Introduction to C

Programming Workshop in C (67316)
Fall 2018
Lecture 10
20.11.2018

Pointers

- pointers:

```
int x;
int *p = &x;
```

- pointers to pointers:

```
int x;
int *p = &x;
int **p = &p;
```

- array of pointers:

```
char* names[] = {“I”, “love”, “pointers”};
```

- multidimensional arrays:

```
int x[20][20];
```

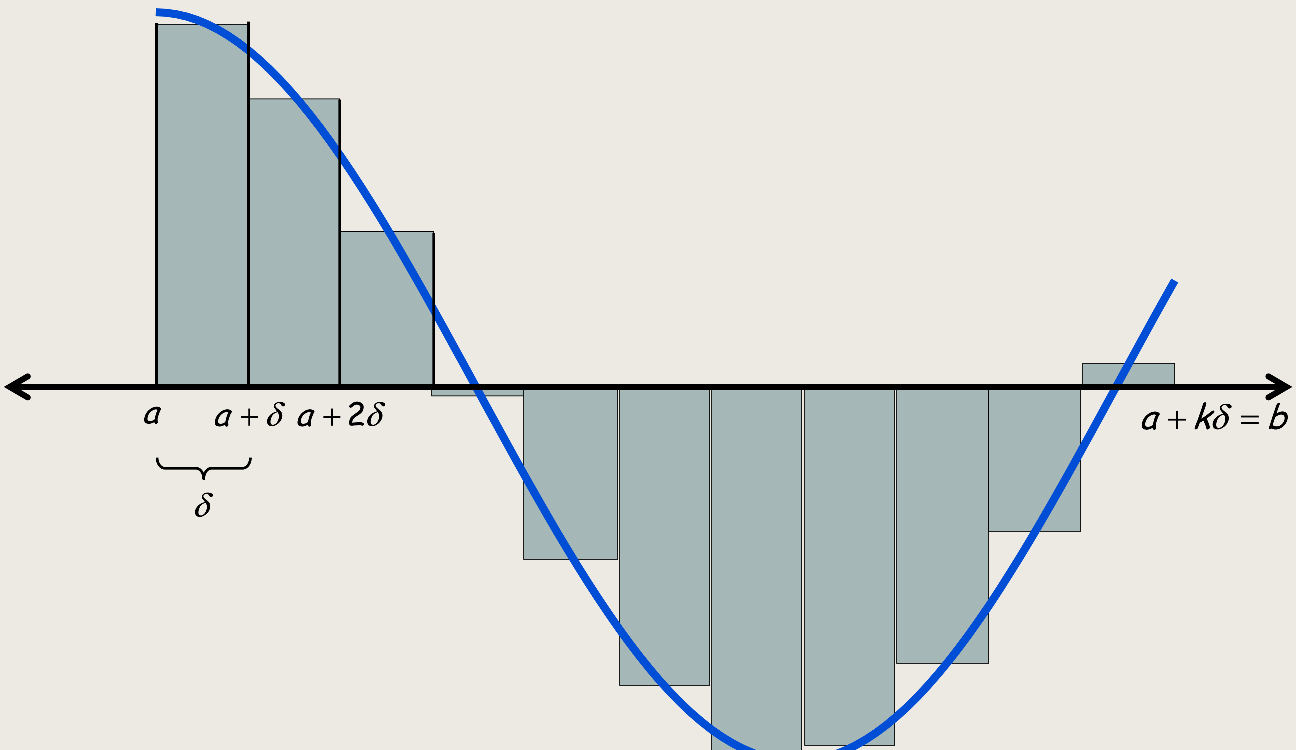
Pointers to functions

Pointers to Functions

- In some programming languages, functions are variables: can be passed to functions, returned from functions
- In C, function is not a variable. However, C allows to declare **a pointer to a function**
- When you would want to pass pointers to functions?

Example: Numerical Integrator

$$\int_a^b f(x)dx \approx \sum_{i=1}^{k} \delta f\left(a + (i - \frac{1}{2})\delta\right) \qquad \delta = \frac{b-a}{k}$$



Pointers to Functions - How to use

```c
void fun(int a) {
    printf("Value of a is %d\n", a);
}

int main()  {
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

Pointers to Functions - How to use

why?

- Declaration:

```
int (*fun_ptr)(int); // notice the () around *fp
int (*fun_ptr)(void*, void*);
int *fun(int); // function returning pointer to int
```

- Function pointers can be assigned, passed to and from function, placed in arrays, etc...

Pointers to Functions

```
int fun(int x) { ... }
int main()
{
    // fun_ptr is a pointer to a function that
    // returns an int and receives an int
    int (*fun_ptr)(int);

    fun_ptr = &fun; // fun_ptr points to fun
    fun_ptr = fun; // same

    int x = (*fun_ptr)(7); // same as x = fun(7)
    int y = fun_ptr(7);    // same as y = fun(7)
}
```

Pointers to Functions

```
int fun(int x) { ... }
int main()
{
    // fun_ptr is a pointer to a function that
    // returns an int and receives something
    int (*fun_ptr)();

    fun_ptr = &fun; // fun_ptr points to fun
    fun_ptr = fun; // same

    int x = (*fun_ptr)(7); // same as x = fun(7)
    int y = fun_ptr(7);    // same as y = fun(7)
}
```

Function name = pointer to the function beginning

- Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
- A function name (label) is converted to a pointer to itself.

```
// add x + y
void add(int x, int y) {
  printf("%d + %d = %d\n", x, y, x + y);
}
// all are the same
void (*p_add1)(int, int) = add;     // OK
void (*p_add2)(int, int) = *add;    // OK
void (*p_add3)(int, int) = &add;    // OK
void (*p_add4)(int, int) = **add;   // OK
```

Function name = pointer to itself

And execution

```
p_add1(1, 2);           // OK
(*p_add1)(1, 2);        // OK
(**p_add1)(1, 2);       // OK
(***p_add1)(1, 2);      // OK
(****p_add1)(1, 2);     // OK

(&p_add1)(1, 2); // NOT OK - ERROR
```

How to simplify?

```
void fun(int a) {
    printf("Value of a is %d\n", a);
}

int main()  {
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

Function name = pointer to itself

```c
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    void (*fun_ptr)(int) = fun;  // & removed

    fun_ptr(10);  // * removed

    return 0;
}
```

Function pointers vs. data pointers

what is different?

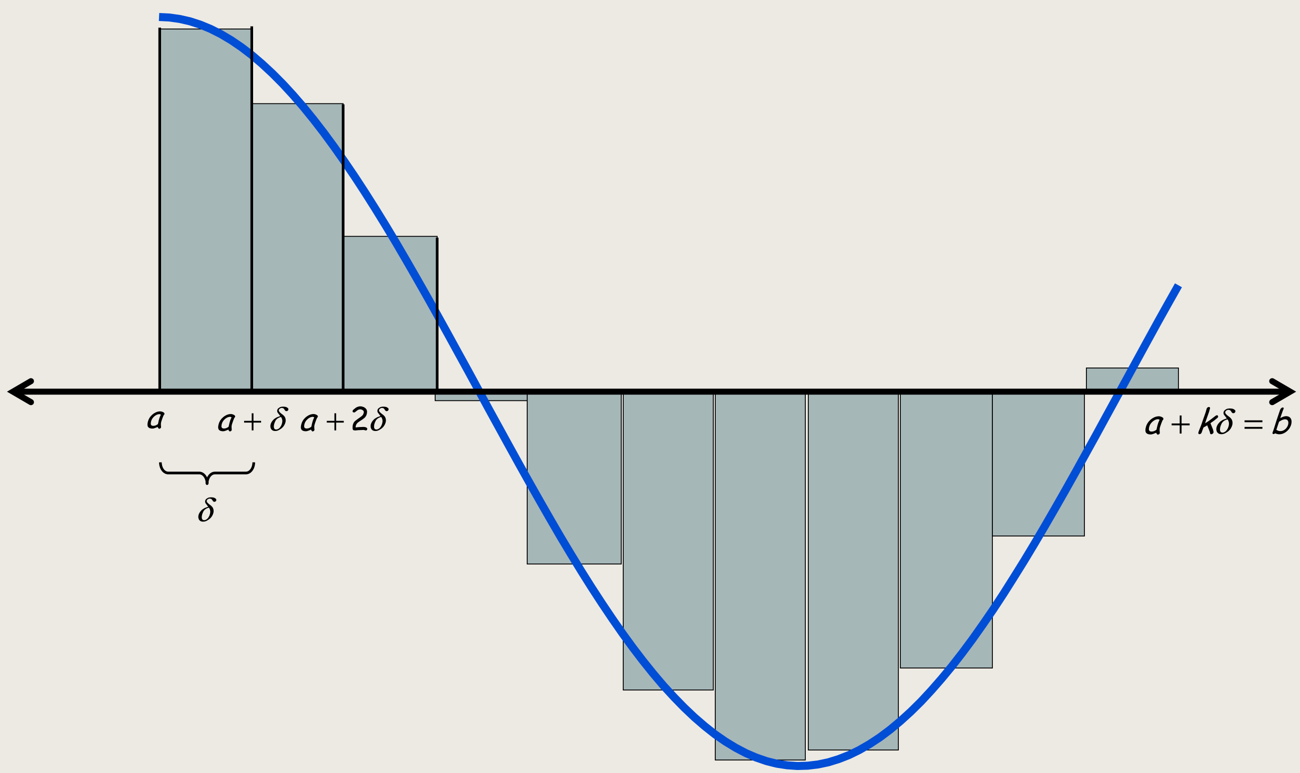
- A function pointer points to code, not data
- No need to allocate de-allocate memory using function pointers
- A function name (label) is converted to a pointer to itself

what is similar?

- A function pointer can be passed as an argument and can also be returned from a function
- We can have an array of function pointers

Example: Numerical Integrator

$$\int_a^b f(x)dx \approx \sum_{i=1}^{k} \delta f\left(a + (i - \frac{1}{2})\delta\right) \qquad \delta = \frac{b-a}{k}$$



Numerical Integrator: Passing function pointer to function

```
double numericalIntegration(
       double a, double b,
       double (*func)(double), int k )
{
   double delta = (b - a)/k;
   double sum = 0;
   for(double x = a + 0.5*delta;
       x < b ;
       x += delta)
   {
      sum += (*func)(x);
   }
   return sum*delta;
}
```

Sort - passing comparator function

```
int arr[] = { 10, 8, 5, 1, 4, 7 }
int asc(void* pa, void* pb)
{
    return (*(int *)pa - *(int *)pb);
}
int desc(void* pa, void* pb)
{
    return (*(int *)pb - *(int *)pa);
}
/ * sort in ascending order */
qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), asc);
/ * sort in descending order */
qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), desc);
```

Array of function pointers instead of switch

```c
enum TYPE { SQUARE, RECT, CIRCLE, TRIANGLE};

struct Shape {
  float params[MAX]; // shape data
  enum TYPE type; // shape type
};

void draw(struct Shape* ps) {
  switch (ps->type) {
      case SQUARE: draw_square(ps); break;
      case RECT: draw_rect(ps); break;
      case CIRCLE: draw_circle(ps); break;
      case TRIANGLE: draw_triangle(ps); break;
}
```

Array of function pointers

The same can be done with an array function pointers:

```
/ * fp is an array of pointers to function */
void (*fp[4])(struct Shape* ps) =
{ &draw_square, &draw_rect, &draw_circle, &draw_triangle};

/ * drawfn is a pointer to function that gets
    Shape pointer and returns nothing */
typedef void (*drawfn)(struct Shape* ps);
drawfn fp[4] =
{ &draw_square, &draw_rect, &draw_circle, &draw_triangle};

void draw(struct Shape* ps) {
  (*fp[ps->type])(ps); // call the correct function
}
```

Function pointers as function arguments

```
// pt2Func is a pointer to a function which
// returns an int and takes a float and two chars
void PassPtr(int (*pt2Func)(float, char, char) )
{
     int result = (*pt2Func)(12.0, 'a', 'b');
}
// execute example code
void Pass_A_Function_Pointer()
{
     PassPtr(&DoIt);
}
```

Function that returns function pointer

```
// What does this signature means?


float (*GetPtr1(const char op))(float, float)
{

                        ...


}
```

Function that returns function pointer

```
// GetPtr1 is a function that gets const char as
// input and returns pointer to function, that
// gets two floats as input and returns a float
float (*GetPtr1(const char op))(float, float)
{
      if(op == '+')
            return &Plus;
      else
            return &Minus;
}
// define a function pointer initialized to NULL
float (*pt2Function)(float, float) = NULL;
pt2Function = GetPtr1('+');          // set value
Float ans = (*pt2Function)(4.5f, 6.5f);      // use
```

typedef is your friend!

```
typedef float(*pt2Func)(float, float);

pt2Func GetPtr2(const char op)
{
      if (op == '+')
            return &Plus;
      else
            return &Minus;
}
```

Function arguments

```
// prototype of function that takes
// undetermined number of arguments
void foo();

// prototype of function that takes
// no arguments
void foo(void);
```

Function arguments

Playing with
the signature of
the pointer vs.
the function –
look at home

Function arguments

```
// add x + y
void add(int x, int y) {
  printf("%d + %d = %d\n", x, y, x + y);
}
// add 1 + 2
void add1and2() {
  printf("1 + 2 = 3\n");
}
```

Function arguments

```
int main() {
   void (*p_add)(); // num args undetermined

   p_add = add;
   p_add(1, 2);

   p_add = add1and2;
   p_add();
   p_add(3, 4); // OK, but prints “1 + 2 = 3”

}
```

Function arguments

```
int main() {
   void (*p_add)(int, int);

   p_add = add;
   p_add(1, 2);

   p_add = add1and2; // OK
   p_add(); // ERROR – too few arguments
   p_add(3, 4); // OK, but prints “1 + 2 = 3”
}
```

Function arguments

```c
// add x + y
void add(int x, int y) {
  printf("%d + %d = %d\n", x, y, x + y);
}
// add 1 + 2
void add1and2(void) {
  printf("1 + 2 = 3\n");
}
```

Function arguments

```
int main() {
   void (*p_add)(); // num args undetermined

   p_add = add;
   p_add(1, 2);

   p_add = add1and2; // OK
   p_add();
   p_add(3, 4); // OK, but prints “1 + 2 = 3”

}
```

Function arguments

```
int main() {
    void (*p_add)(int, int);

    p_add = add;
    p_add(1, 2);

    p_add = add1and2; // WARNING - incompatible
                      //           types
    p_add(); // ERROR – too few arguments
    p_add(3, 4); // OK, but prints “1 + 2 = 3”
}
```

Function arguments

```
int main() {
    void (*p_add)(void);

    p_add = add; // WARNING – incompatible types
    p_add(1, 2); // ERROR – too many arguments
    p_add(); // OK, but prints garbage

    p_add = add1and2;
    p_add(); // OK
    p_add(3, 4); // ERROR – too many arguments
}
```

Uninitialized function pointers

```
int main() {
   void (*p_add)(void);

   p_add(); // WARNING – uninitialized
            // when used
}
```

➔ Most likely will cause the program to crash

Generic programming using pointers to functions

Example: list of ints

Suppose we implement an interface of a list of ints, which is defined as:

```
struct IntList
```

Example

```c
typedef struct IntList IntList;

IntList* intListNew(); // Allocates a new list

void intListFree       (IntList* List );
void intListPushFront (IntList* List, int x);
void intListPushBack  (IntList* List, int x);
int  intListPopFront  (IntList* List);
int  intListPopBack   (IntList* List);
int  intListIsEmpty   (IntList const* List);

typedef void (*funcInt)( int x, void* Data );
void intListMAP( IntList* List,
                 funcInt Func, void* Data );
```

MAP: Perform a function on every member

```c
typedef void (*funcInt)( int x, void* Data );

// Apply Func to each element of the list
void intListMAP(
   IntList* List,
   funcInt Func,
   void* Data
);
```

Implementation of MAP

```
void intListMAP(
     IntList* List, funcInt Func, void* Data)
{
   IntListNode* p;
   for (p=List->start; p!=NULL; p=p->next)
   {
      (*Func)(p->value, Data);
   }
}
```

Usage of MAP

```
typedef struct ListStats {
   int n;
   int sum;
   int sumOfSquares;
} ListStats;

void recordStatistics(int x, void* Data) {
   ListStats *s = (ListStats*) Data;
   s->n++;
   s->sum += x;
   s->sumOfSquares += x * x;
}

void intListStats(IntList* list, double* avg, double* var) {
   ListStats stats =  {  0, 0, 0  };
   intListMAP(list, recordStatistics, &stats);
   if (stats.n > 0)  {
      *avg = stats.sum / (double) stats.n;
      *var = stats.sumOfSquares / (double) stats.n - (*avg) * (*avg);
   }
   else {
      *avg = 0;
      *var = 0;
   }
```

“Generic” interface

Pointers to functions provide a way to write code that receives functions as arguments

MAP is a uniform way of performing computations over list elements - the given function provides the different functional element

Another Example: qsort

Library procedure:

```
void qsort(
  void* base, size_t n, size_t size,
  int (*compare)(void const*, void const*)
);
// base - start of an array
// n - number of elements
// size - size of each element
// compare - comparison function
```

Using qsort

```
int compareInt(void const *p, void const *q)
{
    int a = *(int const*)p;
    int b = *(int const*) q;
    if( a < b )
    {
        return -1;
    {
    return a > b;
}

int array[10] = { 6, 0, 3, 4, 5, 1, 8, 2, 9, 7 };
qsort( array, 10, sizeof(int), compareInt );
```

Variable-Length Array (VLA)

New in C99

Variable-Length Array (VLA)

How to allocate an array with size defined only at run time (instead of compile time)?

ANSI C - use malloc:

```
int length = atoi(argv[ARR_LENGTH]);
int *a = (int *) malloc(length * sizeof(int));
```

C99 - introduce VLA:

```
int length = atoi(argv[ARR_LENGTH]);
int a[length];
```

Variable-Length Array (VLA)

Also…

```
void foo(int x, int y, int a[x][y]) {
    ...
}

int main() {
    int x, y;
    ...
    int a[x][y];
    foo(x, y, a);
    return 0;
}
```

VLA - disadvantages

- There is no way to deal with errors
  - what happens when a size is too large to fit on the stack?
- Underlying memory allocation is not specified: the VLA may be allocated on the stack, heap (or none…).
  - For example, GNU C Compiler allocates memory for VLAs on the stack
  - many embedded operating systems have small stack size
  - you can't `free` the memory
  - you can't use `realloc`
- Not accepted by C++ standard (use `std::vector` instead)
- **Not allowed during the course - compile with `-Wvla`**

Unions

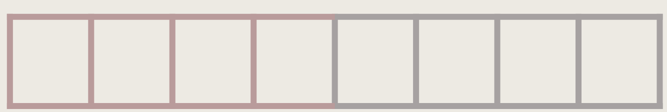
union

- A **union** is data type that enables you to store different data types in the same memory location
- The value of at most **one** of the data members can be stored in a union at any time
- Unions provide an efficient way of using the same memory location for multiple purposes
- The size of a union is sufficient to contain the largest of its data members
- Each data member is allocated as if it were the sole member of a struct

union

Union definition:

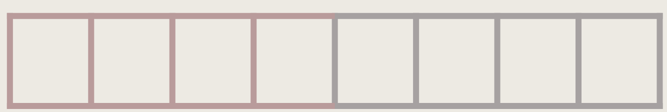
```
typedef union u_all
{
    int i_val;
    double d_val;
} u_all;
```



union

```
u_all u; //definition of the variable u
u.i_val= 3;//assignment to the int member of u
printf("%d\n", u.i_val);

u.d_val= 3.22; //this corrupts the previous
                        assignment
printf("%f\n", u.d_val);
```



It is your responsibility to remember which union member is currently assigned (‘active’)!

union

- **It is your responsibility to remember which union member is currently assigned (‘active’)!**
- You have to store information about the currently active member somewhere else:

```
struct d_union {
   int active;
   union {
     int i_val;
     double d_val;
   } u_all;
}
```

Using union for lexers

A lexer is a program that performs lexical analysis.

The process of separating a stream of characters into different words, called 'tokens' .

Each token is related to a different value

```
typedef enum {KEYWORD, INT_CONST, STRING_CONST} TokenType;
typedef struct Token {
  TokenType type;
  union data {
    struct { int line; } noVal;
    struct { int line; int val; } intVal;
    struct { int line; struct string val; } stringVal;
  };
} Token;
```

See also: http://stackoverflow.com/a/740686

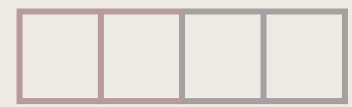
Using union for lexers

```
int main() {
  Token t = parse_a_token(...);
  switch (t.type) {
    case KEYWORD:
      ...
    case INT_CONST:
      printf(“line %d, INT, v=%d\n”,
             t.data.intVal.line, t.data.intVal.val);
      break;
    default:
      ....
  }
  return 0;
}
```

Remember: it is your responsibility to remember which union member is currently assigned!

Using unions for low-level coding

<u>If</u> supported by the compiler, you can write something like this



```
union float_cast
{
    unsigned short as_short[2];
    float f_val;
};
```

So you can look at the bits representing the float value, as two consecutive shorts (if not supported, behavior is undefined).