CS3157: Advanced Programming

Lecture #10
Nov 14

Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

- Wrap up C
  - Typedef, Union, Enum
- Starting CPP
  - Background
  - Differences
  - Some basics
  - keywords
- Reading:
  - c++ core ch 1-2

Announcement

- Homework 2 out today
- Please start early and keep up with reading
- Make sure you completed the lab.

typedef

- defining your own types using typedef (for ease of use)

```
typedef short int smallNumber;
typedef unsigned char byte;
typedef char String[100];

smallNumber x;
byte b;
String name;
```

enum

- define new integer-like types as enumerated types:

```
enum weather { rain, snow=2, sun=4 };
typedef enum {
Red, Orange, Yellow, Green, Blue, Violet
} Color;
```

- look like C identifiers (names)
- are listed (enumerated) in definition
- treated like integers
  - start with 0 (unless you set value)
  - can add, subtract — e.g., color + weather
  - cannot print as symbol automatically (you have to write code to do the translation)

enum

- just fancy syntax for an ordered collection of integer constants:

```
typedef enum {
Red, Orange, Yellow
} Color;
```

- is like

```
#define Red 0
#define Orange 1
#define Yellow 2
```

- here's another way to define your own boolean:

```
typedef enum {False, True} boolean;
```

Usage

```
enum Boolean {False, True};

...
enum Boolean shouldWait = True;
...
if(shouldWait == False) { .. }
```

struct

- struct is similar to a field in a Java object definition
- it's a way of grouping multiple data types together
- components can be any type (but not recursive)
- accessed using the same syntax struct.field

```
int main() {
struct {
int x;
char y;
float z;
} rec;
rec.x = 3;
rec.y = 'a';
rec.z = 3.1415;
printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} // end of main()
```

struct

- variables of struct types can be declared in two ways:
  - using a tag associated with the struct definition
  - wrapping the struct definition inside a typedef:

```
int main() {
struct record {
int x;
char y;
float z;
};
struct record rec;
rec.x = 3;
rec.y = 'a';
rec.z = 3.1415;
printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} // end of main()
```

- struct can also be combined with typedef to create a new data type

```
int main() {
typedef struct {
int x;
char y;
float z;
} RECORD;
RECORD rec;
rec.x = 3;
rec.y = 'a';
rec.z = 3.1415;
printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} // end of main()
```

- you can also define arrays of structs and pointers to structs
- note the use of malloc where "sizeof" takes the struct type as its argument (not the pointer!)

```
int main() {
typedef struct {
int x;
char y;
float z;
} RECORD;
RECORD *rec = (RECORD *)malloc( sizeof( RECORD ));
rec->x = 3;
rec->y = 'a';
rec->z = 3.1415;
printf( "rec = %d %c %f\n",rec->x,rec->y,rec->z );
} // end of main()
```

- overall size of struct is the sum of the elements, plus padding for alignment (i.e., how many bytes are allocated)
- given previous examples: sizeof(rec) -> 12
- but, it depends on the size and order of content (e.g., ints need to be aligned on word boundaries, since size of char is 1 and size of int is 4):

```
struct {
char x;
int y;
char z;
} s1;
/* x y z */
/* |----|----|----| */
/* sizeof s1 -> 12 */
```

```
struct {
char x, y;
int z;
} s2;
/* xy z */
/* |----|----| */
/* sizeof s2 -> 8 */
```

- pointers to structs are common — especially useful with functions (as arguments to functions or as function type)
- two notations for accessing elements: (*sp).field or sp->field
- (note: *sp.field doesn't work)

```
struct xyz {
int x, y, z;
};
struct xyz s;
struct xyz *sp;
...
s.x = 1;
s.y = 2;
s.z = 3;
sp = &s;
(*sp).z = sp->x + sp->y;
```

- arrays of structs are also common
- notations for accessing elements: arr[i].field

```
struct xyz {
int x, y, z;
};
struct xyz arr[2];
...
arr[0].x = 1;
arr[0].y = 2;
arr[0].z = 3;
arr[1].x = 4;
arr[1].y = 5;
arr[1].z = 6;
```

unions

- union
- like struct:

```
union u_tag {
int ival;
float fval;
char *sval;
} u;
```

- but only one of ival, fval and sval can be used in an instance of u (think container)
- overall size is largest of elements

code

```
#define NAME_LEN 40

struct person {
  char name[NAME_LEN+1];
  float height;
};

int main( void ) {
  struct person p;
  strcpy( p.name,"suzanne" );
  p.height = 60;
  printf( "name   = [%s]\n",p.name );
  printf( "height = %5.2f inches\n",p.height );
} // end of main()
```

Shift Gears

- Hopefully you feel comfortable looking at c and working in c.
- Some background:
  - Why are we covering all these languages so quickly?
  - What are you supposed to be taking out of the course?
  - How does c++ fit into this?
  - Bottom line
- Intro to c++

differences between c++ and c

  - history and background
  - object-oriented programming with classes

- very brief history...
  - C was developed 69-73 at Bell labs.
  - C++ designed by Bjarne Stroustrop at AT&T Bell Labs in the early 1980's
  - originally developed as "C with classes"
  - Idea was to create reusable code
  - development period: 1985-1991
  - ANSI standard C++ released in 1991

Four main OOP concepts

- abstraction
  - creation of well-defined interface for an object, separate from its implementation
  - e.g., Vector in Java
  - e.g., key functionalities (init, add, delete, count, print) which can be called independently of knowing how an object is implemented
- encapsulation
  - keeping implementation details "private", i.e., inside the implementation
- hierarchy
  - an object is defined in terms of other objects
  - Composition => larger objects out of smaller ones
  - Inheritance => properties of smaller objects are "inherited" by larger objects
- polymorphism
  - use code "transparently" for all types of same class of object
  - i.e., "morph" one object into another object within same hierarchy

Basic differences

- Before we talk about OOP, lets discuss language differences:

1. Naming Conventions of files
2. Comments styles
3. Struct treated differently
4. I/O redesigned
5. Function abstraction enforced

Hello.cpp

```
#include <iostream.h>
#include <stdio.h>
main() {
cout << "hello world\n";
cout << "hello" << " world" << "\n";
printf( "hello yet again!\n" );
}
```

- compile using:

```
g++ hello.cpp -o hello
```

- like gcc (default output file is a.out)

No need for typedef in c++

```
struct, enum and union tags are type names

struct User {
char *name;
char *password;
};
User myuser;

enum Color { red, white, blue };
Color foreground;

union Token {
int ival;
double dval;
char *sval;
};
Token mytoken;
```

iostream.h

- it's preferred not to use C's stdio (though you can), because it's not "type safe" (i.e., compiler can't tell if you're passing data of the wrong type, as you know from getting run-time errors...)
- stdio functions are not extensible
- note << is left-shift operator, which iostream "overloads"
- you can string multiple <<'s together, e.g.:
- cout << "hello" << " world" << "\n";
- cout is like stdout
- cerr is like stderr

Defining your own functions

- must be declared/defined before it is called
- a function's "signature" is its name plus number and type of arguments
- you can have multiple functions with same name, as long as the signatures are different
- example:

```
void foo( int a, char b );
void foo( int a, int b );
void foo( int a );
void foo( double f );
main() {
foo( 1,'x' );
foo( 1,2 );
foo( 3 );
foo( 5.79 );
}
```

- OVERLOADING – when function name is used by more than one function

Function II

- Foo() or Foo(void) for void arguments
  - Different than c
- Foo(…) for unchecked parameters
  - See va_list and va_start
  - Better pass in an array
- Foo(int a, int b, int c=10)
  - Foo(4,5,2)
  - Foo(4,5)

Function III

- Inline functions
- Function overloading:
  - void foo(int a, char c)
  - void foo(char c)

  - Not allowed
    - void foo(int a)
    - int foo(int a)

Other additions

- C++ includes many compiler side additions to help the programmer (yes that is you) to write better code
- Other technical changes (will be pointing them out as we pass them)

const

- Idea: declare which variables will not be changing

const int X = 25;

- Better than #define since error message will be easier to understand since preprocessor not involved
- Some confusion
  - int const * X const int * X //variable pointer to const
  - int * const Y //const pointer to int
  - int const * const Z //const point to const
- Very useful in functions to either return const or make sure a pointer doesn't alter the original object

Void pointers

- C allows you to assign and convert void pointers without casting
- C++ needs a cast

```
void * V;
..
Foo *f = (Foo)V;
```

NULL

- null pointer (0)
- in c, it's a language macro:

```
#define NULL (void *)0
```

- in c++, it's user defined because otherwise an explicit cast is needed!

```
#define NULL 0
```

- but book recommends using 0 instead of NULL

enums

- Are treated a little differently in c++
- enum day {Sunday, Monday , .. }
- day X = 1; //only works in c
- day X = Sunday;

main()

- In C main is the first thing to run
- C++ allows things to run before main, through global variables
- Variable which are declared outside of main, have global scope (will cover limits later).
- Can have function calls here

File conventions

- No one convention
  - .C
  - .cc
  - .cp
  - .cpp ←I prefer this
  - .cxx
  - .c++

Keywords c++

- asm
- catch
- class
- friend
- delete
- inline
- new
- operator
- private
- protected
- public
- this
- throw
- template
- try
- virtual

Over view of assignment

- Extend the lab example
- Integrate perl in c and cgi
- Work with graphics
- Have something cool to show off to your friends or on interviews.

- Hints: if you are sending too much time….ask for help
  - examples

For Next Class

- Start homework

- Read:
  - C++ core, chapters 3-6

  - See you in lab Wednesday

  - About Thanksgiving weekend lab…