<table>
<thead>
<tr>
<th>Lecture 13</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CIS 208</td>
<td></td>
</tr>
<tr>
<td>Wednesday, March 8th, 2005</td>
<td></td>
</tr>
</tbody>
</table>
lazy programming with GCC

- GCC allows dynamic array allocation without `malloc/calloc`
- Allows 'anywhere' variable declaration.
- Don't do this. gcc is a C/C++ hybrid compiler.
- Bad programming habits. May not be true on all platforms.
- We will comply with ANSI C standard.
User Defined Types

- 5 ways to define own types
- typedef
- structures (struct)
- union
- enumeration
- bit-field
user defined types

- typedef: defining a new name for existing type. Increases readability.

- structs: user defined data structures. A little like objects, but can only hold data. No inherent functions.

- union: memory location shared by multiple variables.
user defined types

- enumerations: specifying all legal values a variable may have. Again, used for readability/portability

- bit-field: blocks of Boolean values, given labels by programmer
type definitions

- typedef gives new name to existing data type.
- Increases portability/readability
- a little like macros. Instead of having to change all the code, you just reference the defined type. Then the typedef can be changed and all the changes are made automatically.
- We’ll see this later.
typedef

typedef type newname;

typedef float balance; //creates a new type called balance
balance over_due = 1.03; //over_due is a variable of type balance

printf("\%f\n",over_due);
typedef

can typedef another typedef.
typedef float balance;
typedef balance overdraft;
overdraft foo = 3.3;

You have seen this before.
sizeof operator returns a value of type size_t.
size_t is a typedef of int.
typedef scope.

- Normal scope rules are applicable.
- Globals labels
- Local labels.
- Declare at beginning.
Structs

- collection of variables referenced under one name.
- Fundamental component of data structures (linked lists, queues, trees, stacks).

- Can only hold variables (type variables and pointers).
- Once defined, may act just like any other type. It can have pointers, passed as arguments, returned by functions
- No internal functions, no privacy.
struct definition

- Must define what variables are in a struct.

```c
struct addr {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    int zip;
};
```

- ends with semi-colon.
declaration

- definition does not create a variable.
  - addr is the type, not the variable name.
  - addr is just like int, char, float, etc.

- must declare struct variables just like regular variables.

- 2 ways to do it.
declaration

struct addr my_address;

struct foo {
    int a;
    int b;
} my_foo, her_foo;
basic skeleton

struct tag {
    type member_name;
    type member_name;
    .
    .
    .
} structure_variable;

Either the tag or structure_variable may be omitted, but not both.
Accessing Structs

- a little like java. Use the . operator.

```c
struct foo {
    int a;
    int b;
} my_foo;

my_foo.a = 9;
my_foo.b = 10;
printf("%d",my_foo.b);
```
accessing structs

- `structure_variable.member_name`

- period method only works on structure variables.

- Does not work with pointers to structs.
Struct assignments

struct foo a, b;
...
a = b;

- copies each field from b to a. (the values)
- Initial values after declaration are junk.
  - unless it is a global variable
Array of Structs

Just like regular arrays

struct foo a[10];

printf(“%d”, a[0].b);
passing structs to functions

- can pass values in fields, addresses of fields, or the whole struct.

```c
func(my_foo.a);  // passes int value of a
func2(&my_foo.b); // passes address of b
                // & goes before variable name, not field name
func3(my_foo);   // passes struct variable my_foo
func4(&my_foo);  // passes address to variable my_foo
```
functions and structs

\*just like other function arguments*

```c
void func3(struct foo a_foo){}
```

```c
struct foo func5(void){}
```
typedef a struct

- annoying to keep having to say struct.
- create a new type with typedef

```c
typedef struct _foo{
    int a;
    int b;
} foo;

foo c;
```
pointers and structs

- holds address to struct.
- assignment is same

```c
struct foo a,*p;
p = &a;
p now holds the address of a
```
using pointers

- dereferencing a pointer (*p) is the whole struct.

```c
struct foo c,d,*p = &c;
d = *p;
*p = d;
```
accessing with pointers

dereferencing does not work with the interior fields. Can’t just use period.

```c
struct foo c,*p = &c;
printf("%d",c.a, *p.a);    \textbf{BAD}
```

This will cause compiling errors.
accessing pointers

- Must use the arrow operator (->)

```c
struct foo c, *p = &c;
printf("%d", c.a, p->a);
```

- `p->a == (*p).a`
- Don’t dereference `p`, just use the arrow to get the field.
allocation

to create a new struct, must use malloc or calloc

foo *p;
p = (foo *) malloc(1 * sizeof(foo));
free(p);