Imperative programming

7. Functions

Zoltán Porkoláb

Functions

- Functions are **subprograms**, the fundamental building blocks in C
- The language pragmatics prefers writing many small functions
- The compilers can optimize functions and function calls, like **inlining**
- Functions can r**eturn** with a value of some type
- Functions without return value are declared as returning **void**
- Parameters are **passed by value** (i.e. parameters are copied)
- **Recursive** functions are allowed
- **main** is a specifial function (but may have parameters and can be recursive)

Function declaration

- Can appear as global or local declaration
- Prototype declaration: specifies the parameter types
- Non-prototype declaration: does not specifies the parameters
- Variadic parameter list is expressed by ellipsis (…)
- Zero parameter is declared by **(void)**

```
extern double fahr2cels(double f); // global prototype decl.
void f(void);  // global prototype decl., zero parameter

void f(void)   // function definition with zero parameter
{
   extern double cels2fahr(); // local, non-prototype decl.
   extern int    func(void);  // local, prototype, no parameter
   void print(const char *c, ...);  // ellipsis: zero or more
}
```

Function declaration

prototype

signature

formal parameter

actual parameter

argument

```
extern double fahr2cels(double f);

void f(void)
{
   extern double cels2fahr();
   extern int    func(void);
   void print(const char *c, ...);

   double d1 = fahr2cels('A');
   double d2 = cels2fahr('A');
   print("%d %s %l %f", 42, "Hello", 5L, 3.14);
}
```

Function declaration

- Formal parameters can be ignored
- Prototype declaration: conversion on parameter passing
- Non-prototype declaration: only promotions on parameter passing
- Variadic parameter list: only promotions on parameter passing on ellipsis (…)
- Conversion on parameter passing only when prototype declared

```
extern double fahr2cels(double f); // prototype decl.
extern double fahr2cels(double);   // same as above
       double fahr2cels(double);   // same as above

extern double cels2fahr();         // non-prototype decl.
void print(const char *c, ...);    // ellipsis: zero or more

double d1 = fahr2cels('A'); // char 'A' is converted to double
double d2 = cels2fahr('A'); // char 'A' is promoted to int
print("%d %s %l %f", 42,"Hello",5L,3.14); // no conversion
```

Array parameters

- Functions cannot return an array
- Array parameters are automatically converted to pointer parameters
- The leftmost array dimension can be empty in parameter declarations
- Only the leftmost dimension can be empty

```
int *find_first_odd(int *t,     int len); // ok
int *find_first_odd(int t[10], int len); // ok, same as above
int *find_first_odd(int t[],   int len); // ok, same as above

double f(double   (*d)[200]);  // ok, ?x200 array of doubles
double f(double d[100][200]);  // ok, same as above
double f(double    d[][200]);  // ok, same as above
double f(double    d[][]); // error: only leftmost could empty

int[] find_all(int *t, int len); // error: no return with array
int  *find_all(int *t, int len); // ok, can return pointer
```

Function definitions

- Function body is connected with the function name
- Only global definitions, there is no local (nested) function definition
- Function can be static (internal linkage) or global (external linkage)

```
static int *find_first_odd( int *t, int len)  // int t[]
{
  for (int i = 0; i < len; ++i)
  {
    if ( t[i] % 2 ) // *(t+i)
    {
      return t+i;  // &t[i]
    }
  }
  return NULL;
}
```

*Functions an be **inline** since C99 – but we do not learn them here

Function definitions

- **No overloading**, all function names must be unique

```
#include <stdlib.h>
int           abs(int n);
long          labs(long n);
long long     llabs(long long n);    // C99
intmax_t      imaxabs( intmax_t n); // C99

#include <math.h>
double        fabs(double n);
float         fabsf(float n);        // C99
long double fabsl(long double n); // C99
```

- Other languages: C++, Java, C#, Rust, etc. allows if **signature is different**

```
#include <cmath>    // C++

int        abs(int n);
long       abs(long n);
long long abs(long long n);
double     abs(double n);
// ...
```

Function call

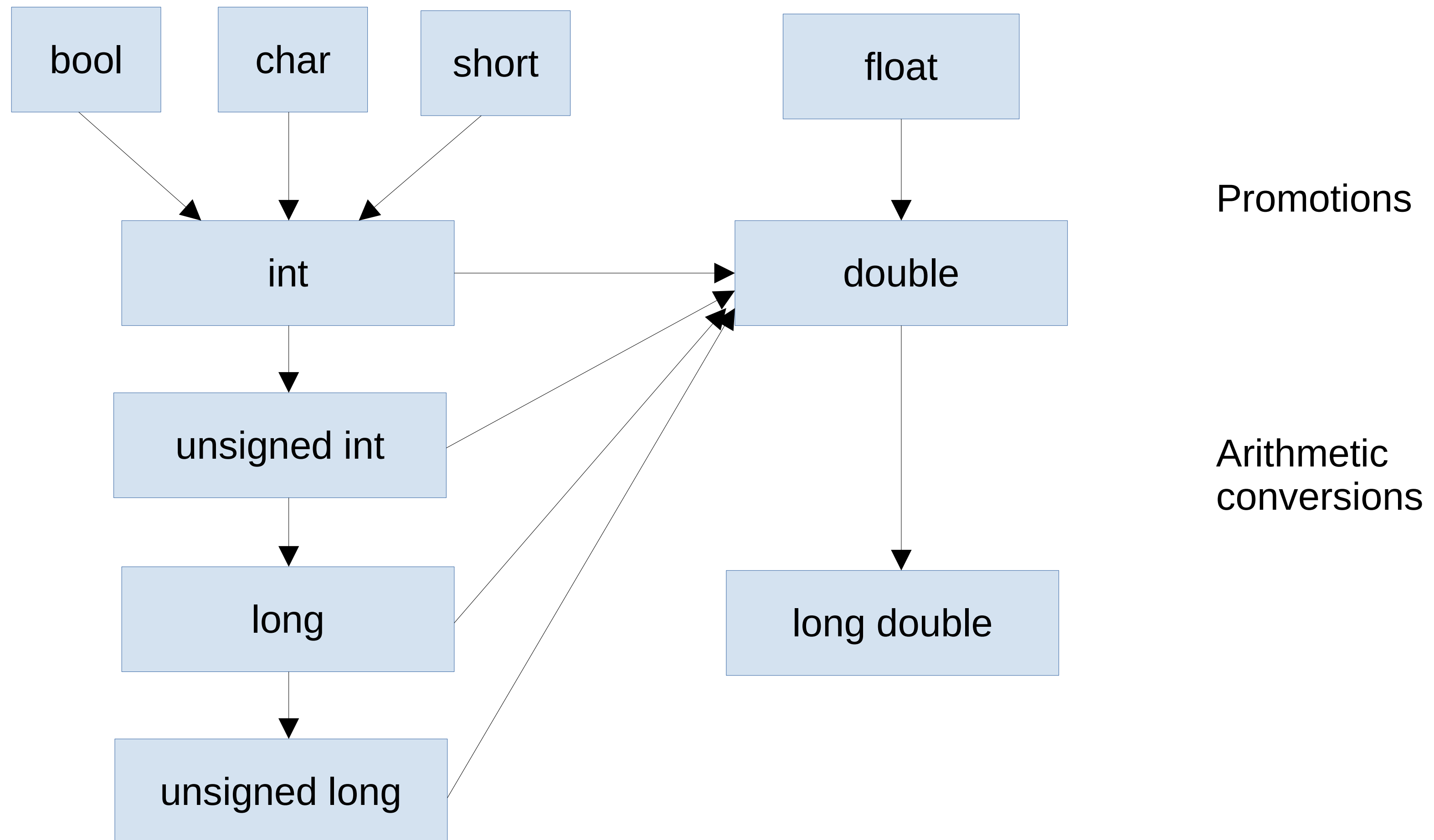
- If the declaration is with prototype
  - The number of formal and actual parameters are the same
  - Each actual parameter is assignable to the type of the formal parameter
  - Each actual parameter is converted to the type of the formal parameter

```
int *find_first_odd(int *t, int len);

int arr[20];
long l = 20;
int *p = find_first_odd(arr,l); // ok, l converted to int
int *q = find_first_odd(arr);  // error: number of parameters
int *r = find_first_odd(arr,"H"); // error: no char* to long
```

(On the slide, the `*q` and `*r` lines are struck through, but not their comments.)

- If the declaration is no prototype
  - Default promotions only
  - The number of parameters are ignored

Implicit conversions



Function call

- A function call is a sequence point
  - All parameters are evaluated before the execution of the function body
  - The order of parameter evaluation is unspecified
- The return statement is also using conversions

```
#include <stdio.h>

int f() {   printf("f\n");  return 2;  }
int g() {   printf("g\n");  return 1;  }
int h() {   printf("h\n");  return 0;  }

void  func() { printf("(f() == g() == h()) == %d\n",
                                        f() == g() == h()); }

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

Recursive functions

- A function is recursive when calls itself
  - Directly
  - Indirectly via other functions
- Infinite recursion is a (run-time) error

```
int factorial( int n)
{

  if ( 1 == n )
    return 1;
  else
    return n*factorial(n-1);
}
```

Recursive functions

- A function is recursive when calls itself
  - Directly
  - Indirectly via other functions
- Infinite recursion is a (run-time) error

```
#include <assert.h>

int factorial( int n)
{
  assert ( n >= 1 ); // defending against infinite recursion
  if ( 1 == n )
    return 1;
  else
    return n*factorial(n-1);
}
```

Recursive functions

- Recursive functions usually more readable, understandable, but less effective
- If the very last statement of the function is the recursive call: **tail recursion**
- Many compilers are capable automatically rewrite tail-recursion to loop

```
#include <assert.h>

int factorial( int n)
{
  assert ( n >= 1 ); // defending against infinite recursion
  int result = 1;
  for ( int i = 2; i <= n; ++i )
  {
    result *= i;
  }
  return result;
}
```

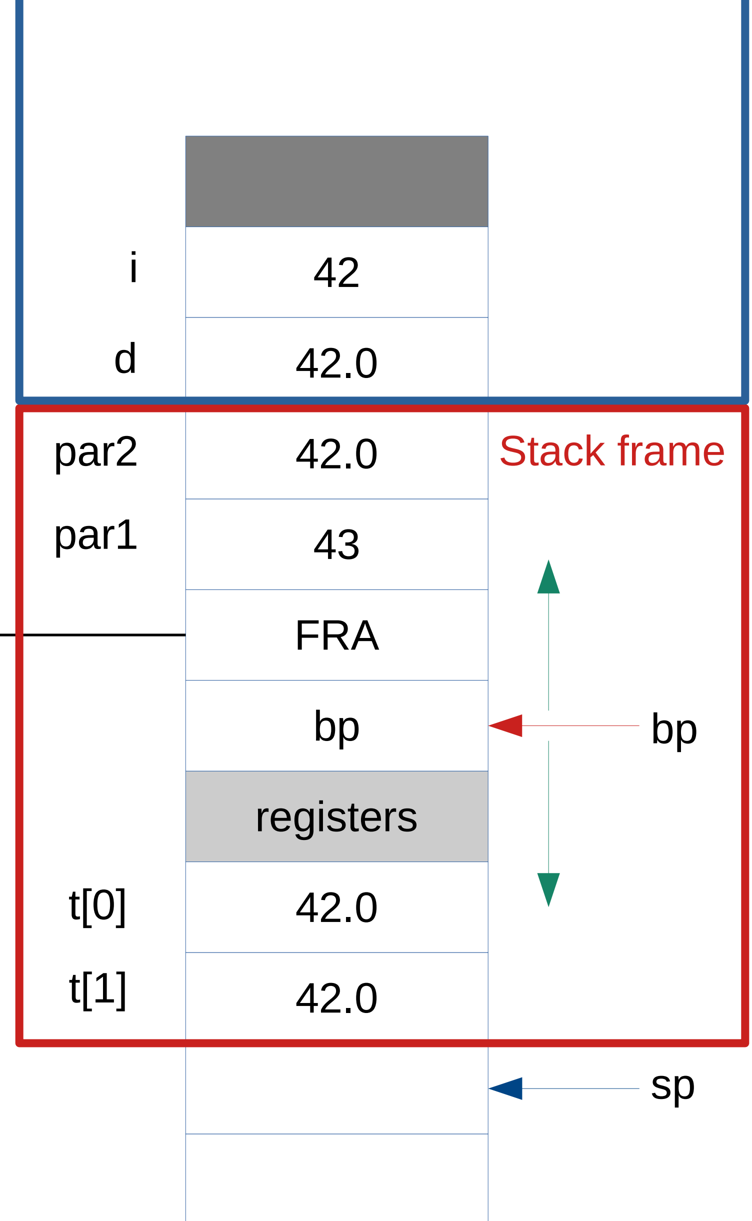
Parameter passing

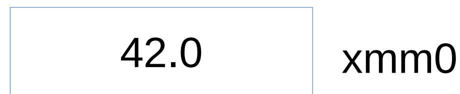
- Call by address (call by reference) (FORTRAN, C++ reference, Java, Pascal)
  - The formal parameter has the same memory address as the actual one
- Call by value (C, C++, Pascal “var”)
  - The actual parameter is copied to the formal parameter
- Call by result (ADA “inout”)
  - Same as by value but at the end of function formal parameter is copied back to the memory of the actual parameter
- Call by name (Algol60, Simula67)
  - The formal parameter is a closure evaluated each time it referred

Parameter passing

- **Call by address (call by reference) (FORTRAN, C++ reference, Java, Pascal)**
  - The formal parameter has the same memory address as the actual one
- **Call by value (C, C++, Pascal “var”)**
  - The actual parameter is copied to the formal parameter
- Call by result (ADA “inout”)
  - Same as by value but at the end of function formal parameter is copied back to the memory of the actual parameter
- Call by name (Algol60, Simula67)
  - The formal parameter is a closure evaluated each time it referred

```
void f(void)
{
   int i;
   double d;
   i = 42;
   d = i;
   d = g( i, d);
   i++;
}
double g( int par1, double par2)
{
   double t[2];
   t[0] = par1;
   t[1] = par2;
   par1++;
   return t[0];
}
```





xmm0

Parameter passing in C

- **By value:** the formal parameter is a local variable initialized by the actual one
- Finishing the function execution the parameter is going out of scope and life
- Any modifications on the parameter is happen on a copy and lost

```
#include <stdio.h>

void increment(int i)
{
  ++i;
  printf("i in increment() = %d\n",i);
}

int main()
{
  int i = 0;
  increment(i);
  increment(i);
  printf("i in main() = %d\n",i);
  return 0;
}
```

Parameter passing in C

- **By value:** the formal parameter is a local variable initialized by the actual one
- Finishing the function execution the parameter is going out of scope and life
- Any modifications on the parameter is happen on a copy and lost

```
#include <stdio.h>

void increment(int i)
{
  ++i;
  printf("i in increment() = %d\n",i);
}

int main()
{
  int i = 0;
  increment(i);
  increment(i);
  printf("i in main() = %d\n",i);
  return 0;
}
```

```
$ ./a.out
i in increment() = 1
i in increment() = 1
i in main() = 0
$
```

Parameter passing in C

- **By address** can be simulated by passing a pointer instead of the value
- This is the way **input** functions e.g., **scanf** are working
- Array parameters are pointers, the work both input and output

```
#include <stdio.h>

void increment(int *ip)
{
  ++*ip;
  printf("i in increment() = %d\n",*ip);
}

int main()
{
  int i = 0;
  increment(&i);
  increment(&i);
  printf("i in main() = %d\n",i);
  return 0;
}
```

Parameter passing in C

- **By address** can be simulated by passing a pointer instead of the value
- This is the way **input** functions e.g., **scanf** are working
- Array parameters are pointers, the work both input and output

```
#include <stdio.h>

void increment(int *ip)
{
  ++*ip;
  printf("i in increment() = %d\n",*ip);
}

int main()
{
  int i = 0;
  increment(&i);
  increment(&i);
  printf("i in main() = %d\n",i);
  return 0;
}
```

```
$ ./a.out
i in increment() = 1
i in increment() = 2
i in main() = 2
$
```

Parameter passing in C

- **By address** can be simulated by passing a pointer instead of the value
- This is the way **input** functions e.g., **scanf** are working
- Array parameters are pointers, the work both input and output

```
#include <stdio.h>

void read(void)
{
  int i;
  double d;
  char c;
  char buffer[20];

  if ( 4 == scanf("%d %f %c %19s", &i, &d, &c, buffer) )
  {
    // all items was read successfully
  }
}
```

Parameter passing in C

- Array parameters are automatically converted to pointer parameters
- Sizeof information is loosing!!!

```
int *find_first_odd( int *t)
{
  for (int i = 0; i < sizeof(t)/sizeof(t[0]); ++i)
    if ( t[i] % 2 )
      return t+i;
  return NULL;
}

int main()
{
  int numbers[] = {1,2,3,4,5,6,7,8};
  for (int i = 0; i < sizeof(t)/sizeof(t[0]); ++i)
    printf("%d, ", numbers[i]);
  int *first_odd_ptr = find_first_odd(numbers);
  if ( first_odd_ptr )
    printf(" first odd is %d\n", *first_odd_ptr);
  return 0;
}
```

Parameter passing in C

- Array parameters are automatically converted to pointer parameters
- Sizeof information is loosing!!!

```
int *find_first_odd( int *t)  // wrong!!!
{
  for (int i = 0; i < sizeof(t)/sizeof(t[0]); ++i) // sizeof(ptr)/
    if ( t[i] % 2 )                                 //     sizeof(int)
      return t+i;
  return NULL;
}

int main()
{
  int numbers[] = {1,2,3,4,5,6,7,8};
  for (int i = 0; i < sizeof(t)/sizeof(t[0]); ++i) // works here
    printf("%d, ", numbers[i]);
  int *first_odd_ptr = find_first_odd(numbers);
  if ( first_odd_ptr )
    printf(" first odd is %d\n", *first_odd_ptr);
  return 0;
}
```

Parameter passing in C

- Array parameters are automatically converted to pointer parameters
- Sizeof information is loosing!!!

```
int *find_first_odd( int *t, int len)  // ok
{
  for (int i = 0; i < len; ++i) // ok
    if ( t[i] % 2 )
      return t+i;
  return NULL;
}

int main()
{
  int numbers[] = {1,2,3,4,5,6,7,8};
  for (int i = 0; i < sizeof(t)/sizeof(t[0]); ++i) // works here
    printf("%d, ", numbers[i]);
  int *first_odd_ptr = find_first_odd(numbers,sizeof(t)/sizeof(t[0]));
  if ( first_odd_ptr )
    printf(" first odd is %d\n", *first_odd_ptr);
  return 0;
}
```

Parameters of main( )

- The **main()** is a special function
- The **argc** parameter is the number of command line arguments (incl. the program)
- The **argv** is an array of argc+1 element with the command line arguments
- If **argc** is defined then **argv[argc]** is NULL pointer

```
int main(void);
int main( int argc, char *argv[]);
int main( int argc, char *argv[], char *envp[]); // if supported by OS
```

Parameters of main( )

- The **main()** is a special function
- The **argc** parameter is the number of command line arguments (incl. the program)
- The **argv** is an array of argc+1 element with the command line arguments
- If **argc** is defined then **argv[argc]** is NULL pointer

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("name of the program = %s\n", argv[0]);
  for (int i = 1; i < argc; ++i)
    printf("argv[%d] = %s\n", i, argv[i]);
  return 0;
}
```

Parameters of main( )

- The **main()** is a special function
- The **argc** parameter is the number of command line arguments (incl. the program)
- The **argv** is an array of argc+1 element with the command line arguments
- If **argc** is defined then **argv[argc]** is NULL pointer

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("name of the program = %s\n", argv[0]);
  for (int i = 1; i < argc; ++i)
    printf("argv[%d] = %s\n", i, argv[i]);
  return 0;
}
$ gcc -ansi -std=c99 -Wall -W -o mainpars mainpars.c
$ ./mainpars
name of the program = ./mainpars
$ ./mainpars first second third
name of the program = ./mainpars
argv[1] = first
argv[2] = second
argv[3] = third
```

Parameters of main( )

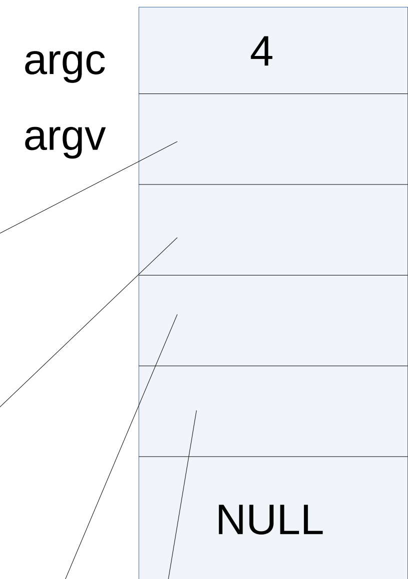
- The **main()** is a special function
- The **argc** parameter is the number of command line arguments (incl. the program)
- The **argv** is an array of argc+1 element with the command line arguments
- If **argc** is defined then **argv[argc]** is NULL pointer

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("name of the program = %s\n", argv[0]);
  for (int i = 1; i < argc; ++i)
    printf("argv[%d] = %s\n", i, argv[i]);
  return 0;
}

$ ./mainpars first second third
```

Parameters of main( )

- The **main()** is a special function
- The **argc** parameter is the number of command line arguments (incl. the program)
- The **argv** is an array of argc+1 element with the command line arguments
- If **argc** is defined then **argv[argc]** is NULL pointer

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("name of the program = %s\n", argv[0]);
  for (int i = 1; i < argc; ++i)
    printf("argv[%d] = %s\n", i, argv[i]);
  return 0;
}

$ ./mainpars first second third
```



READ-ONLY

.	/	m	a	i	n	p	a	r	s	\0	f	i	r	s	t	\0

Function pointers

- Pointers to function exist and identifies functions
- The type of the function pointer includes the whole prototype (incl. return type)
- Pointers with non-prototype declarations also exists
- The name of the function represents the pointer value

```
double sin(double); // or #include <math.h>

double f(void)
{
  double result = 0.0;
  double (*fp)(double);  // fp is a pointer to double(double)

  fp = sin;  // fp now points to double sin(double)

  if ( NULL != fp )
  {
    result = (*fp)(0.5);    // calls sin(0.5)
    result =    fp(0.5);    // same as above
  }
  return result;
}
```

Function pointers

- Pointers can point to compatible functions
- Assignable to function pointer to the same type (or compatible non-prototype)
- Can be compared to NULL
- Can be call the pointed function, if the pointer value is not NULL

```
double sin(double); // or #include <math.h>

double f(void)
{
  double result = 0.0;
  double (*fp)(double);  // fp is a pointer to double(double)

  fp = sin;  // fp now points to double sin(double)

  if ( NULL != fp )
  {
    result = (*fp)(0.5);    // calls sin(0.5)
    result =    fp(0.5);    // same as above
  }
  return result;
}
```

Function pointers

```c
#include <stdio.h>

void increment(int *ip)
{
  ++*ip;
  printf("i in increment() = %d\n",*ip);
}

int main()
{
  void (*fp)(int *);  // pointer to void(int *) prototype
  void (*gp)();       // pointer to void()  non-prototype
  int i = 0;

  fp = increment;
  gp = increment;
  fp(&i);             // calls increment(&i)
  gp(&i);             // calls increment(&i) but no checks
  printf("i in main() = %d\n",i);
  return 0;
}
```

typedef

- Typedef creates a new type synonym
- Typedef does not create a new type

```
typedef double length_t;     // length_t is synony of double
typedef int    index_t;      // index_t is synonym of int
typedef struct date date_t;  // date_t is synonym of struct date

typedef char   *ptr_to_char_t; // ptr_to_char_t is synonym of char *

typedef double (*trigfp_t)(double); // trigfp_t is a type which can
                                   // point to double sin(double)

length_t   len;  // len is double
index_t      i;  // i is int
date_t    exam;  // exam is a record of struct date
ptr_char_t ptr;  // ptr is char *
trigfp_t    fp;  // fp is a pointer to double(double)
```

Typedef and function pointer

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

typedef double (*trigfp_t)(double);
trigfp_t inverse(trigfp_t fun)
{
  static trigfp_t from[] = {  sin,  cos,  tan };
  static trigfp_t to[]   = { asin, acos, atan };

  int i = 0;

  for ( i = 0; i < sizeof(from)/sizeof(from[0]); ++i)
    if ( fun == from[i] ) return to[i];
  return fun;
}
int main()
{
   double   d1  = sin(.5);
   trigfp_t rev = inverse(sin);
   double   d2  = rev(d1);

   printf( "sin(.5) = %f, arc sin(%f) = %f\n", d1, d1, d2);
   return 0;
}
```

Typedef and function pointer

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

typedef double (*trigfp_t)(double);
trigfp_t inverse(trigfp_t fun)
{
  static trigfp_t from[] = {  sin,  cos,  tan };
  static trigfp_t to[]   = { asin, acos, atan };

  int i = 0;

  for ( i = 0; i < sizeof(from)/sizeof(from[0]); ++i)
    if ( fun == from[i] ) return to[i];
  return fun;
}
int main()
{
   double   d1  = sin(.5);
   trigfp_t rev = inverse(sin);
   double   d2  = rev(d1);

   printf( "sin(.5) = %f, arc sin(%f) = %f\n", d1, d1, d2);
   return 0;
}
```

```
$ gcc -ansi -pedantic -Wall inverse.c -lm
$ ./a.out
sin(.5) = 0.479426, arc sin(0.479426) = 0.500000
```