Functions

(aka methods, subroutines, subprograms, procedures)

- When used or called, a function can be treated as a black box
  - User cares only about what the function does, not about how (performance and errors aside)
  - The function’s interface or prototype is a description of how to connect with or call the function
- Pass-by-value input (parameters) may be
  - Constants
  - Variables
  - Expressions

```
5 4
3
```

Function Definition

Elements of a function

- Return value type
- Function name
  - Follows rules for all identifiers
  - 31 characters significant minimum
  - First character is a letter or
  - Subsequent characters are letters, digits, or __
- Argument list
  - Empty or void
  - List of types and names
- Body
  - Local variables
  - Statements

```
header — type name(arguments)
{ local variables;
  statements;
}
```

Function Return Values

Getting a value from a function

- The default return type is int
  - Functions returning an integer should be explicitly typed
  - The void return type specifies a function that does not return a value (i.e., a "procedure")
- Functions fall into one of several categories
  - Function computes and returns a value from its arguments
  - Changes its arguments or global data, or performs some action
    - Does not return a value
      - Returns a status as the functional value
  - Functions terminate by “falling off the end” or calling return
    - return may be called anywhere in the function
    - Function return values are specified with the return statement

Function Examples

User defined functions

```
double calcPayment(double P, double R, int N)
{
    return P * R / (1 - pow(1 + R, -N));
}
```

```
double random()
{
    static double x = 0;
    x = x * (x + 1) % 2147483648L;
    return x;
}
```
Function Prototypes

Function declarations
- Function prototypes have three components
  - Name
  - Return value type
  - Argument list
- Information is placed in the compiler’s symbol table
  - No body implies that no code is generated
  - Permits the compiler to
    - verify usage: number of arguments and void or return value
    - type-check arguments
    - perform appropriate type conversions on arguments and return value
- Introduced in C++ and back-ported to ANSI C
  - C: if a function is called without a prototype, the compiler guesses
  - C++: if a function is called without a prototype, it will not compile

Function Calls

Invoking a function
- Functions are called by name followed by a parameter list
  - Explicit type information is not given in the function call
  - Empty parameter lists still require the parentheses
  - The return value may be ignored
- Note that function calls do not contain typing information
- Examples
  - double payment = calcPayment(P, R, N);
  - double payment = calcPayment(100000, 0.08/12, 360);
  - double pseudo_random = random();
  - calcPayment(100000, 0.08/12, 360);

Function Example

```c
main first

double sqr(double x); /* prototype: declares function sqr */
void main(void) {
 double y;
 y = sqr(2); /* assumes that sqr returns an int */
}
double sqr(double x) /* defines function sqr */
{ return x * x;
}
```

Function Example

```c
Functions first

double sqr(double x) /* defines & declares function sqr */
{ return x * x;
}

int main( )
{ double y;
  y = sqr(2); /* 2 promoted to double before call */
}
```

Global Variables

Variables defined outside of functions
- File scope (program scope with external declarations)
- Values maintained throughout program execution
- Programmer may initialize; compiler initializes to 0
- Local variables supersede (i.e., hide) global variables
- Used to:
  - Reduce the number of parameters
  - "Return" multiple values
  - Allow widely "separated" functions to communicate
  - Return status or error values
- Problems with global data
  - Shared data couples functions—must test & debug as a unit
  - Contributes to name space pollution

Local / Local Variable Example

Local variables hide global variables with the same name
- Not a multiple definition error
  - Each counter variable is in a different scope
  - Local scope supersedes global scope (i.e., local variable hides global variable)

```c
int nlines = 10;
int counter = 100; // global variable

void function( )
{ int counter = 200; // local variable
  printf("nlines \%d counter \%d\n", nlines, counter);
}
```
## extern Variables

Variable declarations: type information for the compiler

- External variable statements declare variables defined at global scope in another file (i.e., they expand the variables scope to the entire program)
- May not be initialized
- A variable may have multiple declarations as long as they are all the same
- Often placed in header files
- `extern` type name

```c
int counter = 100;
void increment(void)
{
    counter++;
}
```

### file1.c

```c
extern int counter;
void report(void)
{
    return counter;
}
```

---

## Variable Scope Example

Function / block scoping rules

```c
void func1(int);
void func2(int);
int increment(void);
int number = 10;
int count = 0;
int main()
{
    int count;
    int i;
    while (count != 0)
    {
        int k;
    }
}
```

```c
extern int number;
void func1(int i)
{
    int j;
}
```

```c
void func2(int i)
{
    int j;
}
```

```c
int increment()
{
    static int count = 0;
    return count++;
}
```

### file2.c

---

## Pass By Value

Bitwise copy from the call to the formal parameter

```c
void func(int y);
int main()
{
    int x = 10;
    func(x);
}
void func(int y)
{
    y = 20;
}
```

---

## Pass By Pointer

Three required changes (passing an address by value)

- Pass by address is used
- When a function must change its argument
- To increase efficiency when passing large data types

```c
void main()
{
    int x = 10;
    func(&x);
}
```

```c
void func(int* y)
{
    *y = 20;
}
```

---

## Swapping Two Variables

The classic pass by address example

```c
void swap(int* v1, int* v2)
{
    int temp = *v1;
    *v1 = *v2;
    *v2 = temp;
}
```

```c
int main()
{
    int a = 10, b = 20;
    swap(&a, &b);
}
```

---

## Pointers To Pointers

Multiple indirection

- Useful for allowing a function to change an argument that is already a pointer

```c
int i1 = 0, i2 = 20;
void func(int **pp)
{
    printf("%d\n", **pp);
    *pp = &i2;
    printf("%d\n", **pp);
}
```

```c
int main()
{
    int *p = &i1;
    printf("%d\n", *p);
    func(&p);
    printf("%d\n", *p);
}
```
Recursive Functions

Essential elements of recursion

- **Direct recursion**: a function calls itself
- **Indirect recursion**: A calls B, B calls C, ..., Y calls Z, Z calls A
- Must have 3 features:
  - Branch (usually in an if) that makes the recursive call
  - Branch (usually in an if) that does not recurse (i.e., terminates the recursion) – condition may be implicit rather than explicit
  - Input must change with each call
- **Theoretically**, recursion may be written as a loop
  - There is an existence proof of this – but it’s not a constructive proof

Recursion Example 1

(Print the digits of an integer one at a time)

```c
void forward(int number) {
    if (number != 0) {
        forward(number / 10);
        printf("%d\n", number % 10);
    }
}
```

Recursion Example 2

(Print the digits of an integer in reverse order)

```c
void reverse(int number) {
    if (number != 0) {
        printf("%d\n", number % 10);
        reverse(number / 10);
    }
}
```

Graphical Representation

Activation records and statement sequencing

<table>
<thead>
<tr>
<th>4th call</th>
<th>number == 0</th>
<th>number == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3rd call</td>
<td>number == 1</td>
<td>number == 1</td>
</tr>
<tr>
<td>2nd call</td>
<td>number == 12</td>
<td>number == 12</td>
</tr>
<tr>
<td>1st call</td>
<td>number == 123</td>
<td>number == 123</td>
</tr>
<tr>
<td></td>
<td>forward(12)</td>
<td>forward(12)</td>
</tr>
<tr>
<td></td>
<td>print(3)</td>
<td>print(3)</td>
</tr>
<tr>
<td></td>
<td>reverse(12)</td>
<td>reverse(12)</td>
</tr>
<tr>
<td></td>
<td>prints digits forward</td>
<td>prints digits reversed</td>
</tr>
</tbody>
</table>

Variable Argument Lists

Functions with an unknown number of arguments

- Functions must have at least one predetermined argument
  - Variable arguments follow the fixed argument(s)
  - Variable arguments are indicated with an ellipsis (…) in definitions and in prototypes
  - Type checking is suspended
  - int printf(char* control, …)
  - float avg(int n, …)

- Based on macros defined in <stdarg.h>
  - va_list variable length argument list type
  - void va_start(va_list ap, lastfix);
  - type va_arg(va_list ap, type);
  - void va_end(va_list ap);

Pointers To Functions

Dynamic function manipulation

- The address of a function is its entry point
  - int func1(char* str) { ... }
  - int func2(char* str) { ... }
- The name of the function, without any adornment, is its address
  - int (*tp) (char* s) = func1;
  - int (*ta[10]) (char* s) = { func1, func2 };
  - fa[0] = func1;
  - fa[1] = func2;
- "Configurable" algorithms
  - fp ("Hello World");
  - fa[1] ("Hello World");
The ANSI qsort Library Function

An implementation of the quick sort algorithm

- `void qsort(void* base, size_t num, size_t size, int (*fcmp)(const void*, const void*))`
  - `base` address of first element in an array
  - `num` number of elements in the array
  - `size` the size in bytes of each array element
  - `fcmp` a pointer to a function, which compares two array elements. Takes two void pointers for parameters and returns an integer: <0, ==0, or >0, if the first element is ordered before, the same, or after the second element