

# Programmieren 1

## Compound Data and Variant Data

# Lectures

#	Date	Topic	HÜ→	HÜ←
1	14.10.	Organization, computers, programming, algorithms, PostFix introduction (execution model, IDE, basic operators, booleans, naming)	1	20.10. 23:59
2	21.10.	PostFix (primitive types, functions, parameters, local variables, tests), recipe for atomic data	2	27.10. 23:59
3	28.10.	PostFix (operators, array operations, string operations), recipes for enumerations, intervals, and itemizations	3	3.11. 23:59
4	4.11.	Recipes for compound and variant data, iteration and recursion, PostFix (loops, association arrays, data definitions)	4	10.11. 23:59
5	11.11.	C introduction (if, variables, functions, loops), Programming I C library	5	17.11. 23:59
6	18.11.	Data types, infix expressions, C language (enum, switch)	6	24.11. 23:59
7	25.11.	Compound and variant data, C language (formatted output, struct, union)	7	1.12. 23:59
8	2.12.	C language (arrays, pointers) arrays: fixed-size collections, linear and binary search	8	8.12. 23:59
9	9.12.	Dynamic memory (malloc, free), recursion (recursive data, recursive algorithms)	9	15.12. 23:59
10	16.12.	Linked lists, binary trees, search trees	10	22.12. 23:59
11	13.1.	C language (program structure, scope, lifetime, linkage), function pointers, pointer lists	11	12.1. 23:59
12	20.1.	List and tree operations (filter, map, reduce), objects, object lists	12	19.1. 23:59
13	27.1.	Dynamic data structures (stacks, queues, maps, sets), iterators, documentation tools	(13)	

# Review

- Binary, octal, hexadecimal, and decimal numbers
  - Place-value notation
- Data types and sizes
  - How to interpret bit patterns, value ranges, floating-point representation
- Operators, precedence, and associativity
  - Infix expressions require definition of operator precedence
- Type conversions
  - Extend / restrict representation
- Expressions, evaluation diagrams
  - Stepwise evaluation of expressions
- Recipe for enumerations
- Recipe for intervals

## Review: Wert von x?

```
double x = 11 / 10;
```

## Review: Wert von x?

```
double x = 11.0 / 10;
```

## Review: Wert von x?

```
double x = (double) 11 / 10;
```

```
double x = ((double) 11) / 10;
```

## Review: Mit Bit-Operatoren drittes Bit auf 1 setzen (alle anderen Bits auf 0)

```
int m = 3;  
int x = 1 << m; // shift left 1 by m binary places  
printf("%08x\n", x); // output: 0000000816 = 10002
```

## Review: Mit Bit-Operatoren die niederwertigsten drei Bits auf 1 setzen (alle anderen Bits auf 0)

```
int m = 3;
int x = (1 << m) - 1; // shift left 1 by m places, then subtract 1
printf("%08x\n", x); // output: 0000000716 = 1112
```



# Preview

- Formatted Output
- Assertions, Preconditions, Postconditions
- Structures and Unions
- Recipe for Compound Data (Product Types)
- Recipe for Variant Data (Sum Types)
- Makefiles (optional topic)

# Code Formatting

- Format your source code consistently
  - Empty lines for structuring
  - At most 80 characters per line
  - Just one statement per line
  - {...} block indentation is 4 spaces (not tabs)
  - Lower case variable and function names, separated\_by\_underscore (orCamelCase)
  - CAPITALIZED\_WITH\_UNDERSCORES for constants
  - CamelCase with an upper-case first letter for type names
  - one space before and after binary operators (1 + 2, not 1+2)
- Also see recommendations in script
  - There are different sensible styles
  - Be consistent, maximize readability

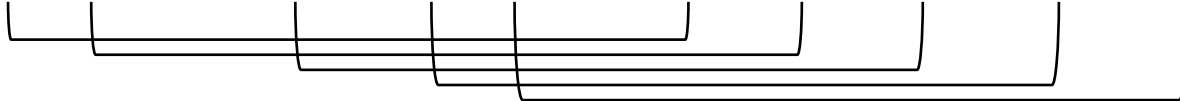
# FORMATTED OUTPUT

# Producing Formatted Output

- Produce output consisting of fixed and variable parts
  - Example: "3 kg times 1.5 is 4.5 kg." (variable parts underlined)
- Tedious with print(i|d|s)[ln]
 

```
int base = 3;
double factor = 1.5;
double total = factor * base;
String unit = "kg";
printi(base); prints(" "); prints(unit);
prints(" times "); printd(factor); prints(" is ");
printd(total); prints(" "); prints(unit); println(".");
```
- Simple with formatted output (printf)
 

```
printf("%d %s times %g is %g %s.\n", base, unit, factor, total, unit);
```



# Characters


- Individual characters are written in single quotes (e.g., 'A')
  - Compiler translates them to their ASCII code (e.g., 65 for 'A')
- Individual characters stored in variables of type char
  - `char c = 'A';` // char type: typically 8 bits
- Thus, can compute with characters
  - `'A' < 'B'` is translated to `65 < 66`, which is true
  - `'A' + 1` is translated to `65 + 1 = 66` (ASCII code of 'B')
- Some characters have a special meaning
  - `'\n'` means "start a new line" and is stored as char 10
- Strings are written in double quotes (e.g., "A" or "ABC")
  - "A" (String, stored as 65, 0) is different from 'A' (char, stored as 65)!

# Control Characters and Escape Sequences

- String literal
  - Example "hello world"
- Inserting special control codes in string literals
  - Example `"\hello\world\n"` \ is called backslash
- \ starts escape sequence, next character is interpreted differently
  - \n – new line (line break)
  - \r – carriage return
  - \t – tabulator
  - \" – double quotation mark
  - \\ – backslash

Ending a line of text:  
Windows: \r\n  
Linux, macOS: \n
- At runtime an escape sequence is a single byte (char)
  - ASCII codes: \n → 10, \r → 13, \t → 9, \" → 34, \\ → 92

# Formatted Output with printf

- printf is not part of C language
- printf is a function in the standard library
  - [www.gnu.org/software/libc](http://www.gnu.org/software/libc)
- Format string contains placeholders (%d)  
 printf("%d\t%d\n", fahrenheit, celsius);
 
- ... and formatting descriptions (%3d)  
 "%3d %6d\n" means  
 %3d: right-justified decimal, 3 characters  
 %6d: right-justified decimal, 6 characters

0	-17
20	-6
40	4
60	15
80	26
100	37

0	-17
20	-6
40	4
60	15
80	26
100	37

123	123456
%3d	%6d

# Formatted Output with printf

```
double fahrenheit = 100;
double celsius = 37.8;
printf("%3.0f %6.1f\n", fahrenheit, celsius);
```

Format string means

**%3.0f**: right-justified float of width 3,  
0 fraction digits

**%6.1f**: right-justified float of width 6,  
1 fraction digit

0	-17.8
20	-6.7
40	4.4
60	15.6
80	26.7
100	37.8
120	48.9
140	60.0
160	71.1
180	82.2
200	93.3
220	104.4
240	115.6
260	126.7
280	137.8
300	148.9
123	123456
%3.0f	%6.1f



# String Alignment

```
String a = "abcde";
printf("|%s|\n", a);
printf("|%10s|\n", a);
printf("|%-10s|\n", a);
printf("|%10.3s|\n", a);
printf("|%-10.3s|\n", a);
```

```
|abcde|
|      abcde|
|abcde      |
|      abc|
|abc        |
```

# Format String Overview

- %d decimal integer
- %6d decimal integer, at least 6 characters wide, right-justified
- %f floating-point number (or %g)
- %.2f float, 2 characters after decimal point
- %6.2f float, at least 6 characters wide, of which 2 are after the decimal point
- %e ["-"] d "." ddddddd "e" ("+"|" -") xx
- %x hexadecimal integer (base 16)
- %#x hexadecimal integer (base 16), preceded by "0x"
- %c character
- %s character string
- %8.4s print 8 characters wide, first 4 characters of string, left-pad with space
- %% the % itself

# ASSERTIONS, PRECONDITIONS, POSTCONDITIONS

# "Partial" Functions

- Some functions have a limited domain
  - Example: sqrt does not work with negative values, sqrt(-3) returns NAN, error may go unnoticed
  - Precondition of sqrt is:  $x \geq 0$
- Explicitly check for allowed argument values
 

```
double my_sqrt(double x) {
    if (x < 0) {
        printf("Cannot compute the square root of a negative number.");
        exit(1); // terminate the program
    }
    return sqrt(x);
}
```
- Would be helpful to include line number and function name locate problem
 

```
printf("line %d: %s's precondition (x >= 0) violated\n", __LINE__, __func__);
```

# Specify What is Required of the Argument Value

- What is required of x?

```
double my_sqrt(double x) {
    require("non-negative argument", x >= 0);
    return sqrt(x);
}
```

- Rather than: if (invalid) {fail}

```
double my_sqrt(double x) {
    if (x < 0) {
        println("Cannot compute the square root of a negative number.");
        exit(1); // terminate the program
    }
    return sqrt(x);
}
```

# Implement Precondition Check as a Macro

Macro with arguments:

```
#define my_require(description, condition) \
    if (!(condition)) { \
        printf("line %d: %s's precondition \"%s\" (%s) violated\n", \
            __LINE__, __func__, description, #condition); \
        exit(1); \
    }
```

quote line break  
(continue line)

treat argument like a  
string: "x >= 0"

# Preprocessor Definitions

```
// preproc.c
```

```
int puts(const char * str);
```

```
#define X 123
```

```
int main(void) {
    if (X > 0) puts("positive!");
    return 0;
}
```

Only run preprocessor: gcc -E preproc.c

```
int puts(const char * str);
```

```
int main(void) {
    if (123 > 0) puts("positive!");
    return 0;
}
```

# Preprocessor Macros

```
// preproc.c
```

```
int puts(const char * str);
```

```
#define X 123
#define cond_msg(cond, msg) \
    if (cond) { puts(msg); }
```

```
int main(void) {
    cond_msg(X > 0, "positive!");
    return 0;
}
```

Only run preprocessor: gcc -E preproc.c

```
int puts(const char * str);
```

```
int main(void) {
    if (123 > 0) { puts("positive!"); };
    return 0;
}
```



# Preconditions Check Arguments, Postconditions Check Result

- Preconditions protect function against bugs in caller
- Postconditions protect function against bugs in function implementation

- Example

```
double my_sqrt(double x) {
    require("non-negative argument", x >= 0); // precondition check
    double r = sqrt(x);
    ensure("non-negative result", r >= 0); // postcondition check
    return r;
}
```

- Output for failed precondition:
  - line 52: my\_sqrt's precondition "non-negative argument" ( $x \geq 0$ ) violated
- Output for failed postcondition:
  - line 54: my\_sqrt's postcondition "non-negative result" ( $r \geq 0$ ) violated

# Precise Postconditions

- Weak postcondition:

```
double my_sqrt(double x) {
    require("non-negative argument", x >= 0); // precondition check
    double r = sqrt(x);
    ensure("non-negative result", r >= 0); // postcondition check
    return r;
}
```

- More specific postcondition:

```
double my_sqrt(double x) {
    require("non-negative argument", x >= 0);
    double r = sqrt(x);
    ensure("correct result", fabs(r * r - x) < 1e-10);
    return r;
}
```

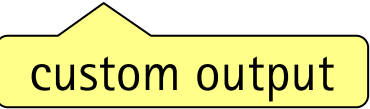
# Custom Failure Output

- Example

```
double my_sqrt(double x) {
    require_x("non-negative argument", x >= 0, "x = %g", x);
    double r = sqrt(x);
    ensure_x("non-negative result", r >= 0, "r = %g", r);
    return r;
}
```

- Output for failed precondition:

- line 52: my\_sqrt's precondition "non-negative argument" violated: x = -4



custom output

# Assertions

- Preconditions and postconditions are special cases of assertions
- Preconditions at the beginning of a function
- Postconditions before returning from a function
- Assertions may appear anywhere in a function
- Assertions check conditions that must be true at that point
- Example: `assert("within bounds", a >= 0 && a < 10);`
- Failure output: `line 57: assertion "within bounds" (a >= 0 && a < 10) violated`

assertion failure  
indicates a bug

# Referring to Old Value, Disabling Checking

- Sometimes helpful to refer to initial value of argument

```
double inc(int i) {
    ensure_code(int old_i = i);
    i = i + 1;
    ensure("incremented", i == old_i + 1);
    return i;
}
```

- Final code should not contain these checks for performance reasons

```
double inc(int i) {
    i = i + 1;
    return i;
}
```

- Checks can be removed by defining NO\_ENSURE, NO\_REQUIRE, NO\_ASSERT:

```
#define NO_ENSURE
#include "base.h"
```

# Preprocessor: Conditional Definition of a Macro

```
int puts(const char * str);
```

```
#define X 123
```

```
#define USE_MESSAGE
```

```
#ifdef USE_MESSAGE
```

```
    #define cond_msg(cond, msg) if (cond) { puts(msg); }
```

```
#else
```

```
    #define cond_msg(cond, msg)
```

```
#endif
```

```
int main(void) {
    cond_msg(X > 0, "positive!");
    return 0;
}
```

Run preprocessor: gcc -E preproc.c

```
int puts(const char * str);
int main(void) {
    if (123 > 0) puts("positive!");
    return 0;
}
```

# Preprocessor: Conditional Definition of a Macro

```
int puts(const char * str);
```

```
#define X 123
```

```
// #define USE_MESSAGE
```

```
#ifndef USE_MESSAGE
```

```
    #define cond_msg(cond, msg) if (cond) { puts(msg); }
```

```
#else
```

```
    #define cond_msg(cond, msg)
```

```
#endif
```

```
int main(void) {
    cond_msg(X > 0, "positive!");
    return 0;
}
```

Run preprocessor: gcc -E preproc.c

```
int puts(const char * str);
int main(void) {
    ;
    return 0;
}
```

# STRUCTURES



# Structures

- Treats a fixed set of related variables as a unit
- Keyword **struct** followed by a name and a list of declarations in braces defines a new type
- Example: Define type Point consisting of two integers

```
struct Point {
    int x;
    int y;
};
```

structure name: Point

members: x and y

needs a semicolon

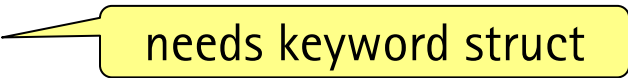
## PostFix:

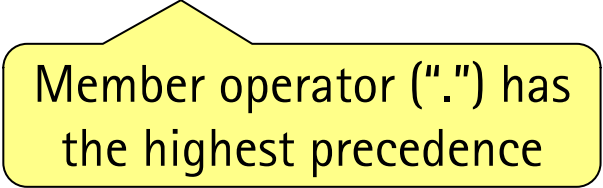
Point: (x :Int, y :Int) data def

- Structures can be nested

```
struct Line {
    struct Point pt1;
    struct Point pt2;
};
```

# Declaring Structure Variables

- A struct declaration defines a type, but does not reserve storage
- Declaring a variable of type struct Point
  - `struct Point pt;` 
- Initializing a structure (with constant expressions)
  - `struct Point pt = { 320, 200 };`
- Accessing members with the member (".") operator
  - `printf("%d, %d", pt.x, pt.y);`



Member operator (".") has  
the highest precedence

# Operations on Structures

- Structures are copied when assigned

```
struct Point p1 = { 100, 200 };
```

```
struct Point p2;
```

```
p2 = p1;
```

copies both members of p1 to p2  
(so p2.x == 100, p2.y == 200)

- Structures are copied when used as function arguments

- Copy by value (all members are copied)

- Structures are copied when used as a function return value

```
struct Point add_point(struct Point p1, struct Point p2) {
```

```
    p1.x += p2.x;
```

```
    p1.y += p2.y;
```

```
    return p1;
```

```
}
```

add\_point operates on its own  
copies of p1 and p2

# Type Definitions with Structures

- typedef creates new type names
- typedef simplifies complex declarations
- typedef for structures allows to omit the struct keyword
- Example typedef for a structure:

```
struct Circle { int x, y, radius; }; // declare struct Circle with members x, y, radius
typedef struct Circle CircleType; // declare new type CircleType
CircleType c; // declare variable of new type, no need for keyword struct
c.x = 100; c.y = 50; c.radius = 30; // use new variable
```

# UNIONS

# Unions

- Similar to structs, but members share memory ("on top of" each other)
  - Tagged unions represent "variants"
  - Only one value at a time, only last member that was set is valid
  - `sizeof(union u) = sizeof(<largest member of union u>)`

- Example:

```
union IntOrFloat { int i; float f; };
printf("sizeof(union IntOrFloat); // output: 4
union IntOrFloat u;
u.f = 0.0f; u.i = 1234567890;
printf("%d, %f\n", u.i, u.f); // output: 1234567890, 1228890.250000
u.i = 0; u.f = 1234567890.0f;
printf("%d, %f\n", u.i, u.f); // output: 1318267910, 1234567936.000000
```

# Tagged Unions

- No type safety
  - Need to explicitly store type information to interpret variant correctly
  - Unions can be anonymous

- Example

```
struct Circle { int x, y, radius; };
struct Rectangle { int x, y, width, height; };
enum ShapeTag { CIRCLE, RECTANGLE };
struct Shape {
    enum ShapeTag tag;
    union { struct Circle circle;
           struct Rectangle rectangle;
    };
};
```

anonymous union  
(union has no name)

## PostFix:

```
Shape: {
    Circle: (x, y, radius)
    Rectangle: (x, y, width, height)
} datadef
```

# Tagged Unions

- No type safety
  - Need to explicitly store type information to interpret variant correctly
  - Unions can be anonymous

- Example

```

struct Circle { int x, y, radius; }; // 3 * 4 bytes
struct Rectangle { int x, y, width, height; }; // 4 * 4 bytes
enum ShapeTag { CIRCLE, RECTANGLE }; // 4 bytes
struct Shape { // 5 * 4 bytes: sizeof ShapeTag + max(sizeof Circle, sizeof Rect.)
    enum ShapeTag tag;
    union { struct Circle circle;
           struct Rectangle rectangle;
    };
};
  
```

anonymous union  
(union has no name)



# Shape: Memory Layout

```
struct Shape {
    enum ShapeTag tag;
    union {
        struct Circle circle;
        struct Rectangle rectangle;
    };
};
```

Memory representation of CIRCLE variant:

4 bytes	4 bytes	4 bytes	4 bytes	4 bytes
CIRCLE tag	x	y	radius	not used

Memory representation of RECTANGLE variant:

4 bytes	4 bytes	4 bytes	4 bytes	4 bytes
RECTANGLE tag	x	y	width	height

# Using Tagged Unions

```
struct Shape {
    enum ShapeTag tag;
    union {
        struct Circle circle;
        struct Rectangle rectangle;
    };
};
```

```
struct Shape s;
s.tag = CIRCLE;
s.circle.x = 100;
s.circle.y = 50;
s.circle.radius = 50;
```

```
switch (s.tag) {
case CIRCLE:
    printf("circle: %d %d %d\n",
           s.circle.x, s.circle.y,
           s.circle.radius);
    break;
case RECTANGLE:
    printf("rectangle: %d %d %d %d\n",
           s.rectangle.x, s.rectangle.y,
           s.rectangle.width,
           s.rectangle.height);
    break;
}
```

# Constructor Functions for Tagged Unions

```
Shape make_circle(int x, int y, int r) {
    require("non-negative radius", r >= 0);
    Shape s;
    s.tag = CIRCLE;
    s.circle.x = x;
    s.circle.y = y;
    s.circle.radius = r;
    return s;
}
```

```
Shape c = make_circle(10, 50, 20);
print_shape(c);
```

```
Shape r = make_rectangle(10, 20, 30, 40);
print_shape(r);
```

```
Shape make_rectangle(int x, int y, int w, int h) {
    require("non-negative width", w >= 0);
    require("non-negative height", h >= 0);
    Shape s;
    s.tag = RECTANGLE;
    s.rectangle.x = x;
    s.rectangle.y = y;
    s.rectangle.width = w;
    s.rectangle.height = h;
    return s;
}
```

# Tagged Unions

Shorter syntax with anonymous enum, union, structs

```
typedef struct {
    enum { CIRCLE, RECTANGLE } tag;
    union {
        struct { int x, y, radius; } circle;
        struct { int x, y, width, height; } rectangle;
    };
} Shape;
```

# RECIPE FOR COMPOUND DATA (PRODUCT TYPES)

# Compound Data (Product Types)

- Aggregate a fixed number of (potentially) different kinds of data into a whole
  - Product types: Cartesian product of components
  - Components may be atomic or structured
- Example
  - 2D point with x- and y-coordinates as components
- Data definition
  - C structures (keyword: struct) represent values that consist of a fixed number of components

# 1. Problem Statement

- Write down the problem statement as a comment.
  - What is the relevant information?
  - What should the function do with the data?

- Example

/\*

Objects are located somewhere on a 2D plane. Design a function that computes the distance of the center of an object to the origin of the coordinate system.

\*/

## 2. Data Definition

- How should domain information be represented as data in the program?  
How to interpret the data as real-world information?
- Data definition

```
struct Point {
    double x; // represents the x-coordinate of a point
    double y; // represents the y-coordinate of a point
};
```

Forgetting the ; is a syntax error (this is a declaration statement)

```
// constructor function for Point
struct Point make_point(double x, double y) {
    struct Point p = { x, y };
    return p;
}
```

No need for a ; (this is a block)



## 3. Function Signature

- Function signature as a comment
  - Parameter types left of the arrow (comma separated)
  - Result type right of the arrow
- Example
  - `// struct Point -> double`

## 4. Function Name

- Preliminary function name
  - Short, non-abbreviated, descriptive name that describes what the function does
- Example  
`distance_to_origin`

## 5. Function Header

- Add function signature to name
- Preliminary parameter names
  - Short, non-abbreviated, descriptive name that describes what the parameter means
- Example  
`double distance_to_origin(struct Point p);`

## 6. Function Stub

- Function stub returns an arbitrary value from the function's range
- The function stub compiles
- Example

```
double distance_to_origin(struct Point p) {  
    return 0.0;  
}
```

## 7. Purpose Statement

- Briefly describes what the function does. Ideally as a single sentence. Multiple short sentences may be necessary.
- Example
  - // Computes the distance from the given point
  - // to the origin of the coordinate system.

## 8. Examples and Expected Results

- For point (0,0) expect a distance to origin of 0.
- For point (1,0) expect a distance to origin of 1.
- For point (-1,0) expect a distance to origin of 1.
- For point (3,4) expect a distance to origin of 5.
- etc.

## 8. Examples and Expected Results (Test Function)

```
static void distance_to_origin_test(void) {
    test_within_d(distance_to_origin(make_point(0, 0)), 0.0, EPSILON);
    test_within_d(distance_to_origin(make_point(1, 0)), 1.0, EPSILON);
    test_within_d(distance_to_origin(make_point(-1, 0)), 1.0, EPSILON);
    test_within_d(distance_to_origin(make_point(0, 2)), 2.0, EPSILON);
    test_within_d(distance_to_origin(make_point(0, -2)), 2.0, EPSILON);
    test_within_d(distance_to_origin(make_point(3, 4)), 5.0, EPSILON);
    test_within_d(distance_to_origin(make_point(3, -4)), 5.0, EPSILON);
}
```

Preprocessor will replace  
EPSILON by 0.00000001  
(defined in base.h)

## 9. Function Body

- Implementation of the function
- Example

// Computes the distance from the given point  
// to the origin of the coordinate system.

```
double distance_to_origin(struct Point p) {
    return sqrt(p.x * p.x + p.y * p.y); // square root in math.h
}
```

dot (.) operator to access  
components of struct

on Linux need to add  
math library: gcc ... -lm



## 10. Testing

- Main function call test function

```
int main(void) {  
    distance_to_origin_test();  
    return 0;  
}
```

- Test results

```
point.c, line 30: check passed  
point.c, line 32: check passed  
point.c, line 34: check passed  
point.c, line 36: check passed  
point.c, line 38: check passed  
point.c, line 40: check passed  
point.c, line 42: check passed  
All 7 tests passed!
```

# RECIPE FOR VARIANT DATA (SUM TYPES)



# 1. Problem Statement

- Write down the problem statement as a comment.
  - What is the relevant information?
  - What should the function do with the data?

- Example

/\*

Points on the 2D plane may be given either in Euclidean coordinates or in polar coordinates. Design a function that computes the distance of such a point to the origin of the coordinate system.

\*/

## 2. Data Definition

- How should domain information be represented as data in the program?  
How to interpret the data as real-world information?
- Data definition

```
enum PointTag {
    TPointEuclid, // tag for a point in Euclidean coordinates
    TPointPolar   // tag for a point in polar coordinates
};

struct Point {
    enum PointTag tag; // indicates which variant follows
    union {
        struct { double x; double y; }; // Euclidean variant
        struct { double theta; double magnitude; }; // polar variant
    };
};
```

## 2. Data Definition

- How should domain information be represented as data in the program? How to interpret the data as real-world information?
- Constructor functions

```
struct Point make_point_euclid(double x, double y) {
    struct Point p;
    p.tag = TPointEuclid;
    p.x = x;
    p.y = y;
    return p;
}
```

```
struct Point make_point_polar(double t, double m) {
    require("valid angle", 0 <= t && t < 360);
    require("non-negative magnitude", m >= 0);
    struct Point p;
    p.tag = TPointPolar;
    p.theta = t; p.magnitude = m;
    return p;
}
```

## 2. Data Definition (unnamed vs. named union members)

- Union with unnamed members

```
struct Point {
    enum PointTag tag;
    union {
        struct { double x; double y; };
        struct { double theta; double magnitude; };
    };
};
```

### Member access:

```
struct Point p;
p.tag = TPointEuclid;
p.x = 12;
p.y = 34;
```

- Union with named members

```
struct Point {
    enum PointTag tag;
    union {
        struct { double x; double y; } euclid;
        struct { double theta; double magnitude; } polar;
    };
};
```

### Member access:

```
struct Point p;
p.tag = TPointEuclid;
p.euclid.x = 12;
p.euclid.y = 34;
```

## 2. Data Definition (alternative: separate structs)

- How should domain information be represented as data in the program? How to interpret the data as real-world information?
- Data definition

```
struct EuclideanPoint {
    double x;
    double y;
};

struct PolarPoint {
    double theta;
    double magnitude;
};
```

```
struct Point {
    enum PointTag tag; // which variant
    union {
        // Euclidean variant
        struct EuclideanPoint euclid;
        // polar variant
        struct PolarPoint polar;
    };
};
```



## 2. Data Definition (alternative: separate structs)

- How should domain information be represented as data in the program? How to interpret the data as real-world information?
- Constructor functions

```
struct Point make_point_euclid(double x, double y) {
    struct Point p;
    p.tag = TPointEuclid;
    p.euclid.x = x;
    p.euclid.y = y;
    return p;
}
```

```
struct Point make_point_polar(double t, double m) {
    require("valid angle", 0 <= t && t < 360);
    require("non-negative magnitude", m >= 0);
    struct Point p;
    p.tag = TPointPolar;
    p.polar.theta = t;
    p.polar.magnitude = m;
    return p;
}
```

### 3. Function Signature

- Function signature as a comment
  - Parameter types left of the arrow (comma separated)
  - Result type right of the arrow
- Example  
`// struct Point -> double`

## 4. Function Name

- Preliminary function name
  - Short, non-abbreviated, descriptive name that describes what the function does
- Example  
`distance_to_origin`

## 5. Function Header

- Add function signature to name
- Preliminary parameter names
  - Short, non-abbreviated, descriptive name that describes what the parameter means
- Example  
`double distance_to_origin(struct Point p);`

## 6. Function Stub

- Function stub returns an arbitrary value from the function's range
- The function stub compiles
- Example

```
double distance_to_origin(struct Point p) {  
    return 0.0;  
}
```

## 7. Purpose Statement

- Briefly describes what the function does. Ideally as a single sentence. Multiple short sentences may be necessary.
- Example
  - // Computes the distance from the given point
  - // to the origin of the coordinate system.

## 8. Examples and Expected Results

- For point (0,0) expect a distance to origin of 0.
- For point (1,0) expect a distance to origin of 1.
- For point (-1,0) expect a distance to origin of 1.
- For point (3,4) expect a distance to origin of 5.
- etc.

## 8. Examples and Expected Results (Test Function)

```
static void distance_to_origin_test(void) {
    // test cases for polar variant
    test_within_d(
        distance_to_origin(make_point_polar(0.0, 0.0)), 0.0, EPSILON);
    test_within_d(
        distance_to_origin(make_point_polar(0.0, 1.0)), 1.0, EPSILON);
    test_within_d(
        distance_to_origin(make_point_polar(22.5, 2.0)), 2.0, EPSILON);
    // test cases for Euclidean variant
    test_within_d(
        distance_to_origin(make_point_euclid(0.0, -2.0)), 2.0, EPSILON);
    test_within_d(
        distance_to_origin(make_point_euclid(2.0, 0.0)), 2.0, EPSILON);
    test_within_d(
        distance_to_origin(make_point_euclid(1.0, 1.0)), sqrt(2.0), EPSILON);
}
```

Preprocessor will replace  
EPSILON by 0.00000001



## 9. Function Body

- Implementation of the function
- Example

// Computes the distance from the given point to the origin.

```
double distance_to_origin(struct Point p) {
    if (p.tag == TPointEuclid) {
        return sqrt(p.x * p.x + p.y * p.y); // square root
    } else if (p.tag == TPointPolar) {
        return p.magnitude;
    }
    return 0.0;
}
```

dot (.) operator to access  
components of struct

## 10. Testing

- Main function call test function

```
int main(void) {  
    distance_to_origin_test();  
    return 0;  
}
```

- Test results

```
point_euclid_polar.c, line 58: check passed  
point_euclid_polar.c, line 61: check passed  
point_euclid_polar.c, line 64: check passed  
point_euclid_polar.c, line 69: check passed  
point_euclid_polar.c, line 72: check passed  
point_euclid_polar.c, line 75: check passed  
All 6 tests passed!
```

## 11. Review and Revise

- Review the products of the steps
  - Improve function name
  - Improve parameter names
  - Improve purpose statement
  - Improve and extend tests
- Improve / generalize the function
  - Switch statement
  - Typedef
  - Templates for future functions on this data

# 11. Review and Revise: Typedef and Switch Statement

```
typedef struct Point Point;
```

```
// Computes the distance from the given point to the origin.
```

```
double distance_to_origin(Point p) {
    switch (p.tag) {
        case TPointEuclid:
            return sqrt(p.x * p.x + p.y * p.y); // square root
        case TPointPolar:
            return p.magnitude;
    }
    return 0;
}
```

because of **typedef** can write **Point p** rather than having to write **struct Point p**

# Reusable Template for Variant-Point Functions

```
... fn_for_points(struct Point p) {  
    switch (p.tag) {  
        case TPointEuclid:  
            ...  
            return... p.x... p.y...;  
        case TPointPolar:  
            ...  
            return... p.theta... p.magnitude...;  
        }  
    return ...;  
}
```

# Check Functions for Basic Data Types

- Predefined check functions only for basic data types
  - Example call: `test_equal_i(2 * 5, 10);` // on line 12 of hello.c
  - Example output: hello.c, line 12: check passed
- File name (hello.c) and line number (12) are set automatically
- Use `base_test_equal_...` to set file name and line number manually
  - `base_test_equal_i("hello.c", 12, 2 * 5, 10);` // file, line, actual, expected
- May use `__FILE__` and `__LINE__` macros:
  - `base_test_equal_i(__FILE__, __LINE__, 2 * 5, 10);`

preprocessor replaces  
`__FILE__` with name  
of source file

preprocessor replaces  
`__LINE__` with line  
number of this call

# Check Functions for Structures

- Checking structures requires checking each member
- Example: Check function for Euclidean points

```
typedef struct { int x, y; } Point;
```

- Example call of check function

```
test_equal_point(__LINE__, make_point(10, 20), make_point(5 + 5, 2 * 10));
```

- Check function checks x and y members separately

```
bool test_equal_point(int line, Point a, Point b) {
    bool x_equal = base_test_equal_i(__FILE__, line, a.x, b.x);
    bool y_equal = base_test_equal_i(__FILE__, line, a.y, b.y);
    return x_equal && y_equal;
}
```

# MAKEFILES

(OPTIONAL TOPIC)



# Build Process: The “make” Tool

- gcc is both a compiler and a linker
- Basic build process
  1. Compiler produces an object file (.o or .obj) for each source file (.c)
  2. Linker takes the object files and creates an executable (.exe)
- For large projects
  - Difficult to manage build process by hand
  - Time consuming to compile every file each time a .c or .h file is changed
- make[.exe] tool helps to manage the build process
  - make takes care of recompiling/linking only the necessary files
  - Command line: **make target**
    - make uses target (explicitly) and Makefile (implicitly) as input
  - Makefile contains a list of rules for creating the executable

# Build Process: Makefiles

- Makefile contains a list of rules
  - Rules specify what file depends on what other file(s)
- Basic Makefile rule syntax
 

target: dependencies	← (aka. prerequisites)
[tab] command	← one '\t' character at the start of the line
- If a dependency is newer than the target,  
then the target is out of date and must be rebuilt.
- Example rule
 

main.o: main.c main.h	← main.o depends on main.c and main.h
gcc -c -Wall main.c	← if main.o is older than a dependency, then call gcc with the specified arguments

# Build Process: Makefile Variables

- Variables are defined as: VAR = value
  - CC = gcc
  - CFLAGS = -Wall
  - SRCS = main.c foo.c bar.c
  
- Variables are referenced as: \$(VAR)
  - \$(CC) -c \$(CFLAGS) main.c      ← becomes gcc -c -Wall main.c
  
- Substitution references
  - OBJS = \$(SRCS:.c=.o)      ← becomes \$(OBJS) = "main.o foo.o bar.o"

# Build Process: Automatic Makefile Variables

- Automatic (predefined) variables
  - `$@` is the file name of the target of the rule
  - `$<` is the name of the first dependency of the rule
  - `^` is the list of all dependencies of the rule
  - `$?` is the list of dependencies that are newer than the target
  
- Example
 

```
$(EXECUTABLE): $(OBJS)      ← executable depends on the object files
    @echo "$@ is older than these dependencies: $?"
```

If one of the object files is newer than the executable,  
output the name of the target (`$@`) and the list of  
object files that are newer than the target (`$?`)

# Build Process: Pattern Rules

- Rules to produce .o files from .c files are very similar
  - Tedious to repeat them for each .c file
- Pattern rule to produce object files from c files
 

`%o.o: %.c`

`$(CC) -c $<`

← a .o-file depends on a .c-file of the same name

← to create a .o-file from a .c-file

use command `gcc -c <first dependency>`

# Build Process: Automatic Dependencies

- Problem: List of dependencies must be updated as new headers are included in .c-files
- Solution: Let gcc parse .c-file and produce dependency rules automatically (option -MM)
  - Example:
  - `gcc -MM main.c` creates this output:
  - `main.o: main.c foo.h bar.h main.h`
- Redirect output to a file:
  - `gcc -MM $< > $(<:.c=.d)`

produces dependency files  
(e.g., main.d)

first dependency

substitute extension

redirect stdio

# Example Makefile (used with assignment templates)

```
CC = gcc
LINKER = gcc
CFLAGS = -std=c99 -Wall -Wpointer-arith -Werror -Wno-error=unused-variable -Wfatal-errors
DEBUG = -g
LIBNAME = prog1
LIBDIR = ../prog1lib/lib # location of library folder relative to Makefile
# disable default suffixes
.SUFFIXES:
# pattern rule for compiling the library
prog1lib:
    cd $(LIBDIR) && make
# pattern rule for compiling .c-file to executable
%: %.c prog1lib # executable depends on .c-file and prog1lib
    $(CC) $(CFLAGS) $(DEBUG) $< -L$(LIBDIR) -l$(LIBNAME) -iquote$(LIBDIR) -o $@
```

first dependency

target

# Multiple Source Files

CC = gcc

LINKER = gcc

EXECUTABLE = myprogram

SRCS = main.c foo.c bar.c # multiple source files

OBJS = \$(SRCS:.c=.o) # main.o foo.o bar.o...

.SUFFIXES: # disable default suffixes

%o.o: %.c # define pattern rule

\$(CC) -c \$(CFLAGS) \$< # compiling .c to .o

\$(CC) -MM \$< > \$(<:.c=.d) # producing dependencies



# Multiple Source Files

```
$(EXECUTABLE): $(OBJS)
```

```
$(LINKER) -o $(EXECUTABLE) $(OBJS) # linking to .exe
```

```
-include $(OBJS:.o=.d) # include dependency rules
```

```
.PHONY: clean # do not treat "clean" as a file name
```

```
clean: # special target to remove files: make clean
```

```
rm -f $(EXECUTABLE)
```

```
rm -f $(OBJS)
```

```
rm -f $(SRCS:.c=.d)
```

# C Keywords (ANSI C, ISO C89)

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

# Summary

- Formatted Output
  - `printf`, placeholders (`%d`, `%f`) must match variable types
- Assertions, Preconditions, Postconditions
  - Assertions check for conditions that must be true at that point
  - Preconditions constrain function arguments, protect function against bugs in caller
  - Postconditions protect function against bugs in function implementation
- Structures and Unions
  - Compound types (structures)
  - Variant types (tagged unions)
- Recipe for Compound Data (Product Types)
- Recipe for Variant Data (Sum Types)
- Makefiles (optional topic)
  - Manage the build process