KU LEUVEN

C: an introduction

Functions

Program: building blocks

- Variables
  - Store data (input, intermediate values, results)
- Expressions
  - Manipulate variables
- Control structures
  - Make decisions (if) or repeat (for, while) statements
- Functions
  - for parameterization and re-use

KU LEUVEN

Example: calculate surface triangle

- Calculate surface for several triangles
  - Write code over and over
  - Write the code once: function
- Files:
  - surface-bruteforce.c
  - surface-function.c

KU LEUVEN


```c
/*
surface-bruteforce.c
calculate the surface of a triangle
*/
#include<stdio.h>

int main(void){

double surftri = 0;  // surface of trangle
double height = 0;   // height
double base = 0;     // base

/* triangle 1 */

base = 10.8;
height = 6.7;
surftri = (base * height) / 2.0;

printf(" triangle - height %7.3f - base %7.3f = surface %7.3f \n", height, base, surftri);

/* triangle 2 */

base = 4.8;
height = 2.0;
surftri = (base * height) / 2.0;

printf(" triangle - height %7.3f - base %7.3f = surface %7.3f \n", height, base, surftri);

/* triangle 3 */

base = 188.8;
height = 65.7;
surftri = (base * height) / 2.0;

printf(" triangle - height %7.3f - base %7.3f = surface %7.3f \n", height, base, surftri);

return 0;
}
```

```c
/*
surface-function.c
calculate the surface of a triangle
*/
#include<stdio.h>

double surftri(double base, double height);  // function declaration

int main(){

double height = 0;   // height
double base = 0;     // base
double surf1 = 0, surf2 = 0;

/* triangle 1 */

base = 10.8;
height = 6.7;
surf1 = surftri(base, height);

printf(" triangle - height %7.3f - base %7.3f = surface %7.3f \n", height, base, surf1);

/* triangle 2 */

base = 4.8;
height = 2.0;
surf2 = surftri(4.8, 2.0);

printf(" triangle - height %7.3f - base %7.3f = surface %7.3f \n", 4.8, 2.0, surf2);

/* triangle 3 */

printf(" triangle - height %7.3f - base %7.3f = surface %7.3f \n", 65.7, 188.8, surftri(188.8, 65.7));

return 0;
}

/* function definition
include the print statement or not */
double surftri(double base, double height)
{ double result;
result = (base * height) / 2.0;
return (result);
}
```

KU LEUVEN

Functions: The key to scalable software

- Problem solving boils down to defining functions that perform a single service.
- Most real programs consist of a number of (usually quite small) functions.
- Functions break large problems into smaller ones that can be solved easily.
- Functions facilitate code reuse.
  Less code duplication
- Functions hide algorithm details behind intuitive interfaces.

Functions: The key to scalable software

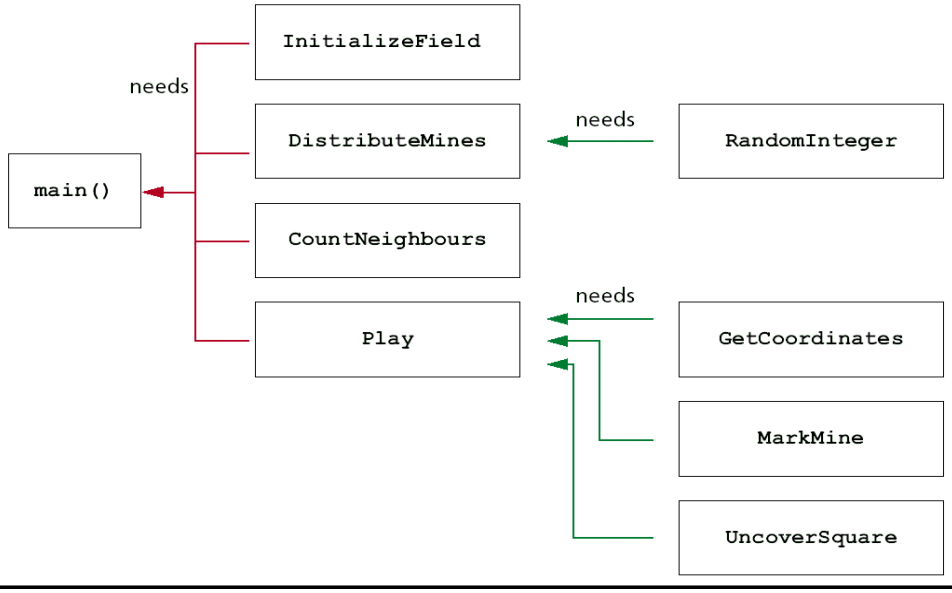
- Functions are an abstraction mechanism that allow code to be understood at a higher level than is possible from plain code.
- Each function can be stored in a separate file
- Debugging gets easier – each function can be tested separately
- Easier to read code / maintenance

Functions: The key to scalable software

- Programs are built up of functions
- Functions
  - have a unique name
  - take in arguments
  - compute something
  - return a result
- The `main()` function
  - where program execution starts

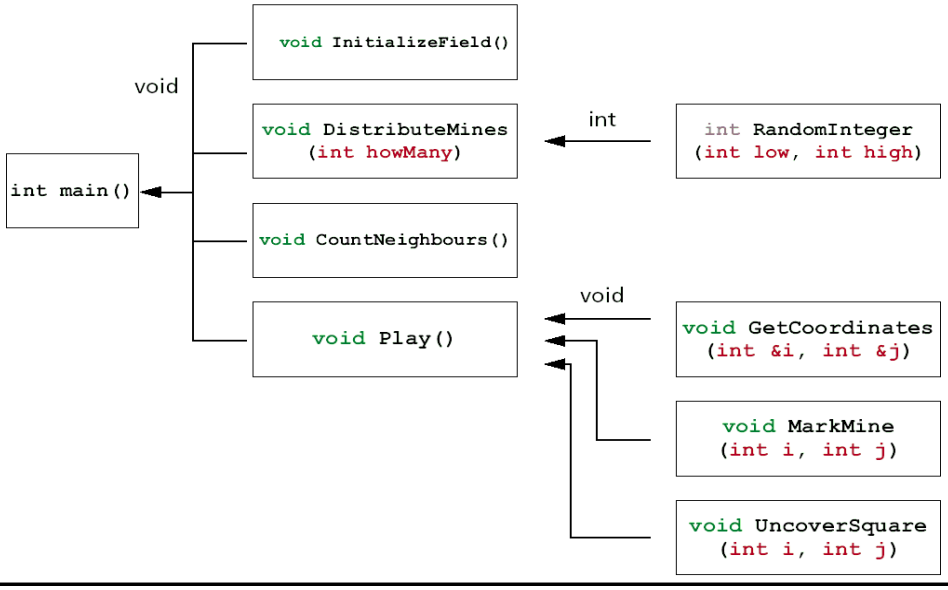
KU LEUVEN

Example: minesweeper



KU LEUVEN

Example: minesweeper



KU LEUVEN

syntax

- General function

return-type function-name(*type*$_1$ *par*$_1$, *type*$_2$ *par*$_2$,…) {
    statements}

- Function with no parameters

return-type function-name(void){
    statements}

Note: A function declaration with no values (e.g. int test()) is not an empty parameter specification, rather it means that its arguments should not be type-checked. Instead, a function with no arguments is declared using void (e.g., int test(void);)

File: function_nocheck.c

- Function returning no result

void *function-name*(*type*$_1$ *par*$_1$, *type*$_2$ *par*$_2$,…) {
    statements}

KU LEUVEN

Functions

- A **function**, **procedure**, or **subroutine** encapsulates some complex computation as a single operation.
- A **function** is a sequence of operations that can be invoked from other places within the software.
- **Call** a function
  - Pass as **arguments** all the information this function needs,
  - Any effect it has will be reflected in either its **return value** or (in some cases) in changes to values pointed to by the arguments,
  - Inside the function, the arguments are copied into local variables,

Function: Declaration and Definition

At least one of the following is required:

1. a function has been *defined* before it is *invoked*
2. the *types* of function arguments and return value have been *declared* before the function is invoked.

- Files:
  - *print_square_naive.c*
  - *print_square_function.c*
  - *print_square_function_proto.c*
  - *print_square_modular.c*
    - *print_square.h*
    - *print_square_f.c*

```c
/*
print_square_naive.c
print the square of an integer
*/

#include<stdio.h>


int main(void) {
   int number=0;


   number = 5;
   printf("%d squared = %d\n", number, number * number);

   number = 8;
   printf("%d squared = %d\n", number, number * number);

   number = 51;
   printf("%d squared = %d\n", number, number * number);

   number = 85;
   printf("%d squared = %d\n", number, number * number);


return 0;
}
```

```c
/*
print_square_function.c
print the square of an integer
*/

#include<stdio.h>


void print_square(int value) {
   printf("%d squared = %d\n", value, value * value);
   return;
}

int main(void) {

   int number=0;


   number = 5;
   print_square(number);

   number = 8;
   print_square(number);

   number = 51;
   print_square(number);

   number = 85;
   print_square(number);

return 0;
}
```

```c
/*
print_square_function_proto.c
print the square of an integer
*/

#include<stdio.h>


void print_square(int);

int main(void) {
   int number=0;


   number = 5;
   print_square(number);

   number = 8;
   print_square(number);

   number = 51;
   print_square(number);

   number = 85;
   print_square(85);

return 0;
}

void print_square(int value) {
   printf("%d squared = %d\n", value, value * value);
   return;
}
```


13

```c
/*
print_square_modular.c
print the square of an integer

use a modular approach

gcc -g -o print_square print_square_modular.c  print_square_f.c
*/
#include<stdio.h>
#include"print_square.h"


int main(void) {
   int number=0;


   number = 5;
   print_square(number);

   number = 8;
   print_square(number);

   number = 51;
   print_square(number);

   number = 85;
   print_square(85);

return 0;
}
```

```c
//print_square.h
void print_square(int);

```

```c
#include <stdio.h>
// print_square_f.c
void print_square(int value) {
   printf("%d squared = %d\n", value, value * value);
   return;
}

```

```
frankvp@CRD-L-08004:.../Functions$ gcc -g -o print_square print_square_modular.c  print_square_f.c
frankvp@CRD-L-08004:.../Functions$ ./print_square
5 squared = 25
8 squared = 64
51 squared = 2601
85 squared = 7225
frankvp@CRD-L-08004:.../Functions$
```


14

Function Prototypes

- It is always a good practice to declare a prototype for the function before you call it.
- A function declaration (or prototype) specifies the syntax (name and input/output parameters)
- Function prototypes are used by the compiler to perform “type-checking”.

```
void a_procedure(void);
int string_length(char* str);
double point_distance(double, double, double);
```

KU LEUVEN

Function Definition

- A function definition specifies the actual program to be executed when the function is called,
- The first line of the definition is similar to the prototype (except that argument names are mandatory), and the function implementation is enclosed in braces (ie, the function block).

```
int myfunc (double m, char n)
{
      statements
}
```

KU LEUVEN

Function Characteristics

- Called by its name and arguments
  - Unique name
  - different functions -> different names
  - communication between calling routine and function goes by the passing parameters and the *return value*
- Are independent:
  - can be called from every program
  - is a unit on its own

Call-by-Value

- Ways to pass parameters:
  - call_by_value
    a copy of the argument's value is made and passed to the function called.
    function can safely modify argument values without causing side effects
    changes to the copy do not affect the original value
  - call_by_reference
    the called function is allowed to modify the original variable's value
- C: only ***call_by_value***
  - no accidental side effects
  - call_by_reference can be simulated

example

- **demo_swap_by_val.c**

```c
/* demo_swap_by_val.c */
#include <stdio.h>

void swap(int i, int j);

void main()
{
    int a,b;
    a=5;
    b=10;
    printf("a=%d, b=%d  -before swap-\n",a,b);
    swap(a,b);
    printf("a=%d, b=%d  -after swap-\n",a,b);
}
void swap(int i, int j)
{
    int t;
    t = i;
    i = j;
    j = t;
}
```

```
frankvp@CRD-L-08004:.../Functions$ ./demo_swap_by_val
a=5, b=10  -before swap-
a=5, b=10  -after swap-
frankvp@CRD-L-08004:.../Functions$
```


Local Variables

- Variables defined inside a block, including inside a function, are *automatic local* variables: they are
  - automatically "created" each time the function is invoked. Their *extent* is *local.*
  - visible only within the block in which they are defined. Their *scope* is *local* or *block*

```
{
  int a=1, b=2;
  ...
}
```

Returning Function Results

- A function can return ***only one*** result to the calling function.
- The syntax is

  `return` *`expression;`*

- The *type* returned must be specified in the function *declaration* and the function *definition*

```
int factorial (int n)
{
  int result = 1;

  while (n)
        result *= n--;
  return result;
}

int main(void)
{
  int a = 10;
  int b = factorial (a);
  printf("%d", b);
  return 0;
}
```


Multiple Return Values

- There are ways to return more than one value from a function using either
  1. Returning a compound type (ie, a **`struct`**), or
  2. Passing arguments using “pass-by-reference” semantics (ie, pointers).

Multiple Return Statements

- Possible to have several **`return`** statements in a function. These define multiple exit-points.
- Thus, a function may only return *one value*, but can return from any number of *places*.

```
int isleapyear(int year)
{
  if (!(year % 400)) return 1;
  if (!(year % 100)) return 0;
  if (!(year % 4)) return 1;
  return 0;
}
```

- If the function returns **`void`**, then an empty **`return;`** may be used.

```
void func(int x)
{
  if (x==0) return;
  other statements
  return; /* optional */
}
```

KU LEUVEN


```
/*
leapyear.c
A leap year must be divisible by 4 but not by 100, except
for years divisible by 40, they are also leap years
*/

#include <stdio.h>

int leapyear(int year);

int main()
{
  int res, year;

  while (1){
  printf("\n Enter a year to check if it is a leap year\n");
  scanf("%d", &year);

  res = leapyear(year);
  if (res == 0) {
     printf(" year %d is not a leap year \n", year);
    }
    else {
     printf(" year %d is a leap year \n", year);
    }
   }

return 0;
}

int leapyear(int year) {
// performs the tests ne by one
  if ( year%400 == 0) {
     return 1;
    }
  if (year%100 == 0) {
     return 0;
    }

  if (year%4 == 0) {
     return 1;
    }

  return 0;
}
```

```
frankvp@CRD-L-08004:.../Functions$ gcc leapyear.c -o leapyear
frankvp@CRD-L-08004:.../Functions$ ./leapyear

 Enter a year to check if it is a leap year
2000
 year 2000 is a leap year

 Enter a year to check if it is a leap year
2020
 year 2020 is a leap year

 Enter a year to check if it is a leap year
2100
 year 2100 is not a leap year

 Enter a year to check if it is a leap year
2021
 year 2021 is not a leap year

 Enter a year to check if it is a leap year
^C
frankvp@CRD-L-08004:.../Functions$
```

KU LEUVEN

Recursion

- C functions may be called recursively
- Programming property:
  - A “base case” – a condition which does NOT make a recursive call because a simple solution exists, this should end the recursive calling.
  - A recursive call with a condition (usually a parameter value) that is CLOSER to the base case than the current condition.
- Each invocation of the function gets its own set of arguments and local variables

Recursive Functions

- Recursive functions are usually less efficient than functions using iterative algorithms (ie, loops).
- Generally, more memory consumption
- But may be more concise or elegant in some cases.

```
int factorial(int n)
{
  if (n==0) return 1;
  return n * factorial(n-1);
}
```

Example

fibonacci.c

fibonacci_seq.c

```c
/*
fibonacci.c
calculate the nth fibonacci number
 fibonacci - recursive
 Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
where:
Fibonacci(0) = 0
Fibonacci(1) = 1

based on http://gribblelab.org/cbootcamp/5_Functions.html

*/


#include <stdio.h>

int Fibonacci(int n);

int main() {
        int n = 11;

        int Fn = Fibonacci(n);

        printf (" enter an integer (larger than 2, but not too large):");
        scanf("%d", &n);
        printf (" %d th fibonacci number = %d \n",n, Fibonacci(n));

    return 0;
}


int Fibonacci(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return Fibonacci(n-1) + Fibonacci(n-2);
}
```

```
frankvp@CRD-L-08004:.../Functions$ gcc fibonacci.c -o fibonacci
frankvp@CRD-L-08004:.../Functions$ ./fibonacci
 enter an integer (larger than 2, but not too large):6
 6 th fibonacci number = 8
frankvp@CRD-L-08004:.../Functions$ ./fibonacci
 enter an integer (larger than 2, but not too large):10
 10 th fibonacci number = 55
frankvp@CRD-L-08004:.../Functions$
```

KU LEUVEN

Function Benefits

- Functions split a program into a set of sub-problems, which in turn may be split.
  - Divide-and-conquer.
- Functions can wrap up difficult algorithms
  - Hide implementations behind simple interfaces
  - Function names provide a higher-level abstraction.
- Functions avoid code duplication.
  - “Code duplication is an error”.

KU LEUVEN

Function Commenting

- Function users need to know
  - What a function does
  - What are its inputs
  - What are its outputs
  - Requirements or conditions on inputs/outputs
    - Eg., input array must contain all zeros
  - Other relevant details
    - Eg., reference to algorithm literature
- File: *commenting.c*

Error Return Values

- Function return value might be purely for status:

```
int func(arguments)
{
  statements
  if (success) return 0;
  return 1;
}
```

- Or might have a normal return, and a special error value:

```
int func(arguments)
{
  int val;
  statements
  if (error) return -1;
  return val; /* normal values are >= 0 */
}
```

Standard Library Functions

- Two types of functions: library functions and functions written by you (or downloaded from the web, etc).
- Library functions are written by the compiler manufacturer.
- The standard library is a valuable resource:
  - The routines are portable and correct.
  - They are an excellent example of good interface design.

KU LEUVEN

Library Functions

- For example: **`printf(),cos(), strlen(), isalpha(), isdigit(), atoi()`**

- Implementations of the standard functions are provided by the compiler manufacturer.
- The function interface is standard, but the implementation is up to the compiler manufacturer

KU LEUVEN

Library Functions

- Library functions provide for common operations:
  - input and output `stdio.h`
  - character type tests `ctype.h`
  - string functions `string.h`
  - maths functions `math.h`
  - standard definitions `stddef.h`
  - sizes of types `limits.h, float.h`
  - string conversion `stdlib.h`
  - memory management `stdlib.h`
  - utility functions `stdlib.h`

  http://en.wikipedia.org/wiki/C_standard_library