Functions and Advanced Program Structure

Devendra Tewari

2024-10-09

Introduction to Functions

- Useful for program structuring
- Make program more modular
- Should be as generally applicable as possible
- Should encapsulate implementation as best as possible
- Cannot be nested (unlike in Pascal)

Function Declaration

- Functions need to be declared before use
- The compiler matches the declaration with the syntax of usage and definition to see if they match
  - the return type should be the same
  - the parameters should be the same type (not name)

```
return-type function-name (argument declarations);
```

Function Definition

```
return-type function-name (argument declarations) {
  declarations and statements
  return statement returns a value of type return-type
}
```

- The return-type can be void or any other type; if not specified, it defaults to `int`
- A `return` statement is optional and can be used to return a value to the caller, the caller may ignore this value

```
return expression;
```

Variable Declaration

- Anywhere in a C source file
- Inside a function

```
int main() {
  int a, b;
}
```

- Inside any code block

```
{
  int a, b;
}
```

Variable Declaration Example

```
int a;

int main() {
  int a = 10; // "a" is local to main
  print(); // prints "a: 0"
}

int print() {
  printf("a: %d\n", a);
}
```

External variables

- Variable defined outside functions or in other source files are external
  - *Definition* indicates the place where a variable is created or assigned storage
- A variable defined before the function definition in a source file is visible to the function, as seen in previous example
- Remember multiple source files example from Introduction?

extern keyword

- The `extern` keyword is used to declare variables defined outside the current function or source file

```
int a;

int main() {
  int a = 10; // "a" is local to main
  print(); // prints "a: 0"
}

int print() {
  extern int a;
  printf("a: %d\n", a);
}
```

auto keyword

- Variables within functions or code blocks that are not declared as extern are auto (for automatic)

```
int a;

int main() {
  auto int a = 10; // "a" is local to main
  print(); // prints "a: 0"
}

int print() {
  extern int a;
  printf("a: %d\n", a);
}
```

static variables

- A variable declared with the keyword `static` within a function or code block retains its value till the program ends
- A static variable anywhere else in the source file is considered local to that file

```
int main() {
  print(); // prints "a: 0"
  print(); // prints "a: 1"
}

int print() {
  static int a;
  printf("a: %d\n", a++);
}
```

register variables

- Useful for advising a compiler to retain a heavily used variable in a CPU register
- Examples

```
register int i;
register char c;
```

Variable initialization

- External and static variables
  - Are guaranteed to be initialized to zero
  - Any values assigned must be constant expressions
- Automatic and register variables
  - Contain garbage unless initialized
  - Can be initialized by specifying expressions containing constants and variables defined earlier

Recursion

- A function can call itself
- The local automatic variables are stored in the stack
- Function parameters are passed using the stack
- Prone to stack overflow
- There is always a danger of creating an infinite loop if the exit criteria is not clear

Recursion – Example

```
int main() {
  print(1);
}

int print(int i) {
  printf("i:%d\n", i++);
  if (i > 5) return;
  else print(i);
}
```

Header files

- Used to include external variable and function definitions
- Allow applications to be compiled in parts
- The remaining parts are resolved during linking from statically or dynamically linked libraries
- Remember the example from Introduction?

Macro definition and substitution

- A macro definition takes the form

  ```
  #define name replacement-text
  ```

- Token name has the same syntax as a variable name
- Everywhere in the source file, where the token name occurs, it is substituted by `replacement-text`
- `replacement-text` is any arbitrary text and it can span several lines by ending each line with a \
- A macro can also be defined or redefined by using the `-D` compiler option

  ```
  gcc -Dname=value
  ```

Un-define macros

- To un-define a macro called `name`

  ```
  #undef name
  ```

- A macro defined in a program can also be undefined by using the `-U` compiler option

  ```
  gcc -Uname
  ```

- where `name` is the name of the macro you want to undefine

Macro with arguments

- Look like functions but result in inline code
- Macro with arguments are applicable to arbitrary types

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))

MAX(1.5,2.9) // ((1.5) > (2.9) ? (1.5) : (2.9))

MAX(a+b, c+d) // ((a+b) > (c+d) ? (a+b) : (c+d))
```

- The parentheses are required to maintain proper expression semantics after substitution

Macro with arguments – additional syntax

```
#define debug_print(expression) printf(\
  #expression " = %g\n", expression)
debug_print(x); //printf("x" " = %g\n", x);

#define concat(prefix, suffix) prefix ## suffix
concat(name, 1); //name1;
```

Conditional inclusion

- Preprocessing provides for means to insert code conditionally
- This can useful to
  - Enable or disable tracing statements
  - Include OS specific code
  - Include a header file just once

Enable and disable tracing

- Only integer constants and the following operators can be used in the expression following `#if`
  - `&&`, `||`, `<`, `>`, `<=`, `>=`, `!`, and `==`

```
#define TRACE_NONE 0
#define TRACE_DEBUG 1
#define TRACE_ALL 2

#define TRACE_LEVEL TRACE_DEBUG

int main() {
#if TRACE_LEVEL == TRACE_ALL || \
    TRACE_LEVEL == TRACE_DEBUG
  printf("within main\n");
#endif
  return 0;
}
```

OS specific codes

```
#if !defined(OSNAME)
  #error OSNAME not specified
#endif

#if OSNAME == LINUX
  printf("Linux\n");
#else
  printf("Windows\n");
#endif
```

- Compile program

  `gcc -DOSNAME -DLINUX macro.`

Include header file just once

```
#ifndef _HDR_H_
#define _HDR_H_
  declarations
#endif
```

Exercise

- A factorial of a number `n`, denoted as `n!`, is calculated as:

  `n * (n-1) * (n-2)... 3 * 2 * 1`

  Thus, `5!=120` and `10!=3628800`

- Write a recursive function to calculate factorial for any number `n`