COMP 1633: Intro to CS II

Structures

Charlotte Curtis

February 12, 2024

Where we left off

- C-strings: a special kind of array
- C-string I/O
- C-string functions
- Separate compilation

> *Textbook Section 8.1*

```
const int SIZE = 64;
char sentence[SIZE];

cout << "Enter a sentence: ";
cin.getline(sentence, SIZE);

int i = 0;
int words = 0;
while (sentence[i] != '\0') {
    if (sentence[i] == ' ')
        words++;
}
```

Today's topics

- Grouping data with structures
- Functions + structures
- Arrays + structures
- Assignment 2

> *Textbook Chapter 10*

But first, some `getline` confusion

- C++ has a function in the `<string>` header called `getline`:

```
getline(istream& is, string& str);
```

- **Do not use this** - the `string` class handles all the low-level memory stuff that I want you to learn about
- Instead, use the `getline` **member function** of the input stream:

```
char str[SIZE];
cin.getline(str, SIZE);
```

More clarification: the `const` modifier

- So far we've used `const` in two places:
  - Defining a **named constant**, e.g. `const double GST = 0.05`
- Defining a function parameter as `const`, e.g.:

```
int calc_average(const int values[], int n_vals);
```

- Arguments assigned to a `const` parameter **do not need to be `const`**
- This simply says that the function cannot modify the array
- Similarly, a `const int` can be assigned to the `n_vals` parameter, as the value is *copied* into the function parameter

Separate Compilation

- Typical lab structure:
  - `lab.h`
  - `lab.cpp` - `#include "lab.h"`
  - `main.cpp` - `#include "lab.h"`
- Prevents duplication of the code in `lab.h`, keeps main logic clear
- We could compile in multiple steps:
  - `g++ -c lab.cpp` - compiles `lab.cpp` into `lab.o`
  - `g++ -c main.cpp` - compiles `main.cpp` into `main.o`
  - `g++ -o main main.o lab.o` - links the two object files

What happens when you run `make`?

Compiling in multiple steps is annoying, so we dump it in a `makefile`

```
# This is "Makefile". Notice that comments begin with "#"
program: lab.o main.o
	g++ main.o lab.o –o program
main.o: main.cpp
	g++ -c main.cpp
lab.o: lab.cpp
	g++ -c lab.cpp
```

- Important: the indentation is a **tab**, not spaces! (emacs knows this)

Protecting against multiple `#include`s

- Most projects have many different modules (a somewhat random example)
- For example, in assignment 2 (not yet released):
  - `main.cpp` includes `applicant.h` and `score.h`
  - `score.h` includes `applicant.h`
- Problem: `#include` means "copy and paste" so we're defining stuff twice!
- Solution: **header guards**
  - Wrap your header file in `#ifndef` and `#endif` directives

Header guards

```
#ifndef APPLICANT_H
#define APPLICANT_H

... // contents of applicant.h

#endif // APPLICANT_H
```

- `#ifndef` checks if the macro `APPLICANT_H` is defined
- If it is, the preprocessor skips to the `#endif`
- By convention, the macro name is the header file name in all caps
- Also conventional to put a comment after the `#endif`

Separate Compilation check-in 1/2

Which of the following are good reasons to use separate compilation? Select all that apply.

A. It allows us to reuse code in multiple projects

B. It allows us to separate the main logic from other logical groupings

C. It prevents duplication of code

D. It prevents re-compiling code that hasn't changed

E. It allows us to use `make` to compile our code

Separate Compilation check-in 2/2

The `#include` directive is a preprocessor directive that means:

A. Check if a header has already been included, then include it
B. Copy and paste the contents of the header file into the source file
C. Cross-reference to the associated `.cpp` file
D. Compile the header file into an object file

Moving on to structures

Functions with long lists of parameters are painful:

```
// Calculates the amount owed by the customer based on usage and account limits
void calculate_bill(double base_charge, double usage_limit, double maxMB_used,
                    double endMB, double& over_charge, double& penalty_charge,
                    double& gst_owed, double& total);

// Displays the final bill. If no surcharges are owing, these are not shown.
void print_bill(int account_number, double usage_limit, double beginMB,
                double maxMB_used, double endMB, double base_charge,
                double over_charge, double penalty_charge, double gst_owed,
                double total);
```

- Wouldn't it be nice if we could bundle all that stuff into a single variable?

Structure syntax

- General form:

```
struct <type name> {
    <field1 declaration>;
    <field2 declaration>;
    ...
    <fieldn declaration>;
};
```

- This says "define a new type named `<type name>` with the given fields

> *A **field** (aka member variable or data member) is a term used to describe a single piece of data associated with a common **record***

A structure for bill calculations

```
struct BillInfo {
    int account_num;
    double base_charge;
    double usage_limit;
    double maxMB_used;
    ...
    bool valid;
}
```

- This defines a new type called `BillInfo` with the given fields
- This **does not** declare a variable of type `BillInfo`!

Using your new type

- Once you've defined a new type, you can use it to declare variables:

```
BillInfo user_bill; // a BillInfo instance
BillInfo another_bill; // another BillInfo instance
```

- This is now allocating memory for all the fields in the structure
- Common practice:
  - define structures **globally** so all functions are aware of the new type
  - name structures using UpperCamelCase (PascalCase)

Accessing structure fields

- Like class objects, structure fields can be accessed with **dot syntax**:
  - `user_bill.account_num = 12345;`
  - `user_bill.base_charge = 10.00;`
  - `cout << "Account: " << user_bill.account_num << endl;`
- Once you've accessed via `.`, fields behave just like normal variables
- The fields of a given **instance** are not related to another instance
- Memory for each field is allocated **in order**

Initializing structure fields

- You can initialize structure fields at declaration time:

```
BillInfo user_bill = {12345, 10.00, 1000, 100, true};
```

- But this requires remembering the order of fields and can be error prone
- Like arrays, missing values are initialized to a `0` value of their data type

```
BillInfo user_bill = {};
```

Operations on structures

- You can pass structures to functions:

```
void print_bill(BillInfo bill);
```

- You can return structures from functions:

```
BillInfo read_and_process();
```

- You can even **copy** structures:

```
BillInfo bill1 = {12345, 10.00, 1000, 100, true};
BillInfo bill2 = bill1;
bill2.valid = false; // What happens to bill1.valid?
```

- But you **can't compare them** with `==` or `!=`

Structures and functions

- Unlike arrays, structures are **passed by value** by default
- You can (and usually should) pass structures by reference:

```
void read_bill(BillInfo& bill);
```

- What happens in memory with the following function prototype?

```
void print_bill(BillInfo bill);
```

- Instead of passing by value, good idea to pass by `const` reference

Returning structures from functions

- **Unlike arrays**, structures can be declared in a function and returned
- The structure is **copied** into the caller's memory:

```
BillInfo read_and_process() {
    BillInfo bill;
    // read data into bill
    return bill;
}

// in main
BillInfo user_bill = read_and_process();
```

- For **small structures** this is fine, but for **large structures** this passing a reference is more efficient (visualization)

Structures vs arrays 1/2

Which of the following is **false**?

A. Arrays must contain values of the same type

B. Structures must contain values of the same type

C. Arrays are always passed by reference

D. Structures are passed by value by default

E. Array elements must be accessed by index position

Structures vs arrays 2/2

What can you infer from the function prototypes shown?

A. `a` cannot modify the array
B. `b` cannot modify the structure
C. Both `A` and `B`
D. Neither `A` nor `B`

```
void a(int arr[]);
void b(BillInfo bill);
```

Structures with array fields

- Structures can contain **arrays** (including C-strings) as fields

```
struct Student {
    char name[64];
    int number;
    double gpa;
};
```

- Oddly, this now allows for whole array operations like copying!

```
Student a = {"Bob", 12345, 3.5};
Student b = a;
strcpy(b.name, "Alice");
```

Arrays of structures

- You can also declare arrays of structures:

```
BillInfo bills[10];

for (int i = 0; i < 10; i++) {
    read_bill(bills[i]);
}
```

- This allocates memory for **all fields** of **all instances** in the array
- Standard array rules apply for passing to/returning from functions:

```
void read_and_process(BillInfo bills[]); // passed by reference
void print_bills(const BillInfo bills[]); // mark as read-only
```

Arrays of structures continued

- To access a field of a structure in an array:
  - First, use indexing to access the element in the array
  - Then, use dot notation to access the field of the element
- For example, to set the `i` th bill's account number:

```
bills[i].account_num = 12345;
```

- You can have arrays of structures that have arrays as fields...
- Or even arrays of structures that have arrays of structures as fields...

> *But this is getting a little ridiculous, and probably an indication that your implementation needs work*

Coming up next

- Lab: Structures
- Lecture: File I/O and command line arguments
- Assignment 2 now available