Lecture 9

- C-structs continued
- Pointers
- Dynamic memory allocation
- Pointers vs. arrays
- Pointer arithmetic

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**)

sizeof Operator

```
int main() {
   struct Point { int x, y; };
   struct Foo { char a; int b; char c; };
   struct Bar { char a; char c; int b; };

   cout << sizeof(Point) << ' ';
   cout << sizeof(Foo) << ' ' << sizeof(Bar) << endl;
   return 0;
}

-> 8 12 8   what's going on here? why not 8 6 6?
```

- Unary operator
- Syntax: sizeof(<expression>) or sizeof(<type>)
- Computes the size of an object or type measured in bytes

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 memory:
  y.a    1 byte
  y.b    1 byte
  unused 2 bytes
  y.c    4 bytes   total 8
```

Structure Memory Layout Cont.

- 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 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!
```

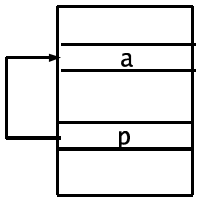
- Save memory with **__attribute__((packed))**
- packed structures: smaller, but slower access
- **non-standard** C language extension
- Compiles only with gcc/g++

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

```
int x = 1, y = 2;
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 = *ip + 10; // increments x by 10
```

- The unary operator * is used for **indirection** or **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
++*ip;    // dito

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


Dynamic Variable Allocation Preview

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

```
// allocates memory holding one int

int *pi = new int;
// do stuff with *pi
delete pi;  // integer no longer needed

struct Point { int x, y; };

// allocates one Point
Point *pp = new Point;
// ...
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

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 (cont.)

- 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 of 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