1 Introduction to C

- Developed in the early 1970s by Dennis Ritchie (Bell Labs).
- Unix written in C (a little assembly).
- C is a subset of C++, or C++ is a superset of C ("C with classes").
- Still very popular.

1.1 What's the Same?

- Basic data types
- Comments (most C compilers support `// inline comments`)
- Syntax
- Naming conventions

1.1.1 Basic data types

- `char`
- `int` (`short`, `long`)
- `double` (`float`)

1.1.2 Syntax

- Function definitions (parameter passing is different)
- Conditionals: `if`, `switch`
- Loops: `for`, `while`, `do--while`
- Arrays
- Abstract data type: `structs`, `unions`

1.1.3 `union` Example

```
union {
    uint32_t my_int;
    uint8_t  my_bytes[4];
} endian_tester;
endian_tester et;

et.my_int = 0x0a0b0c0d;
if( et.my_bytes[0] == 0x0a )
    printf( "I'm on a big-endian system\n" );
else
    printf( "I'm on a little-endian system\n" );
```

Reference: *Byte and Bit Order*, *Linux Journal*,
`http://www.linuxjournal.com/article.php?sid=6788`

1.2 What's Different?

- Compiler invocation (`cc` instead of `CC`, `gcc` instead of `g++`)
- I/O (input/output)
- Variables must be declared at the *top* of a function
- Parameter passing—pointers
- Strings are character arrays (or a pointer to `char`)
- Memory manipulation
- No generics (templates in C++)
- Macros (C Preprocessor, `cpp`)

1.3 I/O

I/O is function-based in C.

Operation	Functions
input	`scanf`, `read`
output	`printf`, `write`

`read` and `write` are low-level. May not used in this class.

Examples

Read a character

```
  scanf( "%c", &c );
```

Need address of variable (pointer)

Read an integer

```
  scanf( "%d", &i );
```

Print an integer

```
  printf( "%3d\n", i );
```

NOTE: The arguments to `scanf` and `sscanf` *must* be pointers! Common error when trying to read an integer: Using `scanf( "%d", i );` instead of `scanf( "%d", &i );`

1.3.1 Format characters

Table 1: C Format Conversion Characters

`%c`	character
`%d`	integer
`%e`	single precision —exponential
`%f`	single precision
`%g`	floating point—exponential if needed
`%o`	octal integer
`%u`	unsigned integer
`%x`	hexadecimal integer
`%hd`	short
`%ld`	long
`%lf`	double
`%s`	string
`%%`	literal `%`

Examples:
`%10.3lf`
`%20s`

The conversion characters `d`, `i`, `o`, `u`, and `x` may be preceded by `h` to indicate that a pointer to `short` rather than `int` appears in the argument list, or by `l` (letter ell) to indicate that a pointer to `long` appears in the argument list. Similarly, the conversion characters `e`, `f`, and `g` may be preceeded by `l` to indicate a pointer to `double` rather than `float` is in the argument list.

1.3.2 File I/O

All files are represented by one type: `FILE *`. `FILE *` is defined in `stdio.h`

	C++	C

header	`iostream`
	`(iostream.h)`
		`stdio.h`

input	`cin`	`stdin`
output	`cout`	`stdout`
error	`cerr`	`stderr`

1.4 Function Definition

Prototypes were an addition to the ANSI standard.

Consider the problem of displaying an integer with a message preceding it.

Table 2: C-style

```
void PrintInt( a, s )
int a;
char *s;
{
    printf( "%s: %d\n", s, a );
}
```

Table 3: ANSI C-style

```
void PrintInt( int a, char *s )
{
    printf( "%s: %d\n", s, a );
}
```

Does the order of arguments matter?

1.5 Parameter passing

All variables are passed by value or passed by pointer. Consider a function to swap two integers:

```
void Swap( int *a, int *b )
{
    int iTmp = *a;
    *a = *b;
    *b = iTmp;
}
```

Usage: `Swap( &i, &j );`

1.6 File Operation Code

Typical file operations: Open (input/output), Close, Read/Write.

1.6.1 Opening Files for Input

Table 4: C++

```
ifstream fIn;

fIn.open( fName, ios::in );
if( !fIn )
{
   cerr << "Unable to open: "
        << fName << endl;
   exit( -1 );
}
```

Table 5: C

```
FILE *fpIn;

fpIn = fopen( fName, "r" );
if( fpIn == NULL )
{
   printf( "Unable to open: %s\n",
             fName );
   exit( -1 );
}
```

Note: `fName` is the name of the file to open. The prototype for `exit()` is defined in `stdlib.h`.

1.6.2 Opening Files for Output

Table 6: C++

```
ofstream fOut;

fOut.open( fName, ios::out );
if( !fOut )
{
    cerr << "Unable to open: "
         << fName << endl;
    exit( -1 );
}
```

Table 7: C

```
FILE *fpOut;

fpOut = fopen( fName, "w" );
if( fpOut == NULL )
{
    printf( "Unable to open: %s\n",
             fName );
    exit( -1 );
}
```

Append to a file:

C++:

```
fOut.open( fName, ios::out | ios::app );
```

C:

```
fpOut = fopen( fName, "w+" );
```

1.6.3 Closing Files

	C++	C
input	`fIn.close();`	`fclose( fpIn );`
output	`fout.close();`	`fclose( fpOut );`

1.6.4 Example: Copy a File to Standard Output

The following program copies a file character by character to the standard output (stdout, the terminal), unless redirected.

```
#include <stdio.h>

int main( int argc, char **argv )
{
    FILE *fp;
    int  c;    //  not char!

    if( (fp = fopen(*++argv, "r")) != NULL )
    {
       while( (c = getc(fp)) != EOF )
          putc( c, stdout );
    }

    fclose( fp );
}
```

Note: `char **argv` same as `char *argv[]`
Usage of `getc()` and `putc()`and that the argument is an `int` not a `char`! Why?

1.7 String Manipulation

Strings are character arrays in C (no string class!). The standard string library functions are defined in `string.h`. Typical string manipulation functions: `strlen`, `strcat`, `strcmp`, etc.

We can read and write from/to strings using the function `sscanf` for input and `sprintf` output.

Read an integer: `sscanf( s, "%d", &i );`

Write an integer into a string: `sprintf( s, "%d", i );`

1.8 Dynamic Memory

Dynamically allocated memory is manipulated using *operators* in C++, and *functions* in C.

	C++	C
allocation	`new`	`malloc`, `alloc`, `calloc`
deallocation	`delete`	`free`

1.8.1 Example: One-Dimensional Array Manipulation Code

Declare/Allocate/Deallocate a one-dimensional integer array containing `N` elements.

	C++	C
declare	`int *pA;`	`int *pA;`
allocate	`pA = new int[N];`	`pA = (int *)malloc(N*sizeof(int));`
release	`delete [] pA;`	`free( (void *)pA );`

Note the use of casting when allocating/deallocating memory.

Treat the dynamically allocated array just as if it had been declared statically.

1.8.2 Example: Two-Dimensional Array Manipulation Code

Declare/Allocate/Deallocate a two-dimensional integer array containing `nRows` rows and `nColumns` columns elements.

C++

```
   // Declare
int **arr2D;

   // Allocate
arr2D = new int *[nRows];

for( int i = 0 ; i < nRows ; i++ )
{
    arr2D[i] = new int[nCols];
    if( arr2D == NULL )
    {
        ERROR MESSAGE
    }
}

   // Deallocate (release)
for( int i = 0 ; i < nRows ; i++ ) {
    delete arr2D[i];
}

delete [] arr2D;
```

C

```
   // Declare
int **arr2D;

   // Allocate
arr2D = (int **)malloc(nRows*sizeof(int *));
for( i = 0 ; i < nRows ; i++ )
{
    arr2D[i] = (int *)malloc(nCols*sizeof(int));
}

   // Deallocate (release)
for( i = 0 ; i < nRows ; i++ )
     free( (void *)(arr2D[i]) );
free( (void *)arr2D );
```

Treat the dynamically allocated array just as if it had been declared statically.

1.8.3 Example: List Manipulation

Singly-linked lists are the most fundamental data structure in most languages. C is not an exception.

```
struct Node
{
    int data;
    struct Node* next;
};

typedef struct Node* NodePtr;

NodePtr head = NULL;    /* in main  */
```

```
void AddNodeRecursive( NodePtr* h, int x )
{
    if( *h != NULL )
    {
        AddNodeRecursive( &(*h)->next, x );
             /*          ^^--------NOTE!  */
    }
    else
    {
        NodePtr n;

        n = (NodePtr)malloc( sizeof(struct Node) );
        n->info = x;
        n->next = NULL;

        *h = n;
    }
}
```

Why `AddNodeRecursive( &(*h)->next, x );`?
Why not `AddNodeRecursive( (*h)->next, x );`?

Usage

```
NodePtr head = NULL;    /* in main  */

AddNodeRecursive( &head, 2 );
AddNodeRecursive( &head, 4 );
```

1.9 Resources

The C Programming Language, Second edition, Kernighan and Ritchie, Prentice-Hall, 1988

C: An Advanced Introduction, Narain Gehani, Computer Science Press, 1985 (1994 more recent)