FUNCTIONS

Functions

- The heart of effective problem solving is **problem decomposition**.
  - breaking a problem into small, manageable pieces
  - In C, the function construct
- A program consists of one or more files
  - Each file contains zero or more functions, one of them being a **main()** function.

Function Definition

type function_name (*parameter list*) : function header
{
    declarations
    statements : function body
}

- A function definition starts with the *type* of the function.
  - If no value is returned, then the *type* is **void**.
  - If NOT **void**, then the value returned by the function will be converted, if necessary, to this *type*.
- *parameter list*
  - a comma-separated list of declarations
  - formal parameters of the function

Function Definition

```
int factorial(int n)          /* header */
{                              /* body starts here */
   int i, product = 1;
   for (i = 2; i <= n; ++i)
      product *= i;
   return product;
}

void main(void)
{
     …
     factorial(7);       /* The function, factorial(), is called, or invoked */
     …
 }
```

Function Definition

```
void nothing(void) { }    /* this function does nothing */

double twice(double x)
{
   return 2.0 * x;
}
```

```
int all_add(int a, int b)
{
    int c;
     …
    return (a+b+c);
 }
```

⇔

```
all_add(int a, int b)
{
     …..
```

Function Definition

- "local" variables vs. "global" variables (internal vs external)
  - "local" variables : any variables declared in the body of a function
  - "global" variables : other variables declared **external** to the function

```
#include <stdio.h>

int   a = 33;        /* a is external*/

int main(void) {
   int   b = 77;                  /* b is local to main() */

   printf("a = %d\n", a);         /* a is global to main() */
   printf("b = %d\n", b);
   return 0;
}
```

Function Definition

- Important reasons to write programs as collections of many small functions
  - It is simpler to correctly write a small function to do one job.
    - easier writing & debugging
  - It is easier to maintain or modify such a program
  - Small functions tend to be self-documenting and highly readable.

return Statement

return;

return *expression*;

- When a **return** statement is encountered,
  - execution of the function is terminated and
  - control is passed back to the calling environment

  < Examples >

  ```
  return;
  return ++a;
  return (a*b);
  ```

return Statement

```
float f(char a, char b, char c)
{
   int i;
   …
   return i;  /*value returned will be converted to a float*/
}

double absolute_value(double x)
{
   if (x>=0.0)
      return x;
   else
      return –x;
}
```

```
while (….) {
   getchar(); /*get a char, but do nothing with it */
   c = getchar();
}
```

Function Prototypes

- Functions should be declared before they are used.
- *Function prototype*
  - tells the compiler the # and type of argument passed to the function
  - tells the type of the value returned by the function
  - allows the compiler to check the code more thoroughly

type function_name (*parameter type list*);

double sqrt(double);

  - Identifiers are optional.

void f(char c, int i); ⇔ **void f(char, int);**

Styles for Function Definition Order

- A way to write a program in a single file
  1. **#includes** and **#defines** at the top of file
  2. enumeration types, structures, and unions
  3. a list of function prototypes
  4. function definitions, starting with main()

Styles for Function Definition Order

```
#include <stdio.h>
#define   N       7
void prn_header(void);
long power(int, int);
void prn_tbl_powers(int);
int main(void)
{
    prn_header();
    prn_tbl_of_powers(N);
    return 0;
}
void prn_header(void)
{
    ….
}
long power(int m, int n)
{
    …
}
void prn_tbl_powers(int n)
{
    …
    printf("%ld", power(i, j);
    …
}
```

```
#include <stdio.h>
#define   N       7
void prn_header(void)
{
    ….
}
long power(int m, int n)
{
    …
}
void prn_tbl_powers(int n)
{
    …
    printf("%ld", power(i, j);
    …
}
int main(void)
{
    prn_header();
    prn_tbl_of_powers(N);
    return 0;
}
```

Function Invocation and Call-by-Value

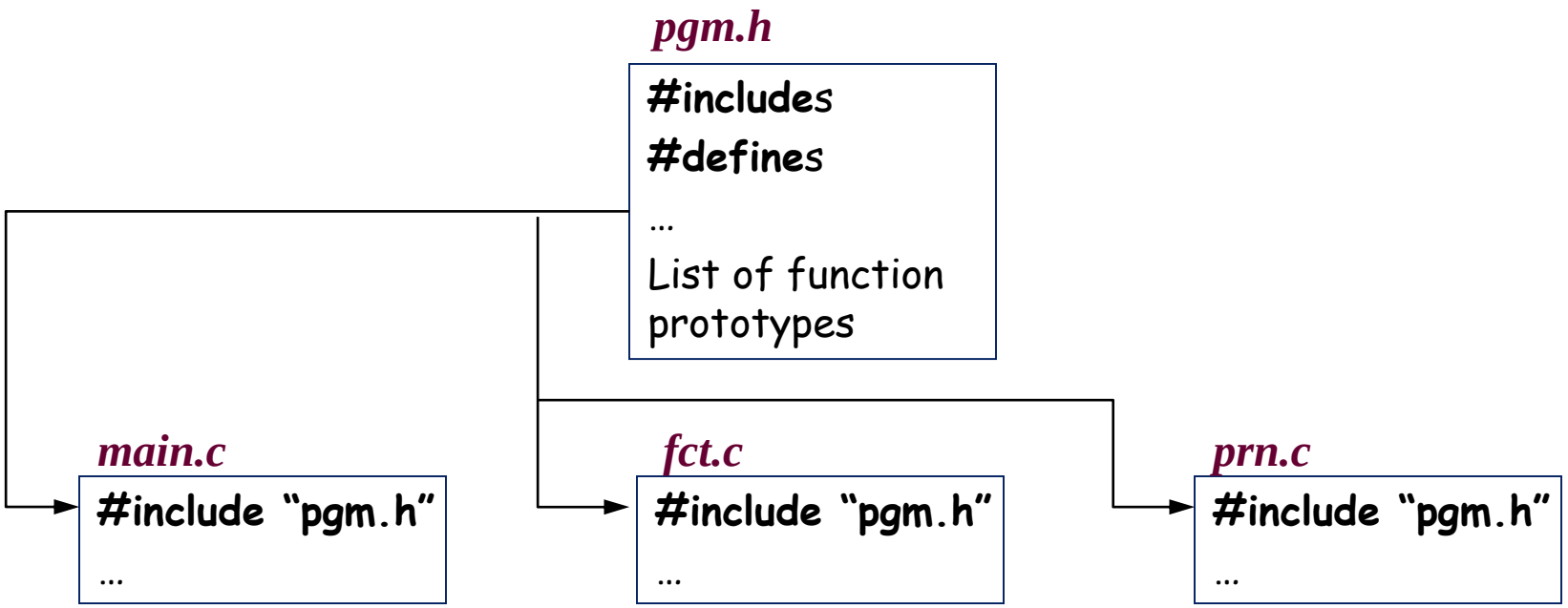
- When program control encounters a function name,
  - the function is called, or, invoked
    - The program control passes to that function
  - After the function does its work, the program control is passed back to the calling environment.
- Functions are invoked
  - by writing their name and a list of arguments within ().
- All arguments for a function are passed "call-by-value"
  - Each argument is evaluated, and its value is used locally.
  - The stored value of that variable in the calling environment will NOT be changed.

Function Invocation and Call-by-Value

```
#include <stdio.h>
int compute_sum(int n);
int main(void)
{
   int n = 3, sum;
   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)
{
   int sum = 0;
   for ( ; n>0; --n)         /*n is changed*/
      sum += n;
   return sum;
}
```

Developing a Large Program

- A large program is typically written in a collection of .h and .c files.



Because *pgm.h* occurs at the top of each .c file, it acts as the "glue" that binds our program together.

Using Assertions

- a macro, **assert(***expr***)**
  - in the standard header file assert.h
  - If *expr* is false, the system will print a message, and the program will be aborted.

```
#include <assert.h>
#include <stdio.h>
int   f(int a, int b);
int   g(int c);
int   main(void)
{
    int  a,b,c;
    …
    scanf("%d%d", &a, &b);
    …
    c = f(a,b);
    assert(c > 0);  /*an assertion */
    ….
}
```

```
program: program.c:5: main: Assertion 'c > 0' failed.
Abort (core dumped)
```

Scope Rules

- Basic rules of scoping
  - Identifiers are accessible only within the block in which they are declared.
  - They are unknown outside the boundaries of that block.

Scope Rules

```
{
   int a = 2;              /*outer block a*/
   printf("%d\n", a);      /*2 is printed*/
   {
      int a = 5;           /*outer block a*/
      printf("%d\n", a); /*5 is printed*/
   }                       /*back to the outer block*/
   printf("%d\n", ++a); /*3 is printed*/
}
```

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

- ✓ An outer block name is valid unless an inner block redefines it.
- ✓ If redefined, the outer block name is hidden, or masked, from the inner block

Scope Rules

```
{
   int a = 1, b = 2, c = 3;
   printf("%3d%3d%3d\n", a, b, c);                  /*  1  2  3 */
   {
      int       b = 4;
      float     c = 5.0;
      printf("%3d%3d%5.lf\n", a, b, c);             /*  1  4  5.0 */
      a = b;
      {
          int   c;
          c = b;
          printf("%3d%3d%3d\n", a, b, c);           /*  4  4  4 */
      }
       printf("%3d%3d%5.lf\n", a, b, c);            /*  4  4  5.0 */
   }
   printf("%3d%3d%3d\n", a, b, c);                  /*  4  2  3 */
}
```

Scope Rules

- Parallel and Nested Blocks
  - Two blocks can come one after another
    - The 2nd block has no knowledge of the var.s declared in the 1st block.
    - Parallel blocks
- Why blocks?
  - to allow memory for variables to be allocated where needed
  - Block exit releases the allocated storage

```
{
   int a, b;
   …
   {                  /* inner block 1 */
      float b;
      …               /* int a is known, but not int b */
   }
   …
   {                  /* inner block 2 */
      float a;
      …               /*int b is known, but not int a*/
                      /* nothing in inner block 1 is known*/
   }
}
```

Storage Classes

- Every variable and function in C has two attributes
  - *type* and *storage class*
- Four storage classes

  auto **extern** **register** **static**
  - **auto** : automatic
    - The most common storage class for variable

Storage Class **auto**

- Variables declared within function bodies are automatic by default
  - When a block is entered, the system allocates memory for the automatic variables.
    - These variables are "local" to the block
  - When the block is exited, the memory is automatically released (the value is lost)

```
void f(int m)
{
   int a,b,c;
   float f;
   ….
}
```

Storage Class **extern**

- One method of transmitting information across blocks and functions is To Use External variables
- When a var. is declared outside a function,
  - storage is permanently assigned to it.
  - its storage class is **extern**
  - The var. is "global" to all functions declared **after** it.

- Information can be passed into a function two ways
  ① by use of external variables
  ② by use of the parameter mechanism

Storage Class extern

```
#include <stdio.h>
int   a = 1, b = 2, c = 3;                  /* global variables */
                                            /* definition and declaration */
int   f(void);
int main(void)
{
   printf("%3d\n", f());                    /* 12 is printed */
   printf("%3d%3d%3d\n", a, b, c);          /*  4  2  3 is printed */
   return 0;
}
int f(void)
{
   int b, c;                                /* b and c are local */
   a = b = c= 4;                            /* global b, c are masked */
   return (a+b+c);
}
```

Storage Class **extern**

In file file1.c

```
#include <stdio.h>
int   a = 1, b = 2, c = 3;                    /* external variables */
intf(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);
}
```

- ✓ The keyword **extern** is used to tell compiler to "look for elsewhere, either in this file or in some other file."

Definition & Declaration

- External Variables

- Definition: the variable is created or assigned storage
- Declaration: the nature of the variable is stated but no storage is allocated

- The scope of an external variable or a **function** lasts from the point at which it is declared to the end of the file being compiled.

Storage Class **register**

- The storage class **register**
  - tells the compiler that the associated var.s should be stored in high-speed memory registers
  - aims to improve execution speed
    - declares var.s most frequently accessed as **register**

```
{
   register int  i;                    /* ⇔ register i */
   for (i = 0; i < LIMIT; i++) {
      …
   }
}     /* block exit will free the register */
```

Storage Class **static**

- The storage class **static**
  - allows a local var. to retain its previous value when the block is reentered.
  - in contrast to ordinary **auto** variables

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

   ++cnt;
   if(cnt % 2 == 0)
      …                    /* do something */
   else
      …                    /* do something different */
}
```

- ✓ The first time the function **f()** is invoked, **cnt** is initialized to zero.
- ✓ On function exit, **cnt** is preserved in memory
- ✓ Whenever **f()** is invoked again, **cnt** is not reinitialized

Default Initialization

- **external** and **static** variables
  - initialized to zero by the system, if not explicitly initialized by programmers

- **auto** and **register** variables
  - usually not initialized by the system
  - have "garbage" values.

Recursion

- A function is recursive if it calls itself, either directly or indirectly

```
#include <stdio.h>
int sum(int n);
int main(void)
{
   int n = 5;
   printf("The sum from 1 to %d is %d\n", n, sum(n));
   return 0;
}
int sum(int n)
{
   if (n <= 1)
      return n;
   else
      return (n + sum(n-1));
}
```

Function call	Value returned		
sum(1)	1		
sum(2)	2 + sum(1)	or	2 + 1
sum(3)	3 + sum(2)	or	3 + 2 + 1
sum(4)	4 + sum(3)	or	4 + 3 + 2 + 1

Recursion

```
#include <stdio.h>
int factorial(int n);
int main(void)
{
   int n = 5;
   printf("The factorial from 1 to %d is %d\n", n, sum(n));
   return 0;
}
int factorial (int n) /*recursive version */
{
   if (n <= 1)
      return 1;
   else
      return (n * factorial(n-1));
}
```

```
int factorial (int n) /* iterative version*/
{
   int product = 1;
   for (; n > 1 ; --n)
      product *= n;

   return product;
}
```