C (Basic) Data Types

different data representations need different types in programming
In C, data type categorized as:

1. Primitive Types in ANSI C (C89)/ISO C (C90) - char, short, int, float and double.
2. Primitive Types added to ISO C (C99) - long long
3. User Defined Types – struct, union, enum and typedef (will be discussed in separate session).
4. Derived Types – pointer, array and function pointer (will be discussed in separate session).
## C BASIC (DATA) TYPES

<table>
<thead>
<tr>
<th>Type</th>
<th>Size in Bits</th>
<th>Comments</th>
<th>Other Names</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>char</strong></td>
<td>≥ 8</td>
<td>• <code>sizeof()</code> will give the size in units of <code>char</code>s.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• need not be 8-bit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The number of bits is given by the CHAR_BIT macro in the limits.h header.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Integer operations can be performed portably only for the range: 0 ~ 127 (2^8 / 2).</td>
<td></td>
</tr>
<tr>
<td><strong>signed char</strong></td>
<td>Same as <code>char</code> but guaranteed to be signed</td>
<td>• Can store integers in the range: -127 ~ 127 (2^8) portably.</td>
<td></td>
</tr>
<tr>
<td><strong>unsigned char</strong></td>
<td>Same as <code>char</code> but guaranteed to be unsigned.</td>
<td>• Can store integers in the range: 0 ~ 255 (2^8) portably.</td>
<td></td>
</tr>
</tbody>
</table>

---

[char type program example](www.tenouk.com)
C BASIC (DATA) TYPES

- A sample output.
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size Comparison</th>
<th>Features</th>
<th>Examples</th>
</tr>
</thead>
</table>
| **short**       | ≥ 16, ≥ size of `char` | - Can store integers in the range: -32767 ~ 32767 \((2^{16} / 2)\) portably.  
- Reduce memory usage though the resulting executable may be larger and probably slower as compared to using `int`. | `short int`, `signed short`, `signed short int` |
| **unsigned short** | Same as `short` but unsigned | - Can store integers in the range: 0 ~ 65535 \((2^{16})\) portably.  
- Used to reduce memory usage though the resulting executable may be larger and probably slower as compared to using `int`. | `unsigned short int` |
| **int**         | ≥ 16, ≥ size of `short` | - Basic signed integer type.  
- Represent a typical processor’s data size which is word-size  
- An integral data-type.  
- Can store integers in the range: -32767 ~ 32767 \((2^{16} / 2)\) portably. | `signed`, `signed int` |
| **unsigned int** | Same as `int` but unsigned. | - Can store integers in the range: 0 ~ 65535 \((2^{16})\) portably. | `unsigned` |
C BASIC (DATA) TYPES

- A sample output.

```
short value is 10 with 2 bytes in size.
short int value is -20 with 2 bytes in size.
signed short value is 30 with 2 bytes in size.
signed short int value is -40 with 2 bytes in size.
unsigned short value is 50 with 2 bytes in size.
unsigned short int value is 60 with 2 bytes in size.
int value is 70 with 4 bytes in size.
signed value is -80 with 4 bytes in size.
signed int value is 90 with 4 bytes in size.
unsigned value is -100 with 4 bytes in size.
unsigned int value is 110 with 4 bytes in size.
Press any key to continue . . .
```
### C BASIC (DATA) TYPES

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size Information</th>
<th>Details</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>≥ 32, ≥ size of int</td>
<td>• long signed integer type.</td>
<td>long int, signed long, signed long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Can store integers in the range: -2147483647 ~ 2147483647 (2^{32} / 2) portably.</td>
<td></td>
</tr>
<tr>
<td>unsigned long</td>
<td>Same as long but unsigned</td>
<td>• Can store integers in the range: 0 ~ 4294967295 (2^{32}) portably.</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>float</td>
<td>≥ size of char</td>
<td>• Used to reduce memory usage when the values used do not vary widely.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The format used is implementation defined and unnecessarily obeys the IEEE 754 single-precision format.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• unsigned cannot be specified.</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>≥ size of float</td>
<td>• Typical floating-point data type used by processor.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The format used is implementation defined and unnecessarily obeys the IEEE 754 double-precision format.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• unsigned cannot be specified.</td>
<td></td>
</tr>
<tr>
<td>long double</td>
<td>≥ size of double</td>
<td>• unsigned cannot be specified.</td>
<td></td>
</tr>
</tbody>
</table>
C BASIC (DATA) TYPES

- A sample output.

![A sample output](image.png)
### C BASIC (DATA) TYPES

<table>
<thead>
<tr>
<th>Type</th>
<th>Size in Bits</th>
<th>Comments</th>
<th>Other Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive Types added to ISO C (C99)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long long</td>
<td>≥ 64, ≥ size of long</td>
<td>Can store integers in the range: - 922337203685477580 7 ~ 922337203685477580 7 (2^{64} / 2) portably.</td>
<td>long long int, signed long long, signed long long int</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>Same as long long, but unsigned.</td>
<td>Can store integers in the range: 0 ~ 184467440737095516 15 (2^{64}) portably.</td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>

**Long long int type program example**

[www.tenouk.com](http://www.tenouk.com)
C BASIC (DATA) TYPES

- A sample output.

```c
long long value is -10000 with 8 bytes in size.
long long int value is 200000 with 8 bytes in size.
signed long long value is -3000000 with 8 bytes in size.
signed long long int value is 40000000 with 8 bytes in size.
unsigned long long value is 500000000 with 8 bytes in size.
unsigned long long int value is 12345678 with 8 bytes in size.
Press any key to continue . . .
```
## C BASIC (DATA) TYPES

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>intmax_t</strong></td>
<td>Signed integer types capable of representing any value of any signed integer type.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- It is a typedef that represents the signed integer type with the largest possible range.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- If you want an integer with the widest range possible on the platform on which it is being used.</td>
<td></td>
</tr>
<tr>
<td><strong>uintmax_t</strong></td>
<td>Unsigned integer types capable of representing any value of any unsigned integer type.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- It is a typedef that represents the unsigned integer type with the largest possible range.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- If you want an integer with the widest range possible on the platform on which it is being used.</td>
<td></td>
</tr>
</tbody>
</table>

*Not supported by MSVC++ < 2012*
C BASIC (DATA) TYPES

- A sample output.

![Console program output](image)
Unfortunately this is not supported by MSVC++ < 2012

`inttypes.h` vs. `stdint.h`: The C99 standard says that `inttypes.h` includes `stdint.h`, so there's no need to include `stdint.h` separately in a standard environment.

Some implementations have `inttypes.h` but not `stdint.h`.

VS/VC++ users may want to use `msinttypes`.

Other references,

2. [http://pubs.opengroup.org/onlinepubs/007904975/basedefs/stdint.h.html](http://pubs.opengroup.org/onlinepubs/007904975/basedefs/stdint.h.html)
C BASIC (DATA) TYPES

- Actual size of integer types varies by implementation: Windows, Linux, BSD etc.
- The only guarantee is that the long long is not smaller than long, which is not smaller than int, which is not smaller than short.

long long > long > int > short

- int should be the integer type that the target processor is most efficient working with. For example, all types can be 64-bit.
- Actual size of floating point types also varies by implementation.
- The only guarantee is that the long double is not smaller than double, which is not smaller than float.

long double > double > float

- The 32-bit and 64-bit IEEE 754 floating point formats should be used.
The boolean (true/false) type is \texttt{bool} defined in \texttt{stdbool.h}.

The \texttt{stdbool.h} type also defines a few useful identifiers as macros: \texttt{bool} is defined as \texttt{Bool}, true as \texttt{1}, false as \texttt{0}.

Additionally, \texttt{__bool_true_false_are_defined} is defined as \texttt{1}.

The \texttt{Bool} type and \texttt{stdbool.h} header did not exist in pre-1999 versions of the standard.
C BASIC (DATA) TYPES

Size and pointer difference types

- Separate `size_t` and `ptrdiff_t` types to represent memory-related quantities.

- Existing types were inadequate, because their size is defined according to the target processor's arithmetic capabilities, not the memory capabilities, such as the address space availability.

- Both of these types are defined in the `stddef.h` header file (cstddef in C++).

- `size_t` is used to represent the maximum size of any object (including arrays) in the particular implementation.

- An unsigned integer type used to represent the sizes of objects.

 programmed example

www.tenouk.com, ©
C BASIC (DATA) TYPES

- A sample output.

![Console program output](image-url)
C BASIC (DATA) TYPES

- Used as the return type of the `sizeof()` operator.
- The maximum size of `size_t` is provided via `SIZE_MAX`, a macro constant which is defined in the `stdint.h` header file (`cstdint` in C++).
- It is guaranteed to be at least 65535.
- `ptrdiff_t` is used to represent the difference between pointers.
- Is the signed integer type of the result of subtracting two pointers.
- The type's size is chosen so that it could store the maximum size of a theoretically possible array of any type.
- On a 32-bit system `ptrdiff_t` will take 32 bits and on a 64-bit one - 64 bits and it is portable.

[size_t and ptrdiff_t: a story](www.tenouk.com)  [ptrdiff_t program example](www.tenouk.com)
C BASIC (DATA) TYPES

- A sample output.
Information about the actual properties, such as size, of the basic arithmetic types, is provided via macro constants in two header files,

1) `limits.h` header (`
climits`
in C++) defines macros for integer types.

2) `float.h` header (`
cfloat`
in C++) defines macros for floating-point types.

The actual values depend on the implementation.
Fixed width integer types

- C99 standard includes definitions of several new integer types to enhance programs’ portability.
- Existing basic integer types were considered inadequate; because their actual sizes are implementation defined and may vary across different systems.
- The new types are especially useful in embedded environments where hardware supports limited to several types and varies from system to system.
- All new types are defined in `inttypes.h` (cinttypes in C++) and `stdint.h` (cstdint in C++) header files.
- The types can be grouped into the following categories:
C BASIC (DATA) TYPES

- **Exact width** integer types - are guaranteed to have the same number $N$ of bits across all implementations. Included only if it is available in the implementation.

- **Least width** integer types - are guaranteed to be the smallest type available in the implementation, that has at least specified number $N$ of bits. Guaranteed to be specified for at least $N=8, 16, 32, 64$.

- **Fastest** integer types - are guaranteed to be the fastest integer type available in the implementation, that has at least specified number $N$ of bits. Guaranteed to be specified for at least $N=8, 16, 32, 64$.

- **Pointer** integer types - are guaranteed to be able to hold a pointer.

- **Maximum width** integer types - are guaranteed to be the largest integer type in the implementation.
The following table summarizes the types and the interface to acquire the implementation details (N refers to the number of bits).

<table>
<thead>
<tr>
<th>Type category</th>
<th>Signed types</th>
<th>Unsigned types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Type</td>
<td>Min value</td>
</tr>
<tr>
<td>Exact width</td>
<td>int(N_t)</td>
<td>INT(N_MIN)</td>
</tr>
<tr>
<td>Least width</td>
<td>int_least(N_t)</td>
<td>INT_LEAST(N_MIN)</td>
</tr>
<tr>
<td>Fastest</td>
<td>int_fast(N_t)</td>
<td>INT_FAST(N_MIN)</td>
</tr>
<tr>
<td>Pointer</td>
<td>intptr_t</td>
<td>INTPTR_MIN</td>
</tr>
<tr>
<td>Maximum width</td>
<td>intmax_t</td>
<td>INTMAX_MIN</td>
</tr>
</tbody>
</table>
### USER DEFINED (DATA) TYPES

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Size</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>struct</strong></td>
<td>≥ sum of size of each member</td>
<td>An aggregate type which can contain more than one different types.</td>
</tr>
</tbody>
</table>

**tag or label is optional**

```c
typedef struct
{
    int x;
    int SomeArray[100];
} MyFoo;

int main()
{
    MyFoo structVar;
    return 0;
}
```

```c
struct theEmployee {  
    int age;
    double salary;
    char department;
    char name[15];
    char address[5][25];
};
struct theEmployee workerRec;
```

```c
struct newPoint {  
    short xPoint;
    short yPoint;
} justPoint;
```

justPoint thePoint;
### USER DEFINED (DATA) TYPES

<table>
<thead>
<tr>
<th>union</th>
<th>≥ size of the largest member</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>An aggregate type which can contain more than one other types. <strong>union</strong> uses <strong>shared memory space</strong> compared to <strong>struct</strong>, so only one member can be accessed at one time.</td>
</tr>
</tbody>
</table>

```c
union someData
{
    int pNum;
    float qNum;
    double rNum;
};
union someData simpleData;
union OtherData{
    char aNum;
    int xNum;
    float fNum;
} simpleData;
simpleData saveData;
```
Enumerations are a separate type from `int`s, though they are mutually convertible. Used to declare identifiers as constants in an ordered manner.

<table>
<thead>
<tr>
<th>enum</th>
<th>(\geq) size of <code>char</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enum</code></td>
<td></td>
</tr>
<tr>
<td><code>ndays</code></td>
<td><code>{Mon, Tue, Wed, Thu, Fri, Sat, Sun}</code></td>
</tr>
<tr>
<td><code>ndays</code></td>
<td><code>ndayCount</code></td>
</tr>
<tr>
<td><code>trafficDirection</code></td>
<td><code>north, south, east, west</code></td>
</tr>
<tr>
<td><code>trafficDirection</code></td>
<td><code>newDirection</code></td>
</tr>
<tr>
<td><code>cColor</code></td>
<td><code>{red = 2, green, blue, black}</code></td>
</tr>
<tr>
<td><code>cColor</code></td>
<td><code>ccolorCode</code></td>
</tr>
</tbody>
</table>

```c
enum ndays {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

/* Creates enum days type, which the identifiers are set automatically to the integers 0 to 6. */
enum ndays ndayCount;
enum trafficDirection{
    north,
    south,
    east,
    west
};
enum cColor = {red = 2, green, blue, black};
Enum cColor ccolorCode;
```
### typedef

<table>
<thead>
<tr>
<th><strong>typedef</strong></th>
<th>same as the type; being given a new name</th>
<th><strong>typedef</strong> used to give new identifier names or alias (to simplify the long identifier names), normally used for aggregate defined types.</th>
</tr>
</thead>
</table>

```c
typedef unsigned char BYTE; /* Declares BYTE to be a synonym for unsigned char */
typedef float FLOAT; /* Declares FLOAT (uppercase letter) to be a synonym for unsigned float (lowercase) */
```

### tag or label is optional

```c
typedef struct simpleData
{    int nData;
    char cData;
} newNameType;
Or

typedef struct {    int nData;
    char cData;} newNameType;
newNameType strctType;
```

```c
typedef struct TOKEN_SOURCE {
    CHAR SourceName[8];
    LUID SourceIdentifier;
} TOKEN_SOURCE, *PTOKEN_SOURCE;
```

```c
TOKEN_SOURCE newToken;
```

```c
typedef union unData{
    double lngSalary;
    int nDay;
}newUnType;
newUnType lntotalSalary;
```

```c
typedef enum DayNames { Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday, Saturday, Sunday
} Weekdays;
```

```c
Weekdays dayOfWeek;
```
## DERIVED (DATA) TYPES

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Note</th>
</tr>
</thead>
</table>
| type* (a pointer) | ≥ size of `char` | - Hold the memory address which point to the actual data/value.  
- 0 address always represents the null pointer (an address where no data can be placed), irrespective of what bit sequence represents the value of a null pointer.  
- Pointers to different types will have different sizes. So they are not convertible to one another.  
- Even in an implementation which guarantees all data pointers to be of the same size, function pointers and data pointers are in general incompatible with each other.  
- For functions taking a variable number of arguments, the arguments passed must be of appropriate type. |

```c
char *ptoChar;
char csimpleChr = 'T';
char *chptr;
// assignment
chptr = &csimpleChr;
```

```c
int iNumber = 20;
int *imyPtr = &iNumber;
```
**DERIVED (DATA) TYPES**

| **Type [integer]** | **≥ integer × size of type** | **Use to declare a variable with collection of identical properties or types.**  
|**an array**| |**Simplify variable declaration.**  
| | |**In a declaration which also initializes the array (including a function parameter declaration), the size of the array (the integer) can be omitted, which is called unsized.**  
| | |**type [] is not the same as type*. Only under some circumstances one can be converted to the other.**  

### Examples

<table>
<thead>
<tr>
<th><strong>C Code</strong></th>
<th><strong>C Code</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int fstudentNumber[3] = {4,7,1};</code></td>
<td><code>char cName1[] = {'a','r','r','a','y'};</code></td>
</tr>
<tr>
<td><code>int nrowandColumn[1][2] = {34, 21};</code></td>
<td><code>char cName2[] = {&quot;array&quot;};</code></td>
</tr>
<tr>
<td><code>int nlongHeightWidth[3][4][5] = 0;</code></td>
<td><code>char cName3[6] = &quot;array&quot;;</code></td>
</tr>
<tr>
<td><code>int nrowCol[2][3] = {4,2,3,7,2,8};</code></td>
<td></td>
</tr>
</tbody>
</table>
**DERIVED (DATA) TYPES**

<table>
<thead>
<tr>
<th>type (comma-delimited list of types/declarations)</th>
<th>—</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a function pointers)</td>
<td>• allow referencing functions with a particular signature.</td>
</tr>
<tr>
<td></td>
<td>• Function pointers are invoked by name just like normal function calls. Function pointers are separate from pointers and <strong>void</strong> pointers.</td>
</tr>
</tbody>
</table>

```c
/* two arguments function pointer */
int (* fptr) (int arg1, int arg2)

/* to store the address of the standard function stdFunct in the variable myIntFunct */
int (*myIntFunct)(int) = stdFunct;
```
End of the C data types