Objects, functions, and types

Objects

- the C standard refers to objects
  - but these are not the same objects as in object-oriented programming
- in C, an object is a region of data storage in the execution environment, the contents of which can represent values
  - i.e., an object is a chunk of memory that represents a value

Variables

- variables are a type of object
- like Java, variables have a type that is specified when the variable is declared
  - for example:

    ```
    int x;
    ```

    declares a variable named **x** of type **int**

Functions

- functions resemble Java methods
  - no access modifiers in C
- functions are not objects
  - but they have a type in C
    - the type is made up of the return type
    - more on this when we discuss pointers to functions

Pointers

- a pointer is a kind of variable that stores a link to another object
  - the link is the memory address of the object
    - similar to a reference in Java, but you can do a lot more with a pointer in C than you can with a reference in Java

Pointers

- the type of a pointer is derived from the type of the object pointed to
  - if the type of the object pointed to is ***T*** then the type of the pointer is *pointer to* ***T***

```
int* p;    // commonly used in C++
int *p;    // Linux coding standard
int * p;   // less commonly used
```

    all declare a pointer to an **int**
  - there is no agreement on which of the above should be preferred

Pass by value

- C uses pass by value to transfer arguments to functions
  - just like Java
- in Java, it is not possible to write a method that swaps the value of two primitive type variables for the caller

Prints `x = 1, y = 2`

```
public class Swap {

    public static void swap(int x, int y) {
        int tmp = x;
        x = y;
        y = tmp;
    }

    public static void main(String[] args) {
        int x = 1;
        int y = 2;
        swap(x, y);
        System.out.println("x = " + x + ", y = " + y);
    }
}
```

Prints `x = 1, y = 2`

```c
#include <stdio.h>

void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}

int main(void) {
    int x = 1;
    int y = 2;
    swap(x, y);
    printf("x = %d, y = %d\n", x, y);
}
```

Pass by value

- swapping two values for the caller is not possible because the function does not receive the variables storing the values
  - the function receives a copy of the values
- using pointers, it is possible for the function to swap values for the caller

Prints `x = 2, y = 1`

```c
#include <stdio.h>

void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int x = 1;
    int y = 2;
    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```

pointers to **int**

```
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

```
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

If **x** is a pointer, then ***x** is the object that **x** points to. In this context, * is the *pointer dereference* operator.

We say that ***x** dereferences the pointer **x** to access the object that **x** points to.

tmp is assigned the value of the object pointed to by **x**.

```
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

The object pointed to by **x** is assigned the value of the object pointed to by **y**.

```
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

The object pointed to by **y** is assigned the value of **tmp**.

```c
int main(void) {
    int x = 1;
    int y = 2;
    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```

If **x** is an object, then **&x** is the address of **x**. In this context, **&** is the *address-of* operator.

Scope

- the contiguous region of the program where an identifier (name) can be accessed is called the *scope* of the identifier
- four kinds of scope
  1. file scope
  2. block scope
  3. function prototype scope
  4. function scope
     - only applies to labels, function parameters and variables declared inside a function have block scope

File scope

- if an identifier is declared outside of a function or parameter list, then the identifier has file scope
  - the identifier is accessible everywhere in the file it is declared in after the point where it is declared

```c
#include <stdio.h>
int j;

void f(int i) {
    int j = 1;
    i++;
    printf("\tfunc: i = %d, j = %d\n", i, j);
    for (int i = 0; i < 2; i++) {
        int j = 2;
        printf("\t\tfor loop, i = %d, j = %d\n", i, j);
    }
    printf("\tfunc: i = %d, j = %d\n", i, j);
}

int main(void) {
    printf("main: j = %d\n", j);
    f(100);
    printf("main: j = %d\n", j);
}
```

file scope

Block scope

- braces **{ }** denote *blocks* of code
  - similar to Java
- if an identifier is declared inside of a block or in a parameter list, then the identifier has block scope
  - the identifier is accessible everywhere in the block it is declared in after the point where it is declared

```c
#include <stdio.h>
int j;

void f(int i) {
    int j = 1;
    i++;
    printf("\tfunc: i = %d, j = %d\n", i, j);
    for (int i = 0; i < 2; i++) {
        int j = 2;
        printf("\t\tfor loop, i = %d, j = %d\n", i, j);
    }
    printf("\tfunc: i = %d, j = %d\n", i, j);
}

int main(void) {
    printf("main: j = %d\n", j);
    f(100);
    printf("main: j = %d\n", j);
}
```

(outer) block scope

Nested scopes

- scopes can be nested
- identifiers with the same name can be declared in different scopes
  - an identifier declared at an inner scope takes precedence over an identifier declared at an outer scope
    - the outer scope identifier is *hidden* at the inner scope

```c
#include <stdio.h>
int j;

void f(int i) {
    int j = 1;
    i++;
    printf("\tfunc: i = %d, j = %d\n", i, j);
    for (int i = 0; i < 2; i++) {
        int j = 2;
        printf("\t\tfor loop, i = %d, j = %d\n", i, j);
    }
    printf("\tfunc: i = %d, j = %d\n", i, j);
}

int main(void) {
    printf("main: j = %d\n", j);
    f(100);
    printf("main: j = %d\n", j);
}
```

(outer) block scope
- hides **`j`** declared at file scope

```c
#include <stdio.h>
int j;

void f(int i) {
    int j = 1;
    i++;
    printf("\tfunc: i = %d, j = %d\n", i, j);
    for (int i = 0; i < 2; i++) {
        int j = 2;
        printf("\t\tfor loop, i = %d, j = %d\n", i, j);
    }
    printf("\tfunc: i = %d, j = %d\n", i, j);
}

int main(void) {
    printf("main: j = %d\n", j);
    f(100);
    printf("main: j = %d\n", j);
}
```

(inner) block scope
- hides function parameter **i**

```c
#include <stdio.h>
int j;

void f(int i) {
    int j = 1;
    i++;
    printf("\tfunc: i = %d, j = %d\n", i, j);
    for (int i = 0; i < 2; i++) {
        int j = 2;
        printf("\t\tfor loop, i = %d, j = %d\n", i, j);
    }
    printf("\tfunc: i = %d, j = %d\n", i, j);
}

int main(void) {
    printf("main: j = %d\n", j);
    f(100);
    printf("main: j = %d\n", j);
}
```

(inner) block scope
- hides **j** declared at outer block scope

Function prototype scope

- a function prototype is a declaration of a function
  - similar to an abstract method in a Java interface
- includes return type, function name, and parameter types
  - does not include the function body
- a C file that calls a function can be compiled as long as the function prototype is given
  - the function definition is not required for the purposes of compilation

- a parameter in a function prototype has function prototype scope
  - usually not interesting (but see second example below)

```
/* a has function prototype scope */
void some_function(int a);

/* n and a have function prototype scope;
   here, the fact that n is in scope matters */
void another_function(int n, int a[n]);
```

Storage duration

- the lifetime of an object is determined by its *storage duration*
- four kinds of storage duration
  - automatic
  - static
  - allocated
    - discussed later in course
  - thread
    - probably not discussed in CISC220

Automatic storage duration

- the default storage duration of objects declared within a block or as a function parameter
- storage is allocated when the block in which the object was declared is entered and deallocated when it is exited by any means

```c
#include <stdio.h>
int j;

void f(int i) {
    int j = 1;
    i++;
    printf("\tfunc: i = %d, j = %d\n", i, j);
    for (int i = 0; i < 2; i++) {
        int j = 2;
        printf("\t\tfor loop, i = %d, j = %d\n", i, j);
    }
    printf("\tfunc: i = %d, j = %d\n", i, j);
}

int main(void) {
    printf("main: j = %d\n", j);
    f(100);
    printf("main: j = %d\n", j);
}
```

automatic storage duration: lifetime begins when function is entered

lifetime ends when function returns

```c
#include <stdio.h>
int j;

void f(int i) {
    int j = 1;
    i++;
    printf("\tfunc: i = %d, j = %d\n", i, j);
    for (int i = 0; i < 2; i++) {
        int j = 2;
        printf("\t\tfor loop, i = %d, j = %d\n", i, j);
    }
    printf("\tfunc: i = %d, j = %d\n", i, j);
}

int main(void) {
    printf("main: j = %d\n", j);
    f(100);
    printf("main: j = %d\n", j);
}
```

automatic storage duration: lifetime begins when loop is entered

lifetime ends when loop is exited

Static storage duration

- the storage duration of objects declared at file scope
- also the storage duration of block scope variables that are declared **`static`**
  - not the same meaning as **`static`** in Java
- storage duration is the entire execution of the program, and the value stored in the object is initialized only once, prior to the **`main`** function

```c
#include <stdio.h>
int j;

void f(int i) {
    int j = 1;
    i++;
    printf("\tfunc: i = %d, j = %d\n", i, j);
    for (int i = 0; i < 2; i++) {
        int j = 2;
        printf("\t\tfor loop, i = %d, j = %d\n", i, j);
    }
    printf("\tfunc: i = %d, j = %d\n", i, j);
}

int main(void) {
    printf("main: j = %d\n", j);
    f(100);
    printf("main: j = %d\n", j);
}
```

static storage duration: lifetime is the duration of the program

- static local variable example
  - **`counter`** maintains its value between calls to **`increment`**

```c
#include <stdio.h>
int increment(void) {
    static int counter = 0;
    counter++;
    return counter;
}

int main(void) {
    for (int i = 0; i < 5; i++) {
        int count = increment();
        printf("counter = %d\n", count);
    }
    return 0;
}
```

static storage duration: lifetime is the duration of the program

Initialization of static variables

- a static variable is normally initialized when it is declared
- if it is not initialized, then:
  - if it has pointer type, it is initialized to a null pointer;
  - if it has arithmetic type, it is initialized to (positive or unsigned) zero;
  - if it is an aggregate, every member is initialized (recursively) according to these rules;
    - discussed later in course
  - if it is a union, the first named member is initialized (recursively) according to these rules
    - discussed later in course