C Functions

- Functions break large computing tasks into smaller ones:
  - Facilitate code reuse: making function calls to perform specific tasks, instead of writing the code from scratch.
  - Facilitate code maintenance: updating code in one place, instead of everywhere.

C Functions - Continue

- Before a function can be called by other functions, such as main, it has to be defined.
- Alternatively, one can provide a function prototype and define the function later.
- Function can also be defined in multiple source files (*.c) and compiled separately to object files (*.o).
- Linker links all object files (*.o) or libraries (*.a) to generate one final executable.

Syntax

- Function Prototype:
  ```c
  return_type function_name(type1, type2 ..., typen);
  ```
- Function Definition:
  ```c
  return_type function_name(type1 name1, type2 name2, ...,typen namen)
  {
    declarations
    statements
  }
  ```
  ```c
  int max(int, int);
  ```
  ```c
  int main()
  {
    int x;
    x = max(5, 9);
    x = max(x, 7);
  }
  ```
  ```c
  int max(int a, int b)
  {
    return a > b?a:b;
  }
  ```
Some Examples

- Function Prototype:
  - Optional if the function returns type is `int`;

```c
double squared (double);
// inconsistency causes compile-time error
void print_report (int);
// no return value
get_menu_choice (void);
// assume return int value
```

- Function Definition:

```c
double squared (int number) {
    return (number * number);
}
```

// have to declare function prototype if defined after main, since the return type is not int.

```c
void print_report (int);
```

// no return value

```c
get_menu_choice (void);
// assume return int value
```

Passing Arguments

- Arguments are passed through function calls:

```c
func1 (a, b, c);
```

- Function header:

```c
int func1 (int x, int y, int z)
```

- Parameters `x y z` bind to the value of `a b c`

- Each argument can be any valid C expression that has a value:

```c
x = func1(x+1, func1(2,3,4), 5);
```

- Type conversions may occur if types do not match.

Parameters are Passed by Value

- All parameters are passed by value!!

  - This means function parameters are local variables initialized to the values that the function is called with.
  - `x=10; y=20`

  - They can be modified within the function, like other local variables, but these modifications are not visible outside the function.

```c
#include<stdio.h>
swap(int x, int y) {
    int tmp;
    tmp=x; x=y; y=tmp;
}
```

```c
int main() {
    int a=10, b=20;
    swap(a,b);
    printf("%d,%d\n",a,b);
}
```

Pass Parameter by Reference

- The unary operator `&` gives the address of an object:

  ```c
  int c = 10;
  int *p;
  // *p is an int.
  p = &c;
  // p points to c.
  ```

- The unary operator `*` dereference a pointer:

  ```c
  int a=10, b=20;
  swap(&a, &b);
  printf("%d,%d\n",a,b);
  ```

```c
#include<stdio.h>
swap(int *x, int *y) {
    
    int main() {
        int a=10, b=20;
        swap(&a, &b);
        printf("%d,%d\n",a,b);
    }
```
External vs. Internal Variables

- **External (global) variables** are defined outside of any function, and are thus available to many functions.
  - Avoid to reduce function dependency
- **Internal (local) variables** are the function arguments and the variables defined inside functions.
  - Have no meaning outside of the function body

Scope Rules

- The **scope** of a name (variable, constant, function) is the part of the program within which the name can be used.
- The scope of **local variables and constants** defined in a function is the function in which the name is declared. (local function?)
- The scope of an **external variable, constant or a function** lasts from the point at which it is declared to the end of the file being compiled.

External Variables

- Similar to functions, **external variables** must be defined before it can be used.
- Alternatively, you can declare an external variable **extern** and define the variable later.
  ```c
  extern int sp;
  //extern is a declaration, no memory allocation
  main() { sp ...}
  int sp;
  //definition, memory is allocated
  ```
- External variable can be defined in other files and compiled separately to object files, then linked together later.

Placing a small size C program

- Place the program in **one source file** using the following order:
  - All function prototypes
  - All external variables declaration
  - **main()** function
  - All external variable definition
  - All function definition
  ```c
  int f1(int,int);
  extern double v1;
  int main(){
    f1(...) ;
    v1 ...}
  double v1;
  int f1(int a, int b)
  { ... }
  ```
Placing a large size C program

- Place related functions in one .c file.
- Write a .h file containing all the function prototypes, external variables
- #include the header file in the files that uses the functions.
- Advantages:
  - Separate compilation of source files (.c)
  - Reduce the dependency of source files: recompile .c only when its #include .h file is changed (makefile)

The C Preprocessor

- The C preprocessor, often known as cpp, is a macro processor that is used automatically by the C compiler to transform your program before compilation.
- #include "filename" or #include <filename>
  - cpp replaces them with the contents of the file filename.
  - "filename": cpp searches for the file where the source program was found first; if the file is not there, cpp searches the directories used for system header files.
  - <filename>: cpp searches the directories used for system header files (/usr/include or /usr/local/include).

Macro Substitution

- #define name replacement-text
- cpp replaces the subsequent occurrences of name with replacement-text.
- #define max(A,B) ((A)>(B)?(A):(B))
- x=max(p+q, r+s);
  - will be replaced by the line:
  - x=((p+q)>(r+s)?(p+q):(r+s));
- x=max(i++,j++)//what’s wrong?
Macro Substitution - Continue

• Macros are used to avoid the run-time overhead of function calls (create and maintain stacks to store parameter/local variables associated with the function).
• When the size of the code is small and the code is used frequently, macro is a better implementation decision than function.