

# Functions

# Flow of Control Review

- while
- for
- do while
- goto
- break & continue
- switch
- Relational operators
- short circuits

# Functions

- To avoid repetitive similar code
- To structure the whole program as “top-down” approach
  - Breaking up a large problem into smaller pieces and break each piece into smaller ones until each piece is readily expressed in code
  - Each piece should be concise and logical entity

```
long power(int m, int n)
```

header

```
{
```

```
    int    i;
```

```
    long   product = 1;
```

body

```
    for (i = 1; i <= n; ++i)
```

```
        product *= m;
```

```
    return product;
```

```
}
```

# Function Definition

- function type
  - Type of the return value, if any
  - If missing, it is assumed to be int
- parameters are placeholders for values that are passed to the function when it is invoked
- return statements
  - return;
  - return a+b;
  - return (a+b);

# Function Prototypes

- Each function should be declared before it is used
  - what if your program structure is top down?
  - some people prefer bottom-up approaches
- Usually, they are placed before the main( ) function
- Parameter names can be omitted (ANSI C)
  - parameters can be omitted (old C)

```
void f(char c, int i);  
void f(char, int);
```

```
#include <stdio.h>

#define    N    7

long    power(int, int);
void    prn_heading(void);
void    prn_tbl_of_powers(int);

int main(void)
{
    prn_heading();
    prn_tbl_of_powers(N);
    return 0;
}

void prn_heading(void)
{
    printf("\n:::::  A TABLE OF POWERS  :::::\n\n");
}
```

tbl of power.c

```

void prn_tbl_of_powers(int n)
{
    int    i, j;

    for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j)
            if (j == 1)
                printf("%ld", power(i, j));
            else
                printf("%9ld", power(i, j));
        putchar('\n');
    }
}

```

```

long power(int m, int n)
{
    int    i;
    long   product = 1;

    for (i = 1; i <= n; ++i)
        product *= m;
    return product;
}

```



# Function Invocation

- The program starts by invoking the main function
- parameters are passed as call-by-value
  - you can implement call-by-reference with pointers

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int    n = 3, sum, compute_sum(int);
```

```
    printf("%d\n", n);          /* 3 is printed */
```

```
    sum = compute_sum(n);
```

```
    printf("%d\n", n);          /* 3 is printed */
```

```
    printf("%d\n", sum);        /* 6 is printed */
```

```
    return 0;
```

```
}
```

```
int compute_sum(int n)          /* sum the integers from 1 to n */
```

```
{
```

```
    int    sum = 0;
```

```
    for ( ; n > 0; --n)          /* stored value of n is changed */
```

```
        sum += n;
```

```
    return sum;
```

```
}
```

compute\_sum.c

# Developing a Large Program

- Usually developed by several teams
- Comprises many .h and .c files
  - each .c file can be compiled separately
- gcc -o pgm main.c fct.c wrt.c

```
pgm.h pgm.zip  
#include ...  
#define ...  
...  
templates of enum., structure, union types  
list of function prototypes
```

```
main.c      fct.c      wrt.c  
#include "pgm.h"  #include "pgm.h"  #include "pgm.h"  
...         ...         ...
```

# Assertion

- you can make sure a certain condition holds true at any place of program
  - it is a macro defined in the header file `assert.h`
  - `assert(expression);`
  - if the value of the expression is zero abort the program

# Scope Rules

- identifiers are accessible only within the block where they are defined
  - they are invisible from outside

```
{
    int a = 2;
    printf("%d\n", a);
    {
        int a = 5;
        printf("%d\n", a);
    }
    printf("%d\n", ++a);
}
```

# Storage Classes

- Every variable and functions in C has two attributes: type and storage class
- Storage Classes  
auto          extern          register          static

# auto

- the most common class
  - variables defined inside a function
  - variables defined outside a function are **global**
- default class – you may omit it
- the memory space is allocated/released when the function is invoked/exited
- when a function is reentered, the previous values are unknown

# external

- global
- they may be defined somewhere else (in another file)
- they never disappear
  - transmit values across functions
- they may be hidden by re-declaration, but they are not destroyed



```
#include <stdio.h>

int    a = 1, b = 2, c = 3;
int    f(void);

int main(void)
{
    printf("%3d\n", f());
    printf("%3d%3d%3d\n", a, b, c);
    return 0;
}

int f(void)
{
    int    b, c;

    a = b = c = 4;
    return (a + b + c);
}
```

In file file1.c

```
#include <stdio.h>
```

```
int    a = 1, b = 2, c = 3;    /* external variables */  
int    f(void);
```

```
int main(void)  
{  
    printf("%3d\n", f());  
    printf("%3d%3d%3d\n", a, b, c);  
    return 0;  
}
```

In file file2.c

```
int f(void)  
{  
    extern int    a;            /* look for it elsewhere */  
    int           b, c;  
  
    a = b = c = 4;  
    return (a + b + c);  
}
```

# register

- allocate this variable on a register
- to speed up the execution
- not always possible to find a register
- tricky for memory-IO operations

# static

- to preserve the value even after the function exits
- visible only within the same source file
  - static external

```
void f(void)
{
    static int cnt = 0;
    ...
}
```