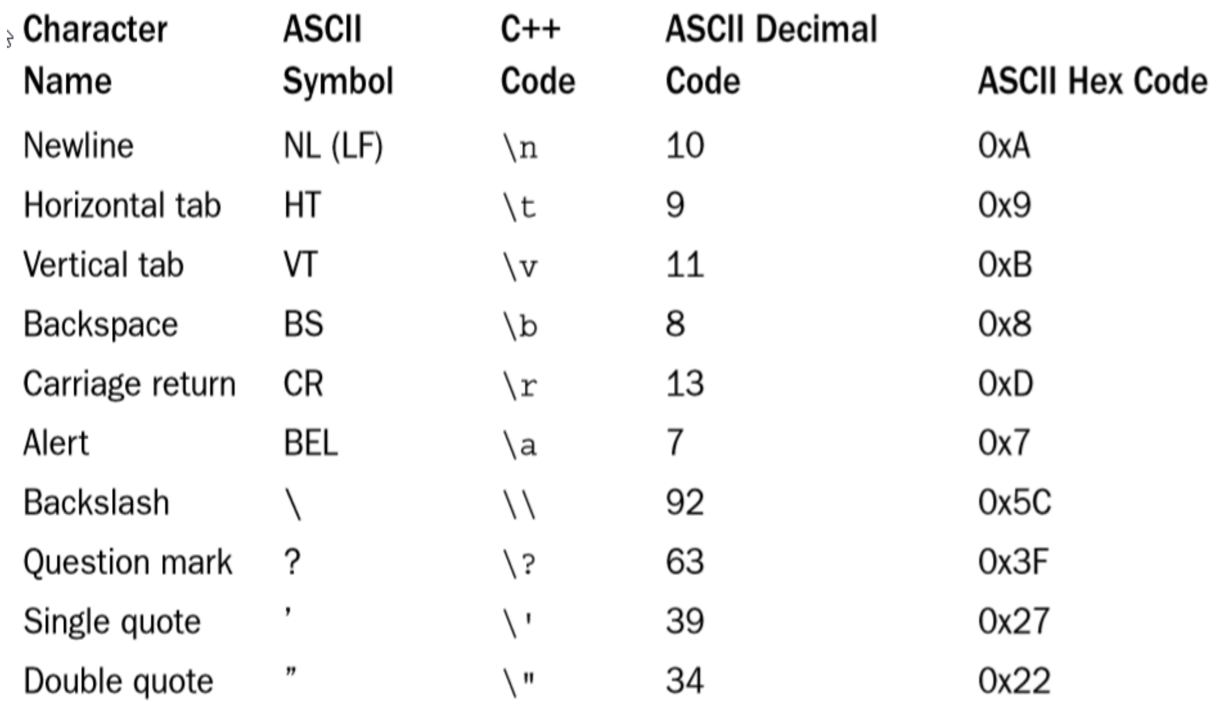
- There are some characters that you can't enter into a program directly from the keyboard. For example, you can't make the newline character part of a string by pressing the Enter key ; instead, the program editor interprets that keystroke as a request for it to start a new line in your source code file.

C++ has special notations called *escape sequences*.

\a represent the alert character which beeps you terminal's speaker or ring its bell.

\n represents a new line.

Here are the escape sequence code:



Character Name	ASCII Symbol	C++ Code	ASCII Decimal Code	ASCII Hex Code
Newline	NL (LF)	`\n`	10	0xA
Horizontal tab	HT	`\t`	9	0x9
Vertical tab	VT	`\v`	11	0xB
Backspace	BS	`\b`	8	0x8
Carriage return	CR	`\r`	13	0xD
Alert	BEL	`\a`	7	0x7
Backslash	\	`\\`	92	0x5C
Question mark	?	`\?`	63	0x3F
Single quote	’	`\'`	39	0x27
Double quote	”	`\"`	34	0x22

Universal Character Names :

A universal characters begins either with \u or \U. The \u form is followed by 8 hexadecimal digits, and the \U form by 16 hexadecimal digits.

These digits represents the ISO 10646 code point for the character. ISO 10646 is an international standard under development that provides a numeric code for wide range of characters.

int k\u00F6rper;

cout << "Let them eat g\u00E2teau.\n";

The ISO 10646 code point for ö is 00F6,and the code point for â is 00E2.

So it print *"Let them eat gâteau".*

If your system doesn't support ISO 10646, it might display some other characters.

signed Char and unsigned char:

int, char is not *signed* by default. Neither it is unsigned by default.

Ex-

char c_one; // maybe signed, maybe unsigned

unsigned char c_two; // definitely unsigned char

signed char c_thre; // definitely signed

wchar_t:

Programs might have to handle character sets that don’t fit within the confines of a single 8-bit byte. Ex – Japanese Kanji system.

The usual 8-bit *char* can represent the basic character set. wchar_t is used for wide character type, it can represent the extended character set.

The *wchar_t* type is an integer type with sufficient space to represent the largest extended character set used on the system.

Note : wchar_t type has the same size and sign properties as one of the other integer types,which is called the underlying type.The choice of underlying type depends on the implementation,so it could be unsigned short on one system and int on another.

The *cin* and *cout* family consider input and output as consisting of streams of chars, so they are not suitable for handling the *wchar_t* type.The iostream header file provides parallel facilities in the form of *wcin* and *wcout* for handling *wchar_t* streams.

Also you can indicate a wide-character constant or string by preceding it with an L.

Ex-

wchar_t w_char_one = L’M’; // a wide-character constant

wcout << L”large” << endl; // printing a wide- character string

New C++11 types : *char16_t* and *char32_t*

Why ?

As the programming community gained more experience with Unicode,it became clear that the *wchar_t* type wasn’t enough. It turns out that encoding characters and strings of characters on a computer system is more complex than just using the Unicode numeric values (called code points).

In particular, it’s useful, when encoding strings of characters, to have a type of definite size and signedness. But the sign and size of *wchar_t* can vary from one implementation to another.

So C++11 introduces the types char16_t,which is unsigned and 16 bits, and *char32_t*,which is unsigned and 32 bits.

The prefixes u and U are used to indicate character literals of types *char16_t* and *char32_t*, respectively.

Ex –

char16_t ch1 = u'q'; // basic character in 16-bit form

char32_t ch2 = U'\U0000222B'; // universal character name in 32-bit form

Note : Like wchar_t,char16_t and char32_t each have an underlying type,which is one of the built-in integer types.But the underlying type can be different on one system from what it is on another.

The **bool** type:

TheANSI/ISO C++ Standard has added a new type (new to C++,that is),called bool. It’s named in honor of the English mathematician George Boole,who developed a mathematical representation of the laws of logic.

A **Boolean variable** is one whose value can either be **true** or **false**.

C++ interprets nonzero values as true and zero values as false. We’ll discuss it later.

Ex –

bool isCorrect = true;

The literals *true* and *false* can be converted to type int by promotion, with true converting to 1 and false to 0.

Int ans = true; // ans assigned 1

Int score = false; // score assigned 0

Also any numeric or pointer value can be converted implicitly (without an explicit type cast) to a *bool* value.

Any nonzero converts to true.

Ex –

bool isReady = 41; // isReady assigned true

bool locationGiven = -784; // locationGiven assigned true

Whereas a zero value converts to false.

bool score = 0; // score assigned false

The ***const*** qualifier:

After initializing a constant value, compiler doesn't let you change it.

General form :

const type name = value;

Ex –

const int noOfDaysInWeek = 7; // noOfDaysInWeek is symbolic constant for 12

If you will try to change a variable, g++ gives you an error message that the program used an assignment of a read-only variable.

const int noOfHands; // value of noOfhands is undefined

noOfHands = 2; // too late!

If you don't define the value of a constant this may be suspicious, so the good idea is to define the constant value when you know it.

Note : If you are coming to C++ from C and you are about to use #define to define a symbolic constant, use const instead.

Floating-Point Number :

These numbers let you represent numbers with fractional parts.

Ex – 1.54, 789.45, etc.

It also provides much greater range in values. If a number is too large to be represented as type *long* you can use one of the floating-point types.

Ex – The number of bacterial cells in a human body (estimated to be greater than 100,000,000,000,000).

A computer stores such values in two parts.

1. One part represents a value.
2. The other part scales that value up or down.

Ex -

Let's take two numbers 83.3478 and 83347.8.

These are identical except for scale.

Representation of each number is as follows:

1. First one is represented as 0.833478 (the base value) and 100 (the scaling factor).
2. Second one can be represented as 0.833478 (the base value) and 100,000 (a bigger scaling factor).

The scaling factor serves to move the decimal point, hence the term **floating-point**.

Note : Floating-point numbers let you represent fractional, very large and very small values, and they have internal representation much different from those of integers.

C++ has two ways of writing floating-point numbers.

1. Use the standard decimal-point notation you've been using much of your life. Ex-78.56,32343.36575,etc.
2. E- notation. Ex – 3.45E6 (3.45 * 1,000,000). Here 6 (in 10^6) is called an *exponent*, and the 3.45 is termed the *mantissa*.

Here are some more examples of E notation.

2.52e+8 //can use E or e, + is optional

8.33E-4 // exponent can be negative

7E5 // same as 7.0E+05

-18.32e12 // can have + or – sign in front

1.69e12 // 2010 Brazilian public debt in reais

5.98E24 // mass of earth in kilogram

9.11e-31 // mass of an electron in kilogram

To use a negative exponent means to divide by a power of 10 instead of to multiply by a power of 10.

So, 8.33E-4 means $8.33/10^4$, or 0.000833.

Note : The form d.dddE+n means move the decimal point n places to the right, and the form d.dddE-n means move the decimal point n places to the left. This moveable decimal point is the origin of the term "floating-point."

Floating-Point Types :

C++ has three floating-point types:

1. **Float**
2. **double**
3. **long double**

These types are described in terms of the number of significant figures they can represent and the minimum allowable range of exponents.

Significant Figures are the meaningful digits in a number.

Ex – 85,458 has 5 significant figures, 85,000 has two significant figures, 85.458 has also significant figures.

C and C++ requirements for significant digits amount to *float* being at least 32 bits, *double* being at least 48 bits and certainly no smaller than *float,* and *long double* being as least as big as *double.* All three can be the same size.

Typically, however, *float* is 32 bits, *double* is 64 bits, and *long double* is 80, 96, or 128 bits. Also the range in exponents for all three types is at least -37 to +37.

You can look in the *cfloat* (for C folat.h header file) to find the limits for your system.

See the program – Float2.cpp

Normally *cout* drop trailing zeros.

Ex., It would display 5555555.750000 as 5555555.75. The call to *cout.setf()* overrides that behavior, at least new implementations.

Notes : Don't rely on the include files only as a source of mystic and arcane knowledge; feel free to open them up and read them.

Floaring-point constants :

By default, floating-point constants such as 6.58 or 3.7E9 are type *double*.

If you want a constant to be type *float*, you use an *f* or *F* suffix.

For type *long double,* you use an l or L suffix (Always try to use uppercase L because the lowercase l sometimes looks like 1).

Ex-

1.234f // a float constant

2.345E14F // a float constant

4.2345633E28 // a double constant

7.8L // a long double constant

Advantages and disadvantages of floating-point numbers :

Floating point numbers have two advantages over integers. First, they can represent values between integers. Second, because of the scaling factor, they can represent a much greater range of values.

On the other hand, floating point operations usually are slightly lower than integer operations, and you can lose precision.

See the program – Float3.cpp

<u>**Classifying Data Types**</u> :

C++ brings some order to its basic types by classifying them into families.

Types signed char, short, int, and long are termed signed integer types. C++11 adds long long to that list.

The unsigned versions are termed unsigned integer types. The bool, char, wchar_t, signed integer, and unsigned integer types together are termed integral types or integer types.

C++11 adds char16_t and char32_t to that list. The float, double, and long double types are termed floating-point types.

Integer and floating-point types are collectively termed arithmetic types.

C++ Arithmetic Operators :

C++ have five basic operators :

1. The **+** operator add its operand. Ex – 4+5 evaluates to 9.
2. The **–** operator subtracts the second operand from the first. For example, 15-6 evaluates to 9.
3. The ***** operator multiplies its operands. Ex - 4*5 evaluates to 20.
4. The **/** operator divides its first operand by the second. Ex – 20/5 evaluates to 4.
5. The **%** operator finds the modulus of its operand with respect to the second. That is, it produces the remainder of dividing the first by second. Ex – 14%3 will be equal to 2. Both operands must be integer type; using the % operator with floating-point values causes a compile-time error. If one of the integer is negative, the sign of the result satisfies the following rule: (a/b)*b + a%b equals a. The modulus operator is particularly useful in problems that require dividing a quantity into different integral unit. Ex – Converting inches to feet, converting dollar to pennies, converting paise to rupees,etc.

See the program – SimpleArth.cpp

Order Of Operation : Operator precedence and Associativity

Which operators get applied first.

Ex – 2+3*4 // 20 or 14

So, when more than one operator can be applied to the same operand, C++ uses *precedence* rule to decide which operator is used first.

The arithmetic operators follows the usual algebraic precedence, with *division, multiplication & modulus* done before the *addition & subtraction.*

So the answer will be 14 not 20.

But you can use parentheses to enforce your own priorities.

Ex – (2+3)*4   // now the answer will be 20 not 14

Sometimes the precedence list is not enough.

Ex –

24 / 3 * 2   // 16 or 4

When two operators have the same precedence, C++ looks whether the operators have a left-to-right *associativity* or a right-to-left *associativity*.

Left-to-right associativity means that if two operators acting on the same operand have the same precedence, you apply the left-hand operator first.

For right-to-left means, you apply the right operator first.

Multiplication and division associate left-to-right.

So, the answer will be 16 not 4 (Because we use 3 with the leftmost operator first).

Note : Precedence and associativity rules comes into play only when two operators share the same operand and vice-vesa.

But consider the following example.

3*4+5*6;

Here precedence tell you that multiplication will take place first and later the addition will be done.

But, neither the precedence nor associativity says which multiplication will take place first.

Here, C++ leaves it to the implementation to decide which order works best on a system (Either order gives the same result).

Division Diversion:

The behavior of this operator depends on the type of operands.

Type int / type int // Operator performs *int* division

Type long / type long // Operator performs *long* division

Type double / type double // Operator performs *double* division

Type float / type float // Operator performs *float* division

See the program – Arth2.cpp

Type Conversions :

C++ profusion of types let you match the type to the need.

With 11 integer types and 3 floating-types, the computer can have a lot to handle especially if you start mixing types.

- ✓ Assigning a *short* value to a *long* value doesn't change the values; It just gives the value a few more bytes.
- ✓ However, assigning a large *long* value to a *float* variable or smaller results in the loss of some precision.Ex- *long* value 2111222333 in *float* rounded to 2.11122E9 (Because float can have just six significant figures).

<u>Conversion Types And Their Problems :</u>

Conversion Types	**Problems**
Bigger floating type to a smaller floating types, such as *double* to *float.*	Loss of precision (significant figures); value might be out of range for target type, in which case the result in undefined.
Floating-point type to integer type	Loss of fractional part; original value might be out of range for target type, in which case the result is undefined.
Bigger integer type to smaller integer type	<u>Original value might be out of range for target type; typically just the low-order bytes are copied.</u>

In programs:

float num = 2; //int converted to float

int num (2.7548); // double converted to int

int num = 5.4E13; // result not defined in C++

See the program - Arth3.cpp

Initialization Conversions When {} is used :

C++11 calls an initialization that uses braces a list-initialization.

This form can be used more generally to provides list of values for more complicated data types.

List-initialization doesn't permit narrowing (which is when the type of a variable may not be able to represent the assigned value.

Ex – Conversions of floating type to integer type is not allowed.

Some code clears this {} conversions:

```
const int num1 = 65;

int num2 = 65;

char c1 {25458}; // narrowing, not allowed

char c2 = {65}; //allowed because char can hold 65

char c3 {num1}; //allowed because char can hold 65

char c4 = {num2}; //not allowed, x is not constant

num2 = 25458;

char c5 = num2; //allowed, by this forms of initialization
```

Conversions in expressions :

Consider what happens when you combine two arithmetic types in one expression.

Ex –

short num1 = 20;

short num2 = 54

short num3 = num1 + num2;

When evaluating expressions, C++ converts *bool*, *char*, *unsigned char*, *signed char*, and *short* values to *int*. In particular, *true* is promoted to 1 and *false* to 0. There conversions are termed **integral promotions**.

The *int* type is generally chosen to be the computer's most natural type, which means the computer probably does calculations fastest for that type.

There are more integral promotions:

- The *unsigned short* type is converted to *int* if *short* is smaller than *int.*
- If the two types are the same size, *unsigned short* is converted into *unsigned int*. This rules ensures that there is no data loss in promoting *unsigned short*.
- Similarly, *wchar_t* is promoted to the first of the following types that is wide enough to accommodate its range:
    - *Int*
    - *unsigned int*
    - long
    - unsigned long

When an operation involves two types, the smaller is converted to larger.

The compiler goes through a checklist to determine which conversions to make in an arithmetic expression. Here's the C++ 11 version of the list, which the compiler goes through in order.

1. If either operand is type *long double*, the other operand is converted to *long double*.
2. Otherwise, if either operand is *double*, the other operand is converted to *double*.
3. Otherwise, if either operand is *float*, the other operand is converted to *float*.
4. Otherwise, the operands are *integer* types and the integral promotions are made.
   - In that case, if both operands are *signed* or if both are *unsigned*, and one is of lower rank than the other, it is converted to the higher rank.
   - Otherwise, one operand is *signed* and one is *unsigned*. If the *unsigned* operand is of higher rank than the *signed* operand, the latter is converted to the type of the *unsigned* operand.
   - Otherwise, if the signed type can represent all values of the unsigned type, the unsigned operand is converted to the type of the signed type.
   - Otherwise, both operands are converted to the *unsigned* version of the *signed* type.

This list introduces the concept of ranking of the integer types.

The basic ranking for signed integer types from high to low is *long long, long, int, short,* and *unsigned char.*

Unsigned types have the same rank as the corresponding signed types.

The three types *char, signed char,* and *unsigned char* all have the same rank.

The *bool* has the lowest rank.

The *wchar_t*, *char16_t* and *char32_t* has the same types as their underlying types.

Type Cast :

C++ empowers you to force type conversions explicitly via the type cast mechanism.

The type cast comes in two forms:

1. (typeName) value; // convert value to typeName type
   This form is straight C. Ex -
   ```
   (long) num1;  // returns a type long conversion of num1
   ```
2. typeName (value); // convert value to typeName type
   This form is pure C++. Ex –
   ```
   Long (num1); // return a type long conversion of num1
   ```

cout << int (‘C’); // Display the integer code for ‘C’.

Note :

C++ also introduces four type cast operators. One of them is *static_cast<>* operator. It looks like this

static_cast<typeName> (value) // converts value to typeName type

Ex –

Static_cast<long> (num1) ; // return a type long conversion of num1

See the programs – TypeCast1.cpp

auto **Declarations in C++11**:

C++11 introduces a facility that allows the compiler to deduce a type from the type of an initialization value.

For this purpose, it redefines the *auto* keyword (A keyword dating back to C, but one hardly used).

Just use *auto* instead of the type name in an initializing declaration, and the compiler assigns the variable the same type as that of the initializer.

Ex –

auto n = 1; // n is an int

auto n = 4.5; // n is double

auto n = 2.34E13L; // n is long double

auto n = 0.0; // n is double because 0.0 is double

auto n = 0; // n is int because 0 is int

Automatic type declaration becomes much more useful when dealing with complicated types.

Ex – Such as those in STL (Standard Template Library)

C++ 98 code might have this:

std::vector<double> scores

std::vector<double> :: iterator mv = socres.begin();

C++ 11 allows you to write this instead:

std::vector<double> scores

auto mv = scores.begin();

THANK YOU