Week 3

XMUT-NWEN 241 - 2024 T2

Systems Programming

Mohammad Nekooei

School of Engineering and Computer Science

Victoria University of Wellington

Admin

- Exercise 1
  - Do not modify messages
  - Do not add any String to the output
  - Otherwise, the automatic marking might not mark your work correctly.

Content

- Functions
- Function-like macros

Functions

Functions

- Unlike Java, C allows functions to exist on their own, i.e., outside any class
  - In C, functions are first-class entities: a C program consists of one or more functions

- A C program must have exactly one `main` function
- Execution begins with the `main` function

- What are functions good for?
  - structuring our thoughts (structured programming)
  - allowing us to re-use code, reducing work and reducing errors

Creating a simple function

- Suppose we frequently wanted to compare two integers and then use the larger.
- We might have code like this repeatedly written in our program:

```
int p, q, l;
...                /* p, q initialised */
if (p > q){
  l = p;
}
Else{
  l = q;
}
...                /* l gets used */
```

Creating a simple function

- How about making it a stand-alone function?

```
l = larger(p, q);
```

- What we need to do:
  - Pick a name for the function: `larger()`
  - Specify what type of variables that `larger()` is going to compare: `larger(int, int)`
  - Specify what type of value that `larger (int, int)` is going to return: `int larger(int, int)`
  - This is called function prototype / declaration `int larger(int, int);`

- Code it: function definition/implementation

Function Prototype

- A declaration specifying the return type, function name, and list of parameter types

```
return_type function_name ( parameter_types_list );
```

- Function prototype is used by compiler to check whether your program is passing the right parameters

- Hence, a function prototype must be declared prior to its use
  - For functions in standard library (and other provided libraries), include the appropriate header files
  - For user-defined functions, you have to declare the prototypes before the main function

Function Prototype

- Examples

```
void say_hello ( void );
```

```
int add ( int a, int b );
```

- No need to provide identifiers to input parameters, the types of the input parameters are sufficient

```
int add ( int, int );
```

```
int larger ( int, int );
```

Functions

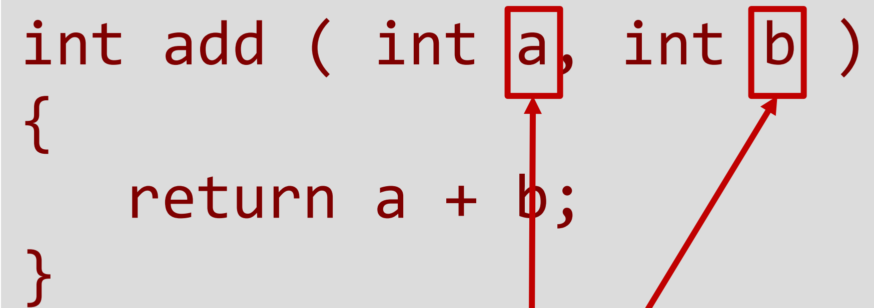
- General form of a C **function definition:**

```
return_type function_name ( parameter_list )
{
    body of the function
}
```

Functions

- Examples

```c
void say_hello ( void )
{
   printf("Hello");
}
```



Formal parameters

Creating a simple function

- Function definition:

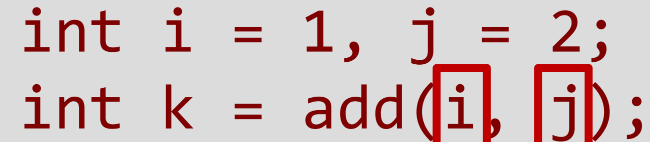
```
int larger(int x, int y)
{
    if (x > y)
    {
        return x;
    }
    else
    {
        return y;
    }
}
```

- x and y are called “formal parameters”, whose scope is the body of the function

Calling Functions

- Example function calls:

```c
say_hello();
```



- Before a function can be called, either the **function definition** or **function prototype** should have been declared prior to the call

Calling a function: example

```
...
int larger(int, int);

int main(void)
{
  ...
  l = larger(p, q);  /* p and q are called “actual */
  ...                /* parameters”. Their values are */
                     /* going to be copied to x and y. */
}
int larger(int x, int y)
{                     /* x and y (NOT p and q) are */
  if (x > y)          /* going to be compared here. */
    return x;         /* the larger value is going to be */
  else return y;     /* returned to larger(p, q). */
}
```

Function-like macros

Macro Substitution

- Recall: C preprocess can used to define constants

```
#define name replacement
```

- Subsequence occurrences of `name` will be replaced by `replacement`

Function-like Macro

- Can *abuse* macro substitution to define **function-like** macros
- To define a function-like macro, just append ( ) to the macro name
- Example:

```
#define READ_CHAR()       getchar()
```

- Can be called like a regular function:

```
…
int c = READ_CHAR();
…
```

Function-like Macro

- Just like functions, function-like macros can take arguments
  - Insert comma-separated parameter names between (and)
  - Parameter names must be valid identifiers

```
#define max(X, Y) ((X) > (Y) ? (X) : (Y))
```

- call just like normal functions

```
z = max(1, 3);
```

```
z = ((1)>(3)?(1):(3));
```

This expression evaluates to **3**

Problems with Function-like Macros

- Suppose:

```
#define SQ(X)       X * X
```

- Then:

```
(int)SQ(r);
```

```
(int)r * r;
```

```
SQ(r1 + r2);
```

```
r1 + r2 * r1 + r2;
```

- Solution: enclose individual variables with (), including the whole replacement text

```
#define SQ(X)       ((X) * (X))
```

Problems with Function-like Macros

- Suppose:

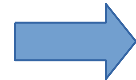
```
#define SQ(X)        ((X) * (X))
```

- Then:

```
(int)SQ(r);
```

```
(int)((r) * (r));
```

```
SQ(r1 + r2);
```



```
((r1 + r2) * (r1 + r2));
```

Problems with Function-like Macros

- Macro

```
#define SQ(x) x * x
```

```
#define SQ(x) (x) * (x)
```

```
#define SQ(x) ((x) * (x))
```

- Example Problematic Usage

```
SQ(1+1)
```

```
(int)SQ(2.0) % 2
```

```
SQ(i++)
```

```
SQ(f())
```

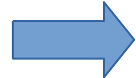
Problems with Function-like Macros

- Suppose:

```
#define SQ(X)      ((X) * (X))
```

- How about these:

```
SQ(++r);
```



```
((++r) * (++r));
```

`r` incremented twice

```
SQ(f());
```

```
((f()) * (f()));
```

`f()` called twice

Be careful when defining and calling function-like macros!

Next Lecture

- Arrays