CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

Subject Matter

- Multiple module (file) programs
- C Data Types
- C Structs

Your Goals

- Look at info for Assessment 3
- Get started on HW4

Modules

- Module: A small coherent piece of C code
  - Each module is arranged in its own file
    - module.c
  - Similar to (but also very different from) a class
  - *How might you decide what goes into a single module?*

```
gcc –o executable main.c module2.c module3.c
```

Modules – header files

- Module: A small coherent piece of C code
  - Modules allow for encapsulation, re-use, implementation independence
- You must have a way to tell other code what is defined in each module
  - module.h
  - Contain 'interface' to a module
  - Forward declaration of functions

```
#include module2.h
#include module3.h
```

Declare

Specifies Name & Datatype

No memory allocated

Can happen many times

```
extern int x;
int fun(float a);
```

Define

Allocates Memory

Functions get body

Can only happen once

```
int x;
int fun(float a)
{return (int)a;}
```

Initialize

Gives a value

Must be defined (have memory)

Could be re-set

```
int x = 10;
```

C datatypes

- **`void`**: a placeholder
- Numbers: **`int`**, **`short`**, **`long`**, **`float`**, **`double`**, … (signed, unsigned)
- **`char`**: really a very short int (1 byte) interpreted as a printable character
- Pointers (**`T*`**): int*, char*, double*, char**…
- Arrays (**`T[]`**): int arr[], char arr[], char* arr[], …
  - Implicit promotion to pointer when passed as an argument to a function or returned from a function
- Booleans? Not defined in C (but **`stdbool.h`**)
  - 0 or NULL is always considered "false" and anything else is true
- Advanced: Union T, Enum E, Function pointers

Type-casting

Convert one type to another

- **Syntax: `(t)e` where `t` is a type and `e` is an expression (same as Java)**
- **If e is a numeric type and t is a numeric type, this is a conversion**
  - **To wider type, get same value**
  - **To narrower type, may not (will get mod)**
  - **From floating-point to integer, will round (may overflow)**
  - **From integer to floating-point, may round (but int to double is exact on most machines)**

```
main() {
    int sum = 17, count = 5;
    double mean;
    mean = (double) sum / count;
    printf("Mean val: %f\n", mean );
}
```

Implicit Casting

- Compiler converts one type to another
  - During arithmetic
  - Convert R-value to L-value
- Type promotion – convert a 'lower' type to a 'higher' type
  - No loss of info
- Demotion – convert a 'higher' type to a 'lower' type
  - May lose some information
  - Floats are *truncated* to ints

Type ranks:

1. **`Char`**
2. **`Short`**
3. **`Int`**
4. **`Long`**
5. **`Float`**
6. **`Double`**
7. **`Long double`**

Pointer casting

- If e has type t1*, then (t2*)e is a (pointer) cast.
- You still have the same pointer (index into the address space).
- Nothing "happens" at run-time.
- Just "getting around" the type system - can write any bits anywhere you want.

```
void evil(int **p, int x) {
  int *q = (int*)p;
  *q = x;
}
void f(int **p) {
  evil(p, 345);
  **p = 17; // writes 17 to address
            // 345 Best case – crash

int *intmem =
  (int*)malloc(sizeof(int))
```

typedef

- Not really a new type - just creating an alias for an existing type
- **`typedef <type> <name>;`**
- In C, strings are "**`char*`**", but if I wanted to actually provide the name "**`string`**", I could!

```
typedef char* string;
int main(int argc, string *argv) {
    string s = "hello, world!";
    printf("%s\n", s);
}
```

structs

- New data collections
  - a record, containing one or more fields
  - Stored adjacently in memory
- Access a field S.f
- If S* Ps then *Ps.f
  - shortcut S->f
- Like Java class, except no methods

```
struct person_info {
  char * name;
  int age;
}
struct person_info *me;
struct person_info you;
me->age = 99;
you.age = 25;
```

Struct Tags

- Has type **`struct person_info`**
- 'Person_info' is a *struct tag*, not a *type*
- Can use typedef to rename

```
struct person_info {
  char * name;
  int age;
}
struct person_info *me;
struct person_info you;
me->age = 99;
you.age = 25;
```

Struct Tags: typdef

- Has type **`struct person_info`**
- 'Person_info' is a *struct tag*, not a *type*
- Can use typedef to rename

```
typdef struct person_info person_info;
struct person_info {
  char * name;
  int age;
}
person_info *me;
person_info you;
me->age = 99;
you.age = 25;
```

Struct Tags: typdef - inline

- Has type **`struct person_info`**
- 'Person_info' is a *struct tag*, not a *type*
- Can use typedef to rename

```
typdef struct person_info {
  char * name;
  int age;
} person_info;

person_info *me;
person_info you;
me->age = 99;
you.age = 25;
```

Structs as parameters & arguments

- Reminder:
- Function parameters initialized with a copy of corresponding argument
- If the argument is a pointer, the parameter value will point to the same thing, of course
- Arrays are passed as pointers (remember?)

- Even with a struct a copy is created
- Since this won't change the original struct, it is more common to use a pointer to the struct
- Avoids copying large objects
- Allows manipulation of original object (can write functions like Java methods)
- But, sometimes, want to pass-by-value. *THINK!!*

Demo: point.c

```c
// Declaring a new struct of type
// "struct Point" and
// typedef to simplify the name to "Point"
typedef struct Point {
  int x;
  int y;
} Point;


// Forward declarations
Point newPoint();
//Point* newDanglingPoint();
Point* newHeapPoint();
```

Demo: point.c

What *is* **`sizeof(Point)`**

```
// construct a point on the heap
Point* newHeapPoint() {
  Point* p = (Point*)malloc (sizeof(Point));
  p->x = 0;
  p->y = 0;
  return p;
  // must free the memory later?
}
```

Structs in Memory

- Allocated contiguous space in memory
  - Members ordered by the order of declaration

```
(void*)pptr+4 ==  &(pptr→y) == &(*pptr).y)
```

Data structures


```
// A single list node that stores an integer as data.
typedef struct IntListNode {
  int data;
  struct IntListNode* next;
} IntListNode;


// Allocates a new node on the heap.
IntListNode* makeNode(int data, IntListNode* next);
```

Data structures - making


```
IntListNode* makeNode(int data, IntListNode* next) {
  IntListNode* n = (IntListNode*) malloc(sizeof(IntListNode));
  if (n) {  // malloc might return null
    n->data = data;
    n->next = next;
  }
  return n;
}
```

Data structures - freeing


```
void freeList(IntListNode* list) {
  while (list != NULL) {
    IntListNode* next = list->next;
    free(list);
    list = next;
  }
}
```