C Reference Notebook

Contents

Part I

Language

Language Basics

Characteristics of C

C is a general-purpose, procedural language developed by Dennis Ritchie in the 1970s at AT&T Bell Labs, designed for Unix system development. Key features include:

- **Source Code Portability**: Independent of specific hardware platforms.
- **Close to the Machine**: Efficient low-level operations.
- **Efficiency**: Optimized for performance.

C's design, influenced by BCPL and B, includes various data types. Its first description by Kernighan and Ritchie (K&R) became the standard reference. C's portability is enhanced by a core language with minimal hardware dependencies and a comprehensive standard library for functions like I/O and memory management.

C is widely used in system programming and embedded systems, as well as high-level applications like word processors and databases.

The Structure of C Programs

C programs are composed of functions, with `main()` as the entry point. Functions execute sequentially and can call others. Programs can use the standard library or custom functions, with portability considerations for nonstandard libraries.

Example structure of a C program:

```c
#include <stdio.h> // Preprocessor directive
double circularArea(double r); // Function prototype

int main() { // Main function
    double radius = 1.0, area = 0.0;
    printf("Areas of Circles\n\n");
    area = circularArea(radius);
    printf("%10.1f     %10.2f\n", radius, area);
    radius = 5.0;
    area = circularArea(radius);
    printf("%10.1f     %10.2f\n", radius, area);
    return 0;
}

double circularArea(double r) { // Calculates circle area
    const double pi = 3.1415926536;
    return pi * r * r;
}
```

Source Files

C programs consist of function definitions, global declarations, and preprocessing directives. Small programs use a single source file, while larger ones are split into multiple files, typically structured as follows:

1. **Preprocessor Directives**
2. **Global Declarations**
3. **Function Definitions**

Source files, labeled with the `.c` suffix, are modular and can be compiled separately. Functions are logically grouped, such as user interface functions.

Comments

Use comments generously in C programs for documentation. There are two types:

- **Block Comments**: `/* ... */` for multi-line comments.
- **Line Comments**: `//` for single-line comments (added in C99, also known as "C++-style").

Examples

- **Block Comment** within a line:

```
int open(const char *name, int mode, ... /* int permissions */);
```

- **Line Comment** in two-column format:

```
const double pi = 3.1415926536; // Pi is constant
```

> **Notes:**
> Inside strings or character constants, /* and // do not start comments:

```
printf("Comments in C begin with /* or //.\n");
```

> Nesting Block Comments: Not allowed. Use block comments to temporarily remove code containing line comments:

```
/* Temporarily removing two lines:
   const double pi = 3.1415926536; // Pi is constant
   area = pi * r * r               // Calculate the area
*/
```

> Conditional Preprocessor Directive: To remove sections containing block comments:

```
Copy code
#if 0
```

```
    const double pi = 3.1415926536; /* Pi is constant */
    area = pi * r * r;              /* Calculate the area */
#endif
```

Character Sets

C distinguishes between the **translation environment** (where source files are compiled) and the **execution environment** (where the compiled program runs). Each environment has its own character set:

- **Source Character Set**: Used in C source code.
- **Execution Character Set**: Interpreted by the running program.

Basic and Extended Character Sets

Both the source and execution character sets include:

- **Latin Alphabet**: A-Z, a-z
- **Decimal Digits**: 0-9
- **Punctuation Marks**: `!"#%&'()*+,-./:;<=>?[\]^_{|}~`
- **Whitespace Characters**: Space, horizontal tab, vertical tab, new line, form feed

The **basic execution character set** also includes:

- **Nonprintable Characters**: null (`\0`), alert (`\a`), backspace (`\b`), carriage return (`\r`)

Character codes may vary across implementations, with conditions:

- Basic characters must be representable in one byte.
- The null character's byte has all bits set to 0.
- Digits increment by one sequentially.

Wide Characters and Multibyte Characters

To accommodate diverse languages, C supports:

- **Wide Characters** (`wchar_t`): Fixed bit width for each character, suitable for large character sets.
- **Multibyte Characters**: Variable length, used for complex scripts (e.g., UTF-8).

C provides functions like `wctomb()` for conversions between wide and multibyte characters. Example:

```
wchar_t wc = L'\x3B1'; // Greek letter alpha (α)
char mbStr[10] = "";
int nBytes = wctomb(mbStr, wc); // Converts to multibyte
```

Digraphs and Trigraphs

C provides **digraphs** and **trigraphs** as alternative representations for certain punctuation marks, useful when specific characters are not available on all keyboards.

Digraphs

Digraphs are two-character sequences that represent single characters. They are not interpreted within character constants or string literals.

Digraph	Equivalent
``<:	``[
``:>	``]
``<%	``{
``%>	``}
``%:	``#
``%:%:	``##

Example with digraphs:

```c
int arr<::> = <% 10, 20, 30 %>;
printf("The second array element is <%d>.\n", arr<:1:>);
```

Trigraphs

Trigraphs are three-character sequences starting with two question marks and represent single characters.

Trigraph	Equivale
??(	[
??)	]
??<	{
??>	}
??=	#
??/	\
??!	\
??'	^
??-	~

Trigraphs are translated in the first phase of compilation and can appear anywhere, including character constants, string literals, and comments.

Example with trigraphs:

```
printf("Cancel| `???(y/n)` ");
```

To prevent a trigraph interpretation, escape the question marks:

```
printf("Cancel\?\?\?(y/n) ");
```

Additional Punctuation Substitutes The header file iso646.h defines macros for logical and bitwise operators, e.g., and for && and xor for ^.

Identifiers

In C programming, identifiers refer to the names of variables, functions, macros, structures, and other objects defined within a program. Here are the rules and considerations for identifiers:

- Identifiers can contain:
    - Letters from the basic character set (a–z, A–Z), and they are case-sensitive.
    - The underscore character `_`.
    - Decimal digits (0–9), but the first character of an identifier cannot be a digit.
    - Universal character names representing letters and digits from other languages, as defined in Annex D of the C standard.
- Multibyte characters in identifiers depend on the C implementation.

> **37 keywords** are reserved in C and cannot be used as identifiers:

`auto`	`enum`	`restrict`	`unsigned`
`break`	`extern`	`return`	`void`
`case`	`float`	`short`	`volatile`
`char`	`for`	`signed`	`while`
`const`	`goto`	`sizeof`	`_Bool`
`continue`	`if`	`static`	`_Complex`
`default`	`inline`	`struct`	`_Imaginary`
`do`	`int`	`switch`	
`double`	`long`	`typedef`	
`else`	`register`	`union`	

- Examples of valid identifiers: `x`, `dollar`, `Break`, `error_handler`, `scale64`.
- Examples of invalid identifiers: `1st_rank`, `switch`, `y/n`, `x-ray`.
- If supported by the compiler, universal character names like `α` can also be valid identifiers:

```c
double α = 0.5;
```

Identifier Name Spaces

All identifiers in C belong to one of four distinct name spaces:

- Label names
- Tags: Identify structure, union, and enumeration types.
- Structure or union member names: Each structure or union defines a separate name space for its members.
- Ordinary identifiers: All other identifiers fall into this category.

Identifiers in different name spaces can share the same name without conflict. For example:

```c
struct pin {
    char pin[16];  /* ... */
};

_Bool check_pin( struct pin *pin ) {
    int len = strlen( pin->pin );
    /* ... */
}
```

In this example, pin serves as:

- A structure tag (struct pin).
- A member name within the structure (pin->pin).
- A parameter name in the function check_pin.

Identifier Scope

The scope of an identifier refers to the part of the translation unit where the identifier is meaningful. In other words, it defines where in the program the identifier can be "seen". The type of scope is determined by the location of the identifier's declaration, except for labels which always have function scope. There are four types of scope in C:

- **File Scope**: An identifier declared outside all blocks and parameter lists has file scope. It can be used anywhere after its declaration and up to the end of the translation unit.

- **Block Scope**: Identifiers declared within a block (excluding labels) have block scope. They are valid from their declaration to the end of the smallest containing block, which is often the body of a function definition. In C99, declarations do not need to precede all statements within a function block.

- **Function Prototype Scope**: Parameter names in a function prototype have function prototype scope. They are only significant within the prototype itself and can be omitted.

- **Function Scope**: Labels have function scope, meaning they are accessible throughout the function block where they are defined, even if nested.

The scope of an identifier generally starts after its declaration. However, type names (tags) of structure, union, and enumeration types, as well as enumeration constants, have their scope immediately after their declaration. This allows them to be referenced within the declaration itself.

In the example below:

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

void printNode( const struct Node *ptrNode );

int printList( const struct Node *first ) {
    struct Node *ptr = first;
    while( ptr != NULL ) {
        printNode( ptr );
        ptr = ptr->next;
    }
}
```

- Identifiers Node, next, printNode, and printList have file scope.
- Parameter ptrNode in printNode has function prototype scope.
- Variables first and ptr have block scope within printList.
  I dentifiers can be re-declared within their scope, hiding outer declarations of the same identifier. For instance:

```
Copy code
double x;                  // Declare a variable x with file scope
long calc( double x );  // Declare a new x with function prototype scope

int main() {
    long x = calc( 2.5 ); // Declare a long variable x with block scope
    if( x < 0 ) {
        float x = 0.0F;  // Declare a new float variable x with block
scope
        /*...*/
    }
    x *= 2;  // Here x refers to the long variable again
    /*...*/
}
```

In this example, the `long` variable `x` declared in `main()` hides the global `double` variable `x`. Within the conditional block, `x` refers to the newly declared `float` variable, which in turn hides the `long` variable `x`.

How the C Compiler Works

Once you've written a source file using a text editor, you can invoke a C compiler to translate it into machine code. The compiler operates on a **translation unit**, which consists of a source file and all the header files referenced by `#include` directives. If the compiler finds no errors, it generates an object file containing the corresponding machine code. Object files usually have the suffix `.o` or `.obj`. The compiler may also generate an assembler listing.

Object files are also called **modules**. A library, such as the C standard library, contains compiled, easily accessible modules of standard functions.

The compiler translates each translation unit of a C program (each source file with any included headers) into a separate object file. Then, the compiler invokes the **linker**, which combines the object files and any used library functions into an executable file. Figure 1-1 illustrates this process.

The C Compiler’s Translation Phases

The compiling process involves eight logical steps, which can be combined by a compiler:

1. **Character Conversion**: Characters from the source file are converted into the source character set. End-of-line indicators are replaced, and trigraph sequences are converted to single characters.
2. **Line Continuation**: Backslashes followed by a newline are deleted, allowing directives to continue on the next line.
3. **Tokenization**: The source file is broken into preprocessor tokens and whitespace. Comments are treated as a single space.
4. **Preprocessing**: Preprocessor directives are executed, and macro calls are expanded. This step applies to included files as well.
5. **Character Set Conversion**: Characters in constants and literals are converted to the execution character set.
6. **String Concatenation**: Adjacent string literals are concatenated into a single string.
7. **Compilation**: The compiler analyzes the tokens and generates machine code.
8. **Linking**: The linker resolves references to external objects/functions and generates the executable file. Unresolved references are taken from libraries.

Tokens

A token can be a keyword, identifier, constant, string literal, or symbol. Symbols are punctuation characters functioning as operators or having syntactic importance. For example, in the statement `printf("Hello, world.\n");`, the tokens are:

- `printf`
- `(`
- `"Hello, world.\n"`
- `)`
- `;`

During preprocessing, tokens have slight differences, such as `#include` recognizing `<filename>` and `"filename"`, and no distinction between integer and floating-point constants. Parsing follows a principle that ensures each character is part of a valid token.