Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 10

- Unions
- Pointers

Unions (1)

```
enum SType { ST_INT, ST_FLOAT, ST_CHAR, ST_DOUBLE };

struct SaveSpace {
  union {       // anonymous union
    int    i;   // all variables stored
    float  f;   // at the same location
    char   c;
    double d;
  };
  SType t;      // what is stored?
};

SaveSpace s;                  // sizeof(s) = 12!
s.f = 3.5; s.t = ST_FLOAT;   // store float value
s.d = 4.7; s.t = ST_DOUBLE;  // store double value
s.i = 5;   s.t = ST_INT;     // store int value
```


Unions (2)

```
union Shared { // regular union
  int    i;    // all variables stored
  float  f;    // at the same location
  char   c;
  double d;
};

struct SaveSpace {
  Shared u;
  SType t;
} s;

s.u.f = 3.5; s.t = ST_FLOAT; // store float value
```

- Space-saving struct (identical syntax)
- All data members are stored at the same location (only works for C types, dangerous!)
- Anonymous unions declare objects rather than types

Pointers and Addresses

- Pointers are variables that contain the address of a variable
- A leading * in a variable declaration indicates a pointer variable; no default initialization!
- In pointer assignments the & (address) operator is used to determine the address of an object in memory (1st byte)

```
int *p; // read: p is a pointer to an int variable
int a;

p = &a;     // the address of a is assigned to p
            // "p points to a"
int *q = p; // q now also points to a
```

Dereferencing Pointers

```
int x = 1, y;
int *ip;    // ip is a pointer to int, or:
            // "the object ip points to is an int"
            // unitialized!
ip = &x;    // ip now points to x
y = *ip;    // y is now 1
*ip = 0;    // x is now 0
*ip += 10;  // increments x by 10
```

- The unary operator * is used for indirection (aka dereferencing)
- When applied to a pointer it accesses the object the pointer points to

Operators & *

- Higher precedence than arithmetic operators
- Same precedence as ++ -- (rtl associativity)
- Sometimes parenthesis are needed!

```
short x = 5;
short *ip = &x;     // a pointer to x
short y = *ip + 1; // takes whatever ip points
                    // to, adds 1 and assigns
                    // the result to y

(*ip)++; // increments what ip points to (x)
++*ip;   // dito

*ip++;   // increments ip! * has no effect here
```


Dynamic Memory Allocation Preview

- Required for dynamic data structures (lists,trees...)
- Reserves memory on memory heap
- Allocate a variable of type T: `T *p = new T;`
- To deallocate (delete) an object a pointer p points to: `delete p;`

```
int *pi = new int; // allocates memory holding one int
// do something with *pi
delete pi;  // integer no longer needed

struct Point { int x, y; };

Point *pp = new Point; // allocates one Point
// do something with *pp
delete pp; // Point no longer needed
```


Pointers and Arrays

- In C there is a strong relationship between pointers and arrays
- Any `[ ]` operation can be expressed by an equivalent pointer expression
- The pointer version used to be faster, but is harder to understand
- Modern compilers generate equally fast code
- Arrays are passed to functions as a pointer to the first element $\leadsto$ size information is lost

Array Example

```
int a[4];
int *pa = &a[0];  // or  = a;  equivalent
   -----------------------------------
  |  a[0]   |  a[1]   |  a[2]   |  a[3]   |

  ^         ^         ^         ^
  pa        pa+1      pa+2      pa+3


*pa     = 1;  // sets a[0] = 1
*(pa+1) = 2;  // sets a[1] = 2
*(pa+2) = 3;  // sets a[2] = 3
*(pa+3) = 4;  // sets a[3] = 4
```


Pointers and Arrays continued

- `pa+C` points to the C-th successor of `*pa`
- `pa-C` points to the C-th predecessor of `*pa`
- The actual address is incremented resp. decremented by `sizeof(*pa) * C`
  E.g. by `4*C` if pa points to an int
- Array variables = constant pointers
  - `pa = a;  // legal`
  - `a = pa;  // illegal`
- `a[i]` equivalent to `*(a+i)`
- `&a[i]` equivalent to `a+i`

Pointer Arithmetic

```
int n; T *p; ...
p = p+n;  // increments p by n*sizeof(T)
p = p-n;  // decrements p by n*sizeof(T)
```

- If p and q point to elements in the same array,
  `==` `!=` `<` `>` `<=` `>=` between p and q work properly
- Pointer subtraction also valid: if p and q point to members of the same array and `p >= q`, then `p-q` is the number of elements from p to q exclusive.
- All other pointer operations are illegal

Pointers and Structures

Two equivalent ways to access structure members via pointers:

- `(*p).member`
- `p->member`

```
struct Point { int x, y; } point, *pp;

pp->x = point.x;
pp->y = point.y;

(*pp).x = point.x; // equivalent
(*pp).y = point.y;

*pp = point;       // equivalent
```

Programming with Pointers Example

- Trees are a special kind of graph
- Graphs consist of nodes and edges that connect two nodes
- Trees: all nodes are connected, no cycles
- In computing science, trees are fundamental dynamic data structures
- Data associated with nodes:
  - Payload
  - Pointers to successor nodes


```
// binary tree: nodes have at most two successors

struct Node {
  int data;            // data associated with node
  Node *left, *right; // pointers to successor nodes
};                     // 0 indicates no successor

// create small tree:     root
//                         /  \
//                        a    b

Node *root = new Node; // all components undefined!
Node *a    = new Node;
Node *b    = new Node;

// *a and *b have no successors (they are "leaves")
a->left = a->right = b->left = b->right = 0;

// connect sucessor nodes a and b to root
root->left = a;  root->right = b;
```


Delete Tree

```
// deleting trees recursively in reverse order
// "what is connected last gets deleted first"

// precondition: n points to the root of a tree

void delete_tree(Node *n)
{
  if (n == 0) return;     // nothing to delete
  delete_tree(n->left);   // delete left subtree
  delete_tree(n->right); // delete right subtree
  delete n;               // finally, delete node
}
```


Pointer Arrays, Pointer to Pointers

```
int *A[4]; // array of 4 pointers to int
A[0] = new int[1]; // row of length 1
A[1] = new int[2]; // row of length 2
A[2] = new int[3]; // row of length 3
A[3] = new int[4]; // row of length 4
A is lower triangular matrix!
access entries with A[i][j] (i:row, j:column)
more memory efficient than multi-dimensional arrays

int **b; // b is a pointer to a pointer to an int
         // or: b points to array of int
```

Pointers are variables themselves, thus

- they can be stored in arrays, and
- can point to pointers