Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 9

- Global / Static Variables
- Arrays
- C-Structures

Global Variables

- Declared outside of any block
- Numbers initialized with default value 0
- Scope is entire program unless the `static` modifier is used to indicate that the variable's scope is local to the current module
- Should be avoided because of potential name conflicts and accidents (every program part can change global variables!)
- Static and global variables are placed in the process data memory segment

Global Variable Example

```
int global;                       // initialized with 0 (*)

float global_pi = 3.1415926535; // everyone can change it!

const float global_e = 2.718;   // const prevents this!

static int I_am_local_to_the_current_file; // initialized to 0

int main()
{
  float global;         // (**), masks (*), uninitialized

  global = 5;           // changes local variable (**)
  global_pi = 0.0;      // possibly not intended
}
```

Global Variables and Multiple Modules

main.c:

```
#include "global.h"

int main()
{
  global_val = 1.0;
  ...
}
```

global.h:

```
#ifndef GLOBAL_H
#define GLOBAL_H

// declaration
extern int global_val;
...
#endif
```

global.c:

```
// definition
int global_val;

...
```

Static Local Variables

```
int do_something() {
  static int numberOfCalls = 0; // assignment only
  ...                           // done once!!!
  ++numberOfCalls;

  if (!(numberOfCalls % 100)) {
    // print statistics every 100-th call
    ...
  }
}
```

- static modifier
- Global variables in disguise
- Initialized before the function is called for the first time
- They keep their values between calls!

sizeof Operator

sizeof operator can be applied to any type or variable

It returns the number of bytes a variable occupies in memory

E.g.

sizeof(char) == 1
sizeof(int) == 4 (usually)
sizeof(double) == 8

```
int x;
cout << sizeof(x) << endl; // -> 4
```

Arrays

Arrays group together variables of identical type

E.g. declaring array a (8 integers): `int a[8];`

Access by index: `a[i] = 0;`

Elements are laid out consecutively in memory

`int a[8];`

```
address   contents      address   contents


x   ..x+3  a[0]        x+16..x+19 a[4]
x+4 ..x+7  a[1]        x+20..x+23 a[5]
x+8 ..x+11 a[2]        x+24..x+27 a[6]
x+12..x+15 a[3]        x+28..x+31 a[7]
```

This array occupies 8·sizeof(int) = 32 bytes in memory

Array Declaration

```
int N;
const int M = 256;

char A[12];    // OK - 12 characters A[0]..A[11]

int B[N];      // not OK! not a constant expresssion

float C[2*M];  // OK - 512 floats C[0]..C[511]
```

- Syntax: <type> <ident> [<const-int-expr>] ;
- Integer expression defines the number of objects in the array. In case of simple types, they are not initialized!
- Array index always starts with 0

Array Initialization

```
int  A[4];        // 4 integers - not initialized!
int  B[4] = { 4, 3, 2, 1 };  // B[0]=4,..B[3]=1
char C[2] = { ’a’,’b’,’c’ }; // invalid! too many
char D[]  = { ’a’,’b’,’c’ }; // OK, declares D[3]
int  E[2] = { 1 };           // OK, E[0]=1 E[1]=0
```

- <type> <indent>[{<const-int-expr>}] = {<const-expr>,...,<const-expr> };
- The list of constant expressions is evaluated and assigned to the array elements
- If list is shorter than array size, 0s are padded
- Array size can be omitted; it is then equal to the list length

Multi-Dimensional Arrays

- Arrays with more than one index

  `char page[ROWS][COLS];`

  `int table4[2][2][2][2];`
- Rectangular array of array of ...
- Flat memory layout (“mailbox” format)

```
address   contents

x       :  page[0][0] page[0][1]  ... page[0][COLS-1]
x+COLS  :  page[1][0] page[1][1]  ... page[1][COLS-1]
...
x+COLS*:   page[ROWS-1][0]... page[ROWS-1][COLS-1]
(ROWS-1)
           total: ROWS*COLS bytes
```

Example

```
int table[2][2] = { { 0,1 } , { 2,3 } };
// after initialization:
// tab[0][0] = 0, tab[0][1] = 1
// tab[1][0] = 2, tab[1][1] = 3

int add_table_entries()
{
  int s = 0;
  for (int i=0; i < 2; ++i) {
    for (int j=0; j < 2; ++j) {
      s += table[i][j]; // table[i,j] is illegal
    }
  }
  return s;
}
```

Array Access

- Syntax: <ident> [<integer-expression>]
- The expression is evaluated and the array element with that index is accessed
- No index out-of-bounds checks!

```
#include <cassert>
const int N = 10;
int A[N];

for (int i=1; i <= N; ++i) cout << A[i];
// oops! that’s a bug which is hard to detect!

for (int i=1; i <= N; ++i) { // buggy!
  assert(i >= 0 && i < N);   // this kills out-of-
  cout << A[i] << " ";       // bounds bugs dead!
}
```

Arrays as Function Parameters

```
const int N = 10;
int A[N];

void sort(int a[]); // doesn't work, what's a's size?
void sort(int a[], int size); // makes more sense
...
sort(A, sizeof(A)/sizeof(A[0]));  // OK
```

- Arrays are passed by reference
- An array parameter is essentially the array starting address. There is no size information attached to it! Need to pass number of elements
- Functions cannot return arrays

Programming with Arrays: Searching and Sorting

- Common computational tasks
- Need to be implemented efficiently
- Details in algorithms/data structure courses such as CMPUT-204
- Here only some basics to illustrate C/C++ programming with arrays:
  - linear search
  - simple sorting

Search 1

- Task: find an element in an array
- if found, return smallest index, otherwise return -1

```
// precondition: A has at least size elements
// postcondition: returned value is smallest
// index of e in array A, or -1 if not found

int find(int e, const int A[], int size)
{
  for (int i=0; i < size; ++i)
    if (A[i] == e)
      return i;
  return -1;
}
```

Search 2

Task: return index of maximum array element

```
// precondition: A has at least size > 0 elements
// postcondition: returned value is the index
// of the maximum array element

int indexOfMax(const int A[], int size)
{
  assert(size > 0);
  int max_ind = 0;    // current index of maximum value
  int max_val = A[0]; // current maximum value

  for (int i=1; i < size; ++i)
    if (A[i] > max_val) {
      max_val = A[i]; max_ind = i;
    }
  return max_ind;
}
```

Sorting

Task: sort an array in increasing order

Idea: find maximal element, move it to the end, and apply the same algorithm to the remaining array part ("Selection Sort")

```
// precondition: A has at least size elements
// postcondition: A[0] <= A[1] <=...<= A[size-1]

int sort(int A[], int size)
{
  for (int l=size; l > 1; --l) {
    // swap maximal element in A[0..l-1] with A[l-1]
    swap(A[indexOfMax(A, l)], A[l-1]);
  }
}
```

C-Structures

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

Point p;

p.x = 100; p.y = 200;

plot(framebuffer, p, color);
```

```
struct Complex {
  float re, im;
};

Complex a, b, c;
a = add(b, c);
```

- Collection of one or more variables
- Grouped together under a single name
- Called “records” in the Algol family
- Structures help organize data

Struct Definition

```
struct PersonInfo {
  int height;
  int weight;
  Date birthday;
};
```

```
PersonInfo x;

x.height         = 180;
x.weight         = 78;
x.birthday.year  = 1965;
x.birthday.month = 4;
x.birthday.day   = 5;
```

- Data members are laid out in consecutive memory locations
- Recursive structure definitions are not allowed
- Data is accessed by the . operator

Struct Initialization

```
struct Date {
  int year;
  int month;
  int day;
};

Date date = { 1965, 4, 5 };
```

- Structure variables are not initialized by default!
- Explicit initialization: add = { <const-expr>, . . . ,<const-expr> }
- Data members are initialized corresponding to their order in definition

Structures and Functions

```
struct Complex {
  float re, im;
};

Complex add(const Complex &a, const Complex &b)
{
  Complex r;
  r.re = a.re + b.re; r.im = a.im + b.im;
  return r;
}
```

- Structures can be passed by value or by reference
- Passing by reference is faster
- Returning structs is allowed
- Difference to Java: C-structs are allocated on stack

Structure Assignment

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

Point p1, p2;

p1 = p2; // equivalent to p1.x = p2.x; p1.y = p2.y;
```

- Structure variables can occur on the lhs of assignments
- Type of the rhs expression must be identical
- All structure members are copied one by one
- By default, structures can't be compared (but see overloading ==, >, ... for C++ classes)

Struct Memory Layout

- Layout and size of structures depend on compiler and machine architecture!
- In g++ under Linux for Intel/AMD x86 CPUs:
  - ints are aligned to addresses divisible by 4
  - shorts are aligned to addresses divisible by 2

```
struct Foo {
  char a; int b; char c;
} x;

How x is stored in memory:
  x.a    1 byte
  unused 3 bytes
  x.b    4 bytes
  x.c    1 byte
  unused 3 bytes  total 12
```

```
struct Bar {
  char a; char c; int b;
} y;

How y is stored in mem.:
  y.a    1 byte
  y.b    1 byte
  unused 2 bytes
  y.c    4 bytes total 8
```

Structure Memory Layout Continued

- Accessing aligned ints is faster than unaligned ints
- Reason: data bus from CPU to memory is 32, 64, or even 128 bits wide
  - aligned int: just one memory access
  - unaligned int: possibly two accesses!

```
physical memory organization: 4-byte words

 0  1  2  3  int stored at 0..3: 1 access
 4  5  6  7  int stored at 5..8: 2 accesses!
 8  9 10 11
12 13 14 15
...
```

Packed Structures in gcc/g++

```
struct Foo {
  char a; int b; char c;
} x;

How x is stored in memory:
  x.a     1 byte
  unused  3 bytes
  x.b     4 bytes
  x.c     1 byte
  unused  3 bytes total 12
```

```
struct
__attribute__((packed)) Foo
{
  char a; int b; char c;
} x;

How x is stored now:
  x.a     1 byte
  x.b     4 bytes
  x.c     1 byte     total 6!
```

- Save memory with `__attribute__((packed))`
- packed structures: smaller, but slower access
- non-standard C language extension
- Compiles only with gcc/g++