Chapter 6: Functions

Outline
Introduction
Function Definitions
Function Prototypes
Function Name by Identifier __func__
Recursive Functions
- Indirect Recursive Functions
Header Files
Type Generic Functions
Function Files
Sample Problem
Variable Number Arguments in Functions

Note: Some slides have been modified from the original.
Introduction

- A C program is generally formed by a set of functions. These functions subsequently consist of many programming statements.
- By using functions, a large task can be broken down into smaller ones.
- Functions are important because of their reusability. That is, users can develop an application program based upon what others have done. They do not have to start from scratch.
Function Definitions

- A function can be defined in the form of

```
return_type function_name(argument declaration)
{
    statements
}
```

Example:

```
int addition(int a, int b) {
    int s;
    s = a + b;
    return s;
}
int main() {
    int sum;
    sum = addition(3, 4);
    printf("sum = %d\n", sum);
    return 0;
}
```
• The return type can be any valid type specifier.
• The `return` statement can be used to return a value from the called function to the calling function as in return expression; If necessary, the expression will be converted to the return type of the function. However, if the expression cannot be converted to the return type of the function according to the built-in data type conversion rules implicitly, it is a syntax error.

**Example:**

```c
int func(void)
{
    double d;
    d = 4.6;
    return d; // OK: C type conversion, return 4
}
```
• If the return type is not \textbf{void}, a return statement is necessary at the end of the function in C99. Otherwise, the default zero will be used as the return value and a warning message will be produced by the system in C90 and Ch.

• A calling function can freely ignore the return value.

\textbf{Example:}

\begin{verbatim}
int func(int i){
    return i+1; // the same as 'return (i+1);'
}

int main() {
    int j;
    j = func(4);
    func(5); // ignore the return value
    return 0;
}
\end{verbatim}
If the return type is `void`, the return statement is optional. However, no expression should follow `return`; otherwise, it is a syntax error.

Example:

```c
void func(int i)
{
    if(i == 3)
    {
        printf("i is equal to 3 \n");
        return i;      // ERROR: return int
    }
    else if(i > 3)
    {
        printf("i is not equal to 3 \n");
        return;        // OK
    }
    i = -1;
    // return not necessary since return type is void
}

int main()
{
    func(2);
    return 0;
}
```
• The data type of an actual argument of the calling function can be different from that of the formal argument of the called function as long as they are compatible. The value of an actual argument will be converted to the data type of its formal definition according to the built-in data conversion rules implicitly at the function interface stage.

**Example:**

```c
int func1(int i) { // argument type is int
    return 2*i;
}

int main(){
    func1(5);         // OK
    func1(5.0);       // OK, 5.0 converted to 5
    return 0;
}
```
Sample Problem:

The system in Figure 1 (a) consists of a single body with mass \( m \) moving on a horizontal surface. An external force \( p \) acts on the body. The coefficient of kinetic friction between body and horizontal surface is \( \mu \). The freebody diagram for the system is shown in Figure 1 (b).

Figure 1: The system diagram and FBD of a sample problem
The nomenclature related to the modeling of the system is listed below.

- $m$ -- mass of the body
- $x$ -- position of the body
- $v$ -- velocity of the body
- $a$ -- acceleration of the body
- $g$ -- gravitational acceleration
- $\mu$ -- friction coefficient
- $f$ -- friction force
- $N$ -- normal force

**Equation of motion:**

The equation of the motion of the system can be derived based on the Newton's second law.

\[
N = mg \quad (1) \\
f = \mu N \quad (2) \\
p-f = ma \quad (3)
\]

From equation (1), (2) and (3), the formula for calculating the acceleration of the rigid body can be derived as follows.

\[
a = (p- \mu mg)/m \quad (4)
\]
Problem Statement:

For the system shown in Figure 1(a), given \( m = 5 \text{ kg}, \ g = 9.81 \text{ m/s}^2, \ \mu = 0.2 \). The external force \( p \) is expressed as a function of time \( t \),

\[ p(t) = 4(t-3)+20 \quad \text{when } t \geq 0 \]

write a program to calculate the acceleration \( a \) when \( t = 2 \text{ seconds} \).

Solve the above problem using two different programs with the features specified below.

1) Use a function to calculate the external force.
2) Use a function with multiple arguments to calculate the acceleration.

\[ a = \frac{(p - \mu mg)}{m} \]
Program 1:

Use a function to calculate the external force.

Output:

Acceleration a = 1.238000 (m/s^2)
Program 2:

Use functions with multiple arguments.

```c
/* File: accelfunm.c */
#include <stdio.h>
#define M_G   9.81

double force(double t) {
    double p;
    p = 4*(t-3)+20;
    return p;
}

double accel(double t, double mu, double m) {
    double a, p;
    p = force(t);
    a = (p-mu*m*M_G)/m;
    return a;
}

int main() {
    double a, mu, m, t;

    mu = 0.2;
    m = 5;
    t = 2;
    a = accel(t, mu, m);
    printf("Acceleration a = %f\n (m/s^2)", a);
    return 0;
}
```
Function Prototypes

• A function prototype is a declaration of a function that declares the return type and types of its parameters. For example,

\[
\begin{align*}
\text{double force(double t);} \\
\text{double accel(double t, double \mu, double m);} \\
\end{align*}
\]

• A function prototype contains no function definition.
• A function prototype can be declared at any lexical level, before and after its function definition.
• If the function is called before it is defined, \texttt{int} is assumed to be the return type of the function. Therefore, a function prototype is required if the return type of the function is not \texttt{int} and the function is called before its definition is processed, or if the function definition is located in a different file or in a library.
Problem Statement:

For the system shown in Figure 1(a), given \( m = 5 \text{ kg} \), \( g = 9.81 \text{ m/s}^2 \), \( \mu = 0.2 \). The external force \( p \) is expressed as a function of time \( t \),

\[
p(t) = 4(t-3)+20 \quad \text{when } t \geq 0
\]

write a program to calculate the acceleration \( a \) when \( t = 2 \) seconds.

Solve the above problem using the features specified below.
1) Use a function with multiple arguments to calculate the acceleration. The program lists function main() before other functions.
C for Engineers and Scientists: An Interpretive Approach

Program 3:

Use function prototypes.

```c
/* File: accelprot.c */
#include <stdio.h>
#define M_G   9.81

double force(double t);
double accel(double t, double mu, double m);

int main() {
    double a, mu, m, t;

    mu = 0.2;
    m = 5.0;
    t = 2.0;
    a = accel(t, mu, m);
    printf("Acceleration a = %f (m/s^2)\n", a);
    return 0;
}
double force(double t) {
    double p;

    p = 4*(t-3)+20;
    return p;
}
double accel(double t, double mu, double m) {
    double a, p;

    p = force(t);
    a = (p-mu*m*M_G)/m;
    return a;
}
```
Parameter names must appear in function definitions, but the parameter names for arguments of function prototypes does not have to be included. For example, the following two function prototypes are the same.

```c
double accel(double, double, double);
double accel(double t, double mu, double m);
```
Standard C Header Files and Libraries

- Header files typically contain function prototypes for performing operations that are related to one another. They may also include macros and type definitions required by some of the functions.

  For example, header file `math.h` contains the function prototypes

  ```
  extern double sin(double x);
  extern double exp(double x);
  ```

  for mathematical functions $\sin(x)$ and $e^x$. The type qualifier `extern` for functions defined in a library or other module.

- Each C compiler and interpreter has its own header files for the standard C libraries. For example, these header files in Ch are located in the directory `C:/Ch/include` in Windows and `/usr/local/ch/include` in Unix by default.
## Some Standard C Header Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>limits.h</td>
<td>Define several macros that expand to various limits and parameters of the standard integer types.</td>
</tr>
<tr>
<td>float.h</td>
<td>Define several macros that expand to various limits and parameters of the standard floating-point types.</td>
</tr>
<tr>
<td>math.h</td>
<td>Declare two types and several functions and define several macros for general mathematical operations.</td>
</tr>
<tr>
<td>stdbool.h</td>
<td>Define macros for boolean operations <em>(C99 and above)</em></td>
</tr>
<tr>
<td>stdio.h</td>
<td>Declare types, macros, and functions for standard input and output.</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>Declare several types, macros, and functions for general utility, such as conversion of numbers to text and text to numbers, memory allocation, and generation of random numbers.</td>
</tr>
<tr>
<td>time.h</td>
<td>Define macros and declare sever types and functions for manipulating time and date.</td>
</tr>
</tbody>
</table>
Standard C functions

• A reference manual with detailed description for each function in the standard C libraries is available in CHHOME/docs/chref.pdf, such as C:/ch/docs/chref.pdf. Show the function sin() in math.h

• Demonstration programs for each function in the standard C libraries are available in CHHOME/demos/lib/libc/, such as C:/Ch/demos/lib/libc/, directory. Show the example C:/Ch/demos/lib/libc/math/sin.c
The traditional function definition, known as K&R C, is still supported in C. In this notation, parameter identifiers in a function definition are separated by the declaration list.

Example:

```c
int addition(a, b)
int a;
int b;
{
    return a + b;
}
or
int addition(a, b)
int a, b;
{
    return a + b;
}
```

Don’t use this!
If the number of arguments passed to the called function is less than or more than the number of formal definitions, it is a syntax error.

**Example:**

```c
int funct1(int i, int j) {return i;}
funct1(8);  // ERROR: number of arguments is 1, need 2 arguments.

int funct2(int i) {return i;}
funct2(8, 9);  // ERROR: number of arguments is 2, need 1 argument.
```
• There can only be one function definition at a given lexical level in a program.

**Example:**

```c
int func1(int i){}
int func1(int i){}    // ERROR: redefine func1()
```
• Variables and function names cannot be the same at the same lexical level.

**Example:**

```c
int func2;
int func2(int i) {}  // ERROR: redefine func2()
```
• If a function is defined without return a value, the return data type should be **void**. It is an error to call a function with the return type of **void** in a context that requires a value.

**Example:**

```c
void funct(int i){
    ...
}
int k;
k = funct(3);  // Error: lvalue and rvalue
// are not compatible
```
Both system built-in and user-defined functions can be used as arguments of functions. System built-in functions will be handled polymorphically.

Furthermore, a function itself can be used as an argument of the function.

Example:

```c
int i;
int func1(int j)
{
    return 2*j;
}
int func2(int j1, int j2, int j3)
{
    return j1+j2+j3;
}
i = func2(abs(-6), func1(func1(2)), func2(1,2,3));
printf("i = %d \n", i);
```

Output:

```
i = 20
```
A Mathematical Formula with a Single Variable

Calculate function
\[
sinc(x) = \frac{\sin(x)}{x}
\]
from \(-10 \leq x \leq 10\) with a step size 5. Modify program for sinc.c.

Generate data points \((x, y)\) for \(x\) in the range \(x_0 \leq x \leq x_f\) with step size \(x_{\text{step}}\) linearly. The number of points

\[
n = \frac{(x_f - x_0)}{x_{\text{step}}} + 1;
\]

Each data point can be calculated by

\[
\text{for}\(i = 0; i < n; i++\) \{ \\
\quad x = x_0 + i*x_{\text{step}}; \\
\quad y = f(x); \\
\}
\]
/ File: sinc.c */
#include <stdio.h> /* for printf() */
#include <math.h>  /* for sin() and fabs() */
#include <float.h> /* for FLT_EPSILON */

double sinc(double x) {
    double retval;
    if(fabs(x) < FLT_EPSILON)
        retval = 1.0;
    else
        retval = sin(x)/x;
    return retval;
}

int main() {
    double x, x0, xf, xstep, result;
    int i, n;

    printf("    x       sinc(x)\n")
    printf("------------------\n")
    x0 = -10.0;            /* initial value */
    xf = 10.0;             /* final value */
    xstep = 5.0;           /* step size */
    n = (xf - x0)/xstep + 1; /* num of points */
    for(i = 0; i < n; i++) {
        x = x0 + i*xstep;    /* value x */
        result = sinc(x);
        printf("%8.4f %8.4f\n", x, result);
    }
    return 0;
}
Calculating Polynomial Functions

Note:
1. $x^2$ can be calculated using either $x*x$ or exponential function $\text{pow}(x, 2)$ declared in header file `math.h` with $\text{pow}(x, y)$ for $x^y$. The expression $x*x$ is faster than $\text{pow}(x, 2)$. 
Plotting with Ch

• 2D plotting of a function
• Plotting more than one function together
• 3D plotting of a function
2D Plotting Function

• `fplotxy()` is a high-level 2D plotting function.
• The prototype for function `fplotxy()` is as follow.

```
int fplotxy(double func(double),
            double x0, double xf, int num,
            char *title, char *xlabel, char *ylabel);
```

• The argument `func` is a pointer to function for a function to be plotted in the range from `x0` to `xf` with the `num` of points. The remaining arguments are for title, label for x-axis, and label for y-axis.
• The first argument of the function `fplotxy()` is a function with the function prototype of
  
  ```
  double func(double x);
  ```
• The function `fplotxy()` can be called as follows.
  ```
fplotxy(func, x0, xf, num, title, xlabel, ylabel);
```
Plot function

\[ \text{sinc}(x) = \frac{\sin(x)}{x} \]

from \(-10 \leq x \leq 10\) with 80 points.

/* File: sinc_fplotxy.c */
#include <math.h>
#include <chplot.h>

double sinc(double x) {
    double retval;

    retval = sin(x)/x;
    return retval;
}

int main() {
    double x0 = -10.0, xf = 10.0;
    int num = 80;

    fplotxy(sinc, x0, xf, num, "function sinc()", "x", "sinc(x)");
    return 0;
}
Output of the program:

function sinc()
Demo

1. Run program sinc_fplotxy.c in ChIDE.
2. Copy the plot into the Clipboard.
   a. Click upper left icon of the displayed plot,
   b. click Options,
   c. click Copy to Clipboard
3. Open Word, paste the copied plot into it.
Copy a plot into clipboard to be pasted in a Word file

function sin(\(x\))

\[\sin(x)\]

Created by Harry H. Cheng, © 2009 McGraw-Hill, Inc. All rights reserved.
• **Problem Statement**

For the system shown in Figure 1(a), given $m = 5$ kg, $g = 9.81$ m/s$^2$, $\mu = 0.2$. The external force $p$ is expressed as a function of time $t$,

$$p(t) = 4 \sin(t-3)+20 \quad \text{when } t \geq 0$$

(1) **Write a program to plot the accelerations $a$ when $t = 0$ to $t = 10$ with 11 points.**

(2) **Write a program to plot the accelerations $a$ during $t = 0$ to $t = 10$ with 100 points.**
/* File: accelsinplot.c */
#include <stdio.h>
#include <math.h>
#include <chplot.h>
#define M_G 9.81

double force(double t) {
    double p;
    p = 4*sin(t-3)+20;
    return p;
}

double accel(double t) {
    double mu, m, a, p;

    mu = 0.2;
    m = 5;
    p = force(t);
    a = (p-mu*m*M_G)/m;
    return a;
}

int main() {
    double t0 = 0, tf = 10;
    int num = 11;

    fplotxy(accel, t0, tf, num,
            "Acceleration Plot",
            "time (s)",
            "accel (m/s^2)");
    return 0;
}
Output of the program with num = 11:
Output of the program with num = 100:

Acceleration Plot
Using the plotting class CPlot to plot multiple functions in the same plot.

\[ \text{fplotxy(func, x0, xf, num, title, xlabel, ylabel);} \]

has the same effect as

\[
\begin{align*}
\text{CPlot plot;} \\
\text{plot.title(title);} \\
\text{plot.label(PLOT_AXIS_X, xlabel);} \\
\text{plot.label(PLOT_AXIS_Y, ylabel);} \\
\text{plot.func2D(x0, xf, num, func);} \\
\text{plot.plotting()}
\end{align*}
\]

CPlot is a data type defined in header file chplot.h. Others are member functions to handle the associated data for the object plot.
Plot two acceleration curves in a single plot

```c
/* File: accelfunc2D.c */
#include <stdio.h>
#include <math.h>
#include <chplot.h>
#define M_G 9.81

double accel(double t) {
    double mu, m, a, p;
    mu = 0.2;
    m = 5;
    p = 4*sin(t-3)+20;
    a = (p-mu*m*M_G)/m;
    return a;
}

int main() {
    double t0 = 0, tf = 10;
    int num1 = 11, num2 = 100;
    CPlot plot;
    plot.title("Acceleration Plot");
    plot.label(PLOT_AXIS_X, "time (s)");
    plot.label(PLOT_AXIS_Y, "accel (m/s^2)");
    plot.func2D(t0, tf, num1, accel);
    plot.legend("11 points", 0);
    plot.func2D(t0, tf, num2, accel);
    plot.legend("100 points", 1);
    plot.plotting();
    return 0;
}
```

Member function call

```c
plot.legend("string", num);
```
adds a legend of string for the data set num. Numbering of data set starts with zero.
Output of the program

Acceleration Plot

11 points
100 points
A Mathematical Formula with Multiple Variables

Calculate function

\[ f_2(x, y) = \frac{\sin(\sqrt{x^2 + y^2})}{\sqrt{x^2 + y^2}} \]

for \( x \) from \(-10 \leq x \leq 10\) with a step size 10 and

\( y \) from \(-10 \leq y \leq 10\) with a step size of 10.
/* File: sinr.c */
#include <stdio.h>
#include <math.h>

double sinr(double x, double y) {
    double retval, r;
    
    r = sqrt(x*x + y*y);
    retval = sin(r)/r;
    return retval;
}

int main() { 
    double x, x0, xf, xstep,
            y, y0, yf, ystep, result;
    int i, j, nx, ny;

    printf("      x          y      sinr(x,y)\n");
    printf("  --------------------------------\n");
    x0 = -10.0;
    xf = 10.0;
    xstep = 10.0;
    nx = (xf - x0)/xstep + 1; /* num of points for x */
C for Engineers and Scientists: An Interpretive Approach

**Output:**

\[
\begin{array}{ccc}
\hline
x & y & \text{sinr}(x,y) \\
\hline
-10.0000 & -10.0000 & 0.0707 \\
-10.0000 & 0.0000 & -0.0544 \\
-10.0000 & 10.0000 & 0.0707 \\
0.0000 & -10.0000 & -0.0544 \\
0.0000 & 0.0000 & \text{NaN} \\
0.0000 & 10.0000 & -0.0544 \\
10.0000 & -10.0000 & 0.0707 \\
10.0000 & 0.0000 & -0.0544 \\
10.0000 & 10.0000 & 0.0707 \\
\hline
\end{array}
\]

\[
\text{sin}(0)/0 \text{ is NaN}
\]

\[
y_0 = -10.0; \\
y_f = 10.0; \\
y_{\text{step}} = 10.0; \\
ny = (y_f - y_0)/y_{\text{step}} + 1; /* \text{num of points for y} */
\]

\[
\text{for}(i = 0; \ i < \text{nx}; \ i++) \ { \\
\quad x = x_0 + i*\text{xstep}; /* \text{calculate value for x} */ \\
\quad \text{for}(j = 0; \ j < \text{ny}; \ j++) \ { \\
\quad \quad y = y_0 + j*y_{\text{step}}; /* \text{calculate value for y} */ \\
\quad \quad \text{result} = \text{sinr}(x, y); \\
\quad \quad \text{printf}("%10.4f %10.4f %8.4f\n", \ x, \ y, \ \text{result}); \\
\quad \}
\}
\]

\[
\text{return} \ 0;
\]

Created by Harry H. Cheng, © 2009 McGraw-Hill, Inc. All rights reserved.
3D Plotting Function

- **fplotxyz()** is a plotting function for functions with two variables.
  - The prototype for function **fplotxyz()** is as follows:
    ```c
    int fplotxyz(double (*func)(double, double),
                  double x0, double xf, double y0, yf,
                  int numx, int numy, char *title,
                  char *xlabel, char *ylabel, char *zlabel);
    ```

- The argument **func** is a pointer to a function for a function to be plotted for
  x-coordinates in the range from \(x_0\) to \(x_f\) with the \(\text{num}_x\) of points and for
  the y-coordinates in the range from \(y_0\) to \(y_f\) with the \(\text{num}_y\) of points. The
  remaining arguments are for title, labels for x-axis, y-axis, and z-axis.
- The first argument of the function **fplotxyz()** is a function with the
  function prototype of
    ```c
    double (*func)(double x, double y);
    ```
- The function **fplotxyz()** can be called as follows.
  ```c
  fplotxy(func, x0, xf, y0, yf, numx, numy,
          title, xlabel, ylabel, zlabel);
  ```
Plot function \( f_2(x, y) = \frac{\sin(\sqrt{x^2 + y^2})}{\sqrt{x^2 + y^2}} \)

for \( x \) from \(-10 \leq x \leq 10\) with 80 points and

\( y \) from \(-10 \leq y \leq 10\) with 100 points.

/* File: hat.c */
#include <math.h>
#include <chplot.h>

double sinr(double x, double y) {
    double retval, r;

    r = sqrt(x*x + y*y);
    retval = sin(r)/r;
    return retval;
}

int main() {
    double x0 = -10.0, xf = 10.0, y0 = -10.0, yf = 10.0;
    int numx = 80, numy = 100;

    fplotxyz(sinr, x0, xf, y0, yf, numx, numy,
             "function sinr(x, y)", "x", "y", "sinr(x, y)");
    return 0;
}
Output of the program:

```
function sinr(x, y)
```

The diagram illustrates the function `sinr(x, y)` in a 3D graph, showing the output values ranging from -10 to 10 on the x and y axes, and from -0.4 to 1 on the z axis.
Recursive Functions

- Functions may be used recursively. This means that a function can call itself directly.
- When a function calls itself recursively, each function call will have a new set of local variables.
- Recursive functions usually contain conditional statements, such as if-else, to allow exit of the function and return control to the calling function.
- A function may also call itself indirectly.
Example:
Calculating a factorial 5! using a function with a for-loop.
The factorial n! is defined as n*(n-1)!

```c
/* File: factorloop.c */
#include <stdio.h>

unsigned int factorial(int n);
int main() {
    int i;
    for(i=0; i<=5; i++)
        printf("%d! = %d\n", i, factorial(i));
    return 0;
}

unsigned int factorial(int n) {
    unsigned int i, f;
    for(i=1, f=1; i<=n; i++) {
        f *= i;
    }
    return f;
}
```

Output:

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
Example:
Calculating a factorial 5! using a recursive function.

/* File: factorrecursive.c */
#include <stdio.h>

unsigned int factorial(int n);

int main() {
    int i;
    for(i=0; i<=5; i++)
        printf("%d! = %u\n", i, factorial(i));
    return 0;
}

unsigned int factorial(int n) {
    if(n <= 1) {
        return 1;
    } else {
        return n*factorial(n-1);
    }
}

Output:
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
int main()
{
    int f = 2;
}

int factorial(int n){ // n=2
    if(n<=1)
        return 1;
    else
        return 2*factorial(1);
}

int factorial(int n){ // n=2
    if(n<=1)
        return 1;
    else
        return 2*1;
}
**Example:** (indirect recursive function call)

```c
/* File: recursiveindirect.c */
#include <stdio.h>
int main() {
    funct1(1);
    return 0;
}
int funct1(int i) {
    i = funct2(i);
    printf("exit funct1() i = %d \n", i);
    return i;
}
int funct2(int j) {
    if(j <= 3) {
        printf("recursively call funct2() j = %d \n", j);
        j++;
        j = funct1(j);
    }
    else {
        printf("exit funct2() j = %d \n", j);
        j++;
    }
    return j;
}
```

**Output:**

recursively call funct2() j = 1
recursively call funct2() j = 2
recursively call funct2() j = 3
exit funct2() j = 4
exit funct1() i = 5
exit funct1() i = 5
exit funct1() i = 5
exit funct1() i = 5
Predefined Identifier `__func__`

- The data type for `__func__` is `char*`.
- Inside a function, it is a pointer to string obtaining the function name.

**Example:**

```c
/* File: funcname.c */
#include <stdio.h>

void funcname(void) {
    printf("The function is %s\n", __func__);
}

int main() {
    funcname();
    return 0;
}
```

**Output:**

The function is funcname
Variable Number Arguments in Functions

- A variable number of arguments can be passed to a function.
- In some applications, the number of arguments passed to a function are unknown and could be different for different cases.
- A typical function which takes a variable number of arguments is defined below.

```c
#include<stdio.h>

type1 func_name(arg_list, type2 paramN, ...)
{
    va_list ap;
    type3 v;            // first unnamed argument
    va_start(ap, paramN); // initialize the list
    v = va_arg(ap, type3);   // get 1st unnamed argument from the list
    ...                    // get the rest of the list
    va_end(ap);            // clean up the argument list
    ...
}
```

where `arg_list` is the argument list of the named argument, `paramN` is the last named argument and `v` is the first unnamed argument of type `type3`. 
Below is a table of macros defined in `stdarg.h` for handling variable argument list.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>va_arg</code></td>
<td>Expands to an expression that has the specified type and the value of the next argument in the calling function.</td>
</tr>
<tr>
<td><code>va_arraydim</code></td>
<td>Obtain the array dimension of the variable argument.</td>
</tr>
<tr>
<td><code>va_arrayextent</code></td>
<td>Obtain the extent for a specified dimension in the array of variable argument.</td>
</tr>
<tr>
<td><code>va_arraynum</code></td>
<td>Obtain the number of elements in the array of variable argument.</td>
</tr>
<tr>
<td><code>va_arraytype</code></td>
<td>Obtain the array type in the array of variable argument.</td>
</tr>
<tr>
<td><code>va_datatype</code></td>
<td>Obtain the data type of variable argument.</td>
</tr>
<tr>
<td><code>va_copy</code></td>
<td>Makes a copy of the <code>va_list</code>.</td>
</tr>
<tr>
<td><code>va_count</code></td>
<td>Obtain the number of variable arguments.</td>
</tr>
<tr>
<td><code>va_end</code></td>
<td>Facilitates a normal return from the function.</td>
</tr>
<tr>
<td><code>va_start</code></td>
<td>Initializes <code>ap</code> for subsequent use by other macros.</td>
</tr>
</tbody>
</table>
Example:

The example below calculates the area of a circle, a square, or a rectangle. The first argument of function `area()` specifies the shape whose area will be evaluated. The rest of arguments are the specifications related to the shape. The number of arguments are different for different shapes. In the cases of a circle or a square, one more argument is needed to specify the radius for a circle or the side for a square. For a rectangle, two more arguments are needed to specify the length and width of a rectangle.
Example:

Example:

/* File: vararg.c */
#include <stdio.h>
#include <stdarg.h>
#include <math.h> // for M_PI
#define CIRCLE 1
#define SQUARE 2
#define RECTANGLE 3

double area (int type, ...) {
    va_list ap;
    double radius;
    double side;
    double length, width;
    double a = 0.0; /* default, 0*/
    va_start(ap, type);
    switch(type) {
    case CIRCLE:
        radius = va_arg(ap, double);
        a = M_PI*radius*radius;
        break;
    case SQUARE:
        side = va_arg(ap, double);
        a = side*side;
        break;
    case RECTANGLE:
        length = va_arg(ap, double);
        width = va_arg(ap, double);
        a = length*width;
        break;
    default:
        printf("Error: unknown shape\n");
        break;
    }
    va_end(ap);
    return a;
}

int main() {
    double radius = 2;
    double side = 2;
    double length = 3, width = 2;
    double a;
    a = area(CIRCLE, radius);
    printf("area of circle with radius 2 meters is %f\n", a);
    a = area(SQUARE, side);
    printf("area of square with side 2 meters is %f (m^2) \n", a);
    a = area(RECTANGLE, length, width);
    printf("area of rectangle with length 3 m and width 2 m is %f (m^2) \n", a);
    return 0;
}