CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Literals & Constants

Literals

Constants
A Simple C Program

Literal Constants

Example

```c
unsigned int a;
unsigned int c;
#define b 2

void main(void)
{
    a = 5;
    c = a + b;
    printf("a=%d, b=%d, c=%d\n", a, b, c);
}
```
Literal Constants

Definition

A literal or a literal constant is a value, such as a number, character or string, which may be assigned to a variable or a constant. It may also be used directly as a function parameter or an operand in an expression.

• Literals
  – Are "hard-coded" values
  – May be numbers, characters or strings
  – May be represented in a number of formats (decimal, hexadecimal, binary, character, etc.)
  – Always represent the same value (5 always represents the quantity five)
Constant vs. Literal

What's the difference?

• Terms are used interchangeably in most programming literature

• A literal is a constant, but a constant is not a literal
  - `#define MAXINT 32767`
  - `const int maxint = 32767;`

• For purposes of this presentation:
  - **Constants** are labels that represent a literal
  - **Literals** are values, often assigned to symbolic constants and variables
Literal Constants

• Four basic types of literals:
  – Integer
  – Floating Point
  – Character
  – String

• Integer and Floating Point are *numeric type* constants:
  – Commas and spaces are not allowed
  – Value cannot exceed type bounds
  – May be preceded by a plus or minus sign
Integer Literals
Decimal (Base 10)

- Cannot start with 0 (except for 0 itself)
- Cannot include a decimal point
- Valid Decimal Integers:
  0  5  127  -1021  65535

- Invalid Decimal Integers:
  32,767  25.0  1,024  0552
Integer Literals

Hexadecimal (Base 16)

- Must begin with \texttt{0x} or \texttt{0X}
- May include digits 0-9 and A-F / a-f
- Valid Hexadecimal Integers:
  \[
  \texttt{0x\ 0X1\ 0xA2B\ BEEF}
  \]
- Invalid Hexadecimal Integers:
  \[
  \texttt{5.3\ EA12\ E8\ 53h}
  \]
Integer Literals
Octal (Base 8)

- Must begin with 0
- Only include digits 0 - 7
- Valid Octal Integers:
  0  01  012  073125
- Invalid Octal Integers:
  05.3  0o12  080  530
Integer Literals

Binary (Base 2)

• Must begin with 0b or 0B
• May include digits 0 and 1
• Valid Binary Integers:

0b 0b1 0b01010011000001111

• Invalid Binary Integers:

0b1.0 01100 0b12 10b

ANSI C does not specify a format for binary integer literals. However, this notation is supported by almost all compilers.
Integer Literals

Qualifiers

• Like variables, literals may be qualified
• A suffix is used to specify the modifier
  – ‘U’ or ‘u’ for unsigned: 25u
  – ‘L’ or ‘l’ for long: 25L
  – 'F' or 'f' for float: 10f or 10.25F
• Suffixes may be combined: 0xF5UL
  – Note: U must precede L
Integer Literals
Unqualified

- Numbers without a suffix are assumed to be signed int
- Automatic promotion based on constant type/suffix:

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal</th>
<th>Octal/Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long long int</td>
</tr>
</tbody>
</table>
## Integer Literals

**Unqualified**

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal</th>
<th>Octal/Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>u/U</td>
<td>unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>I/L</td>
<td>long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td>LI/LL</td>
<td>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>
Floating Point Literals

Decimal (Base 10)

- Like integer literals, but the decimal point is allowed
- Exponential notation can be used: \( k \times 10^{\pm n} \) (e.g., \( 48E8 \))
- Valid Floating Point Literals: 0.56e-5, 10.4378, -48e8, 0.5, 10f
- Invalid Floating Point Literals: 0x5Ae-2, 02.41, F2.33
Character Literals

- Specified within single quotes (')
- May include any single printable character
- May include any single non-printable character using escape sequences (e.g. '\0' = NUL) (also called digraphs)
- Valid Characters: 'a', 'T', '\n', '5', '@', ' ' (space)
- Invalid Characters: 'me', '23', ''
A' + 127 →
String Literals

- Specified within double quotes ("")
- May include any printable or non-printable characters (using escape sequences)
- Terminated by a null character ‘\0’
- Valid Strings: "Microchip", "Hi\n", "PIC", "2500", "rob@microchip.com", "He said, "Hi"
- Invalid Strings: "He said, "Hi""
String Literals

- Strings are a special case of arrays
- The null character is automatically appended to the end of the string:

Example 1 – Wrong Way

```c
char color[3] = "RED";
Is stored as:  
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
```

NOT a complete string – no '\0' at end

Example 2 – Right Way

```c
char color[] = "RED";
Is stored as:  
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
color[3] = '\0'
```
String Literals
How to Include Special Characters in Strings

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\000</td>
<td>Octal number</td>
</tr>
<tr>
<td>\xhh</td>
<td>Hexadecimal</td>
</tr>
</tbody>
</table>

- Equivalent characters
  - '\0', '\x0'
  - '\41', '\x21', '!!'
  - '\144', '\x64', 'd'
Symbolic Constants

Definition

A constant or a symbolic constant is a label that represents a literal. Anywhere the label is encountered in code, it will be interpreted as the value of the literal it represents.

• Constants
  – Once assigned, never change value
  – Make development changes easy
  – Eliminate the use of "magic numbers"
  – Two types of constants
    • Text Substitution Labels
    • const Variables
Symbolic Constants

Text Substitution Labels Using `#define`

- Defines a text substitution label

**Syntax**

```
#define label text
```

- Each instance of `label` will be replaced with `text` by the **preprocessor** unless `label` is inside a string
- Requires no memory

**Example**

```
#define PI 3.14159
#define mol 6.02E23
#define MCU "PIC32MX320F128H"
#define COEF 2 * PI
```
Symbolic Constants

#define Gotchas

• Note: a `#define` directive is **NEVER** terminated with a semi-colon (;), unless you want that to be part of the text substitution.

Example

```c
#define MyConst 5;

c = MyConst + 3;
```

Maxwell James Dunne
Symbolic Constants
Constant Variables Using `const`

- Declaring constants can be done with `const`:

```
Example
const float pi = 3.141593;
```

- This variable is allocated in program memory, but it cannot be changed due to the `const` keyword.

- In the majority of cases, it is better to use `#define` for constants.
Symbolic Constants

Initializing Variables When Declared

• A constant declared with `const` may not be used to initialize a global or static variable when it is declared (though it may be used to initialize local variables...)

Example

```c
#define CONSTANT1 5
const CONSTANT2 = 10;

int variable1 = CONSTANT1;
int variable2;
// Cannot do: int variable2 = CONSTANT2
```
CMPE-013/L

Structs

Maxwell James Dunne
Structures

Structures are collections of variables grouped together under a common name. The variables within a structure are referred to as the structure’s members, and may be accessed individually as needed.

- Structures:
  - May contain any number of members
  - Members may be of any data type
  - Allow a group of related variables to be treated as a single unit, even if different types

Ease the organization of complicated data
Structures
Declaring

Syntax

```
struct StructName {
    type1 memberName1;
    ...
    typeN memberNameN;
};
```

Members are declared just like ordinary variables

Example

```
// Structure to handle complex numbers
struct Complex {
    float re;    // Real part
    float im;    // Imaginary part
};
```
Structures

Instantiating

**Syntax**

```c
struct StructName {
    type_1 memberName_1;
    ...
    type_n memberName_n;
} varName_1, ..., varName_n;
```

**Example**

```c
// Structure to handle complex numbers
struct Complex {
    float re;
    float im;
} x, y;  // Declare x and y of type complex
```
If `StructName` has already been defined:

```c
struct StructName varName1, ..., varName_n;
```

Example:

```c
struct Complex {
    float re;
    float im;
};

struct Complex x, y;  // Declare x and y of type complex
```
Structures

Accessing members

Syntax

```
struct VariableName . memberName
```

Example

```c
struct Complex {
    float re;
    float im;
} x, y; // Declare x and y of type `struct complex`

int main(void)
{
    x.re = 1.25; // Initialize real part of x
    x.im = 2.50; // Initialize imaginary part of x
    y = x; // Set struct y equal to struct x
    ...
```
Structures
Initialization

Syntax
If \texttt{StructName} has already been defined:

\begin{verbatim}
struct \texttt{StructName} \texttt{varName} = \{ \texttt{const}_1, \ldots, \texttt{const}_n \};
\end{verbatim}

Example
\begin{verbatim}
struct Complex {
    float re;
    float im;
};
...
struct Complex x = \{ 1.25, 2.50 \};
\end{verbatim}
Structures

Nesting Structures

Example

```c
struct point {
    float x;
    float y;
};

struct line {
    struct point a;
    struct point b;
};

int main(void)
{
    struct line m = {{{1.2, 7.6}, {38.5, 17.8}}};
    ...
}
```
Structures

Nesting Structures

Example

```c
struct point {
    float x;
    float y;
};

struct line {
    struct point a;
    struct point b;
};

int main(void)
{
    struct line m = {{{1.2, 7.6}, {38.5, 17.8}}};
    printf("Line (%f, %f) <-> (%f, %f)",
            m.a.x, m.a.y, m.b.x, m.b.y);
    ...
}
```
Structures
Arrays and Pointers with Strings

- Strings:
  - May be assigned directly to `char` array member only at declaration
  - May be assigned directly to a pointer to `char` member at any time

Example: Structure
```c
struct Strings {
    char a[4];
    char *b;
} str = {"Bad", "Good"};
```

Example: Initializing Members
```c
int main(void)
{
    struct Strings str;
    str.a[0] = 'B';
    str.a[1] = 'a';
    str.a[2] = 'd';
    str.a[3] = '\0';
    str.b = "Good";
}
```
**Structures**

Creating Arrays of Structures

**Syntax**

If `StructName` has already been defined:

```c
struct StructName arrName[n];
```

**Example**

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex a[3];
```
Structures

Initializing Arrays of Structures at Declaration

Syntax

If StructName has already been defined:

```c
struct StructName arrName[n] = {{list_1}, ..., {list_n}};
```

Example

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex a[3] = {{1.2, 2.5}, {3.9, 6.5}, {7.1, 8.4}};
```
Structures
Using Arrays of Structures

If \textit{arrName} has already been defined:

**Syntax**

\texttt{arrName[n].memberName}

**Example: Definitions**

```c
typedef struct {
    float re;
    float im;
} Complex;
...  
struct Complex a[3];
```

**Example: Usage**

```c
int main(void)
{
    a[0].re = 1.25;
    a[0].im = 2.50;
    ...
}
```
Structures
Creating a Pointer to a Structure

**Syntax**

If `StructName` has already been defined:

```c
struct  StructName  *ptrName;
```

**Example**

```c
struct  Complex  {
    float  re;
    float  im;
};
...
struct  Complex  *a;
```
Structures

How to Use a Pointer to Access Structure Members

If `ptrName` has already been defined:

Syntax

```
ptrName->memberName
```

Pointer must first be initialized to point to the address of the structure itself: `ptrName = &structVariable;`

Example: Definitions

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex x;
struct Complex *p;
```

Example: Usage

```c
int main(void)
{
    p = &x;
    // Set x.re = 1.25 via p
    p->re = 1.25;
    // Set x.im = 2.50 via p
    p->im = 2.50;
}
```
Structures
How to Pass Structures to Functions

Example

```c
struct Complex{
    float re;
    float im;
};

void Display(struct Complex x)
{
    printf("(%f + j%f)\n", x.re, x.im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(a);
    Display(b);
}
```
# Structures

How to Pass Structures to Functions

```c
struct Complex {
    float re;
    float im;
};

void Display(struct Complex *x)
{
    printf("(\%f + j\%f)\n", x->re, x->im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(&a);
    Display(&b);
}
```
Structures
How to Pass Structures to Functions

Example

typedef struct {
    float re;
    float im;
} Complex;

void Display(const struct Complex *x)
{
    printf("(\%f + j\%f)\n", x->re, x->im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(&a);
    Display(&b);
}
typedef
typedef

- Assign new names to existing datatypes
- Interpreted by the compiler (unlike `#define`)

**Syntax**

```c
typedef datatype typeName;
```

- `typedef int Length;`
- `typedef float single;`
typedef

How to Create a Structure Type with typedef

Syntax

```c
typedef struct StructTag optional {
  type_1 memberName_1;
  ...
  type_n memberName_n;
} TypeName;
```

Example

```c
// Structure type to handle complex numbers
typedef struct {
  float re; // Real part
  float im; // Imaginary part
} Complex;

Complex x;
```
typedef
Declaring structs

Example

```c
struct Complex {
    float re;
    float im;
};

typedef struct {
    float re;
    float im;
} Complex;

struct Complex {
    float re;
    float im;
} Complex;

typedef struct {
    float re;
    float im;
} Complex;
```
typedef struct Complex {
  float re;
  float im;
} Complex;