User-defined data types and pointers

William Hendrix

Lecture 12

Outline

- User-defined types
  - `typedef`
  - `enum`
  - `struct`
  - `union`
- Pointers
- Pointer operations

User-defined types

- 5 main ways to define your own variable types:
  - `typedef`
  - `enum`
  - `struct`
  - `union`
  - `class` (later)
- All user-defined types should appear in preamble
- `typedef`
  - Gives another name to an existing type
  - E.g., `typedef double input_t;`
    - Now you can declare variables with type `input_t`
  - Useful if you want to make your code flexible to changing data types (`float` to `double`, `int` to `long`, etc.)

Enumerated types

- Used to denote variables that have a few valid values
  - E.g., weekday: Monday, Tuesday, Wednesday, Thursday, Friday

- Syntax

```
enum weekday {MONDAY, TUESDAY, WEDNESDAY,
THURSDAY, FRIDAY};
```

  - **Convention:** type names are lowercase, values are uppercase
  - Roughly equivalent to using `int` with `MONDAY`, `TUESDAY`, etc., being constants `0`, `1`, `2`, ...
  - Values outside of `enum` declaration are invalid
    - `weekday today = 14; //Wrong`

Structures

- Used to store a collection of data
- Generally used with `typedef`
- Syntax:

```
typedef struct
{
  double price;
  int qty;
  char[12] UPC;
} cart_item_t;
```

- Use `.` to access the elements of the `struct`

```
cart_item_t purchase;
purchase.price = 10.99;
purchase.qty = 3;
```

- structs help to avoid "parallel arrays"
  - Not very common in C++; supplanted by classes
- *Caution*: `struct` elements may be stored in a different order than defined

Unions

- Data type that can store a single value that could represent one of a number of types
- Syntax:

```
typedef union
{
  float f;
  int i;
} numeric_t;
```

  - Each `numeric_t` can hold either a `float` or an `int`
  - Use `.f` or `.i` to treat as `float` or `int` (according to definition)
    - Not the same as a cast; uses binary representation directly
- Size of `union` is max size of any element
- Could be used for an array of mixed types
  - You would need to keep track of "true" data type yourself

Pointers

- Variables that store the memory location for other variables
- Syntax:

```
int* ptr1, * ptr2;
int *ptr1, *ptr2;
```

  - Variable type should precede `*`
  - Dynamic arrays are actually just pointers
  - `char**` is a pointer to a pointer to a `char`
  - Cannot be reference types (no `int*&`)
- Special pointer type: `void*`
  - "Typeless" pointer
  - Can be assigned and cast, but not dereferenced
  - Useful for functions that return a pointer of unknown type or if you want to "obscure" the type
- Pointer size is the same for all types
  - Determined by machine architecture: 32-bit or 64-bit

Pointer operations

- Address (reference) operation: `&`
  - Takes a variable and returns a pointer
  - E.g., `int* ptr = &number;`
    - Always initialize pointers before use!
  - Not to be confused with a reference parameter
    - Reference parameters are implemented with pointers
- Dereference operation: `*`
  - Takes a pointer and returns the variable/value it points to
    - E.g., `x = *ptr + 1;`
  - Can also be an Lvalue (LHS of assignment)
    - `*ptr = *ptr + 1;  //Or (*ptr)++; but not *ptr++;`
- Special pointer value: `NULL`
  - Used to indicate an invalid pointer
    - `(void*) 0`
  - Dereferencing will crash the program

Tonight

Lab 3 is due Monday at 11:59pm