C PROGRAMING

Ketan Kore

Sunbeam Infotech

Preprocessor Directives

- Preprocessor is part of C programming toolchain/SDK.
  - Removes comments from the source code.
  - Expand source code by processing all statements starting with #.
  - Executed before compiler

- All statements starting with # are called as preprocessor directives.
  - Header file include
    - #include
  - Symbolic constants & Macros
    - #define
  - Conditional compilation
    - #if, #else, #elif, #endif
    - #ifdef #ifndef
  - Miscellaneous
    - #pragma, #error

#include

- #include includes header files (.h) in the source code (.c).

- #include <file.h>
  - Find file in standard include directory.
  - If not found, raise error.

- #include "file.h"
  - File file in current source directory.
  - If not found, find file in standard include directory.
  - If not found, raise error.

#define (Symbolic constants)

- Used to define symbolic constants.
  - #define PI 3.142
  - #define SIZE 10

- Predefined constants
  - __LINE__
  - __FILE__
  - __DATE__
  - __TIME__

- Symbolic constants and macros are available from there declaration till the end of file. Their scope is not limited to the function.

#define (Macro)

- Used to define macros (with or without arguments)
  - #define ADD(a, b) (a + b)
  - #define SQUARE(x) ((x) * (x))
  - #define SWAP(a,b,type) { type t = a; a = b; b = t; }
- Macros are replaced with macro expansion by preprocessor directly.
  - May raise logical/compiler errors if not used parenthesis properly.
- Stringizing operator (#)
  - Converts given argument into string.
  - #define PRINT(var) printf(#var " = %d", var)
- Token pasting operator (##)
  - Combines argument(s) of macro with some symbol.
  - #define VAR(a,b) a##b

#define

- Functions
  - Function have declaration, definition and call.
  - Functions are called at runtime by creating FAR on stack.
  - Functions are type-safe.
  - Functions may be recursive.
  - Functions called multiple times doesn't increase code size.
  - Functions execute slower.
  - For bigger reusable code snippets, functions are preferred.

- Macros
  - Macro definition contain macro arguments and expansion.
  - Macros are replaced blindly by the processor before compilation
  - Macros are not type-safe.
  - Macros cannot be recursive.
  - Macros (multi-line) called multiple times increase code size.
  - Macros execute faster.
  - For smaller code snippets/formulas, macros are preferred.

Conditional compilation

- As preprocessing is done before compilation, it can be used to control the source code to be made available for compilation process.

- The condition should be evaluated at preprocessing time (constant values).

- Conditional compilation directives
  - #if, #elif, #else, #endif
  - #ifdef, #ifndef
  - #undef

```c
#define VER 1
int main() {
    #ifndef VER
        #error "VER not defined"
    #endif
    #if VER == 1
        printf("This is Version 1.\n");
    #elif VER == 2
        printf("This is Version 2.\n");
    #else
        printf("This is 3+ Version.\n");
    #endif
    return 0;
}
```

Structure

- Structure is a user-defined data type.
- Structure stores logically related (similar or non-similar) elements in contiguous memory location.
- Structure members can be accessed using "." operator via struct variable.
- Structure members can be accessed using "->" operator via struct pointer.
- Size of struct = Sum of sizes of struct members.
- If struct variable initialized partially at its point of declaration, remaining elements are initialized to zero.

```
// struct data-type declaration (global or local)
struct emp {
    int empno;
    char ename[20];
    double sal;
};
// struct variable declaration
struct emp e1 = {11, "John", 20000.0};
// print struct members
printf("%d%s%lf", e1.empno, e1.ename, e1.sal);
```

Struct – User defined data-type

- int a = 10;
- printf("%d", a);
- scanf("%d", &a);
- int *p = &a;
- printf("%d", *p);

- fun1(a);
- void fun1(int x) { … }
- fun2(&a);
- void fun1(int *p) { … }
- int arr[3] = {11, 22, 33};
- for(int i=0; i<3; i++)
  printf("%d", arr[i]);

- struct emp e = { 11, "John" };
- printf("%d, %s", e.empno, e.ename);
- scanf("%d%s", &e.empno, &e.ename);
- struct emp *p = &e;
- printf("%d, %s", p->empno, p->ename);
- printf("%d, %s", (*p).empno, (*p).ename);
- fun1(e);
- void fun1(struct emp x) { … }
- fun1(&e);
- void fun2(struct emp *p) { … }
- struct emp arr[3] = { {…}, {…}, {…} };
- for(int i=0; i<3; i++)
  printf("%d", arr[i].empno);

Thank you!

Ketan Kore<ketan.kore@sunbeaminfo.com>