CIS 330:

Univ and C/Cpp

Lecture 8: finish structs & unions, function pointers, subtyping
Announcements

• Lecture 9 posted on web (YouTube)
• 3A assigned, on web
• Andy Lab on Friday
  – Will be about installing SW that will really help for Project 3
• Class canceled Weds May 3\textsuperscript{rd} ?????
Review Enum
Enums

• Enums make your own type
  – Type is “list of key words”

• Enums are useful for code clarity
  – Always possible to do the same thing with integers

• Be careful with enums
  – ... you can “contaminate” a bunch of useful words
enum example

C keyword “enum” – means enum definition is coming

```c
enum StudentType {
    HighSchool,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    GradStudent
};
```

This enum contains 6 different student types

semi-colon!!!
enum example

```c
int AverageAge(enum StudentType st)
{
    if (st == HighSchool)
        return 16;
    if (st == Freshman)
        return 18;
    if (st == Sophomore)
        return 19;
    if (st == Junior)
        return 21;
    if (st == Senior)
        return 23;
    if (st == GradStudent)
        return 26;

    return -1;
}
```
But enums can be easier to maintain than integers

```c
enum StudentType
{
    HighSchool,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    PostBacc,
    GradStudent
};

int AverageAge(enum StudentType st)
{
    if (st == HighSchool) return 16;
    if (st == Freshman) return 18;
    if (st == Sophomore) return 19;
    if (st == Junior) return 21;
    if (st == Senior) return 23;
    if (st == PostBacc) return 24;
    if (st == GradStudent) return 26;

    return -1;
}
```

If you had used integers, then this is a bigger change and likely to lead to bugs.
Struct
Data types

- float
- double
- int
- char
- unsigned char

All of these are simple data types
Structs: a complex data type

• Construct that defines a group of variables
  – Variables must be grouped together in contiguous memory

• Also makes accessing variables easier ... they are all part of the same grouping (the struct)
C keyword “struct” – means struct definition is coming

```
struct Ray
{
    double origin[3];
    double direction[3];
};
```

This struct contains 6 doubles, meaning it is 48 bytes

Declaring an instance

```
int main()
{
    struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

“.” accesses data members for a struct
Nested structs

```c
struct Origin {
    double originX;
    double originY;
    double originZ;
};

struct Direction {
    double directionX;
    double directionY;
    double directionZ;
};

struct Ray {
    struct Origin ori;
    struct Direction dir;
};

int main() {
    struct Ray r;
    r.ori.originX = 0;
    r.ori.originY = 0;
    r.ori.originZ = 0;
    r.dir.directionX = 0;
    r.dir.directionY = 0;
    r.dir.directionZ = 0;
}
```

accesses dir part of Ray
accesses directionZ part of Direction (part of Ray)
typedef

- typedef: tell compiler you want to define a new type

```c
typedef struct Ray
{
    double origin[3];
    double direction[3];
} Ray;

int main()
{
    struct Ray r:
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

This saves you from having to type “struct” every time you declare a struct.
Other uses for typedef

• Declare a new type for code clarity
  – typedef int MilesPerHour;
    • Makes a new type called MilesPerHour.
    • MilesPerHour works exactly like an int.

• Also used for enums & unions
  – same trick as for structs ... typedef saves you a word
Project 2C
Worth 4% of your grade

Assignment: You will implement 3 structs and 9 functions. The prototypes for the functions are located in the file prototypes.h (available on the website).

The three structs are Rectangle, Circle, and Triangle, and are described below.

The 3 structs refer to 3 different shapes: Triangle, Circle, and Rectangle. For each shape, there are 3 functions: Initialize, GetArea, and GetBoundingBoxBox. You must implement 9 functions total (3*3).

The prototypes for these 9 functions are available in the file prototypes.h

There is also a driver program, and correct output for the driver program.
Unions
Unions

• Union: special data type
  – store many different memory types in one memory location

```c
typedef union
{
    float x;
    int    y;
    char   z[4];
} cis330_union;
```

When dealing with this union, you can treat it as a float, as an int, or as 4 characters.

This data structure has 4 bytes
Unions

128-223-223-72-wireless:330 hank$ cat union.c
#include <stdio.h>

typedef union
{
  float x;
  int y;
  char z[4];
} cis330_union;

int main()
{
  cis330_union u;
  u.x = 3.5; /* u.x is 3.5, u.y and u.z are not meaningful */
  u.y = 3; /* u.y is 3, now u.x and u.z are not meaningful */
  printf("As u.x = %f, as u.y = %d\n", u.x, u.y);
}
128-223-223-72-wireless:330 hank$ gcc union.c
128-223-223-72-wireless:330 hank$ ./a.out
As u.x = 0.000000, as u.y = 3
Why are Unions useful?

• Allows you to represent multiple data types simultaneously
  – But only if you know you want exactly one of them
• Benefit is space efficiency, which leads to performance efficiency

Unions are also useful for abstracting type. We will re-visit this when we talk about C++’s templates.
Unions Example

typedef struct
{
   int firstNum;
   char letters[3];
   int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
   char letters[3];
   int nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
   int nums[6];
} WY_LICENSE_PLATE;

typedef union
{
   CA_LICENSE_PLATE ca;
   OR_LICENSE_PLATE or;
   WY_LICENSE_PLATE wy;
} LicensePlate;
typedef enum
{
    CA,
    OR,
    WY
} US_State;

typedef struct
{
    char *carMake;
    char *carModel;
    US_State state;
    LicensePlate lp;
} CarInfo;

int main()
{
    CarInfo c;
    c.carMake = "Chevrolet";
    c.carModel = "Camaro";
    c.state = OR;
    c.lp.or.letters[0] = 'X';
    c.lp.or.letters[1] = 'S';
    c.lp.or.letters[2] = 'Z';
    c.lp.or.nums[0] = 0;
    c.lp.or.nums[1] = 7;
    c.lp.or.nums[2] = 5;
}
Function Pointers
Function Pointers

• Idea:
  – You have a pointer to a function
  – This pointer can change based on circumstance
  – When you call the function pointer, it is like calling a known function
Function Pointer Example

128-223-223-72-wireless:cli hank$ cat function_ptr.c
#include <stdio.h>
int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int (*multiplier)(int);
    multiplier = doubler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
    multiplier = tripler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
}
128-223-223-72-wireless:cli hank$ gcc function_ptr.c
128-223-223-72-wireless:cli hank$ ./a.out
Multiplier of 3 = 6
Multiplier of 3 = 9
Function Pointers vs Conditionals

What are the pros and cons of each approach?
**Function Pointer Example #2**

```c
#include <stdio.h>
void doubler(int *X) { X[0] *= 2; X[1] *= 2; }
void tripler(int *X) { X[0] *= 3; X[1] *= 3; }
int main()
{
  void (*multiplier)(int *);
  multiplier = doubler;
  multiplier(A);
  printf("Multiplier of 3 = %d, %d\n", A[0], A[1]);
  multiplier = tripler;
  multiplier(A);
  printf("Multiplier of 3 = %d, %d\n", A[0], A[1]);
}
```

Don’t be scared of extra ‘*’s ... they just come about because of pointers in the arguments or return values.
Simple-to-Exotic Function Pointer Declarations

void (*foo)(void);
void (*foo)(int **, char ***);
char ** (*foo)(int **, void (*)(int));

These sometimes come up on interviews.
Callbacks

• Callbacks: function that is called when a condition is met
  – Commonly used when interfacing between modules that were developed separately.
  – ... libraries use callbacks and developers who use the libraries “register” callbacks.
Callback example

```c
// Callback example

void RegisterErrorHandler(void (*eh)(char *))
{
    error_handler = eh;
}

void Error(char *msg)
{
    if (error_handler != NULL)
        error_handler(msg);
}

double mylogarithm(double x)
{
    if (x <= 0)
    {
        char msg[1024];
        sprintf(msg, "Logarithm of a negative number: %f !!", x);
        Error(msg);
        return 0;
    }

    return log(x);
}
```
Callback example

```c
#include <mylog.h>
#include <stdio.h>

FILE *F1 = NULL;
void HanksErrorHandler(char *msg)
{
    if (F1 == NULL)
    {
        F1 = fopen("error", "w");
    }
    fprintf(F1, "Error: %s\n", msg);
}

int main()
{
    RegisterErrorHandler(HanksErrorHandler);

    mylogarithm(3);
    mylogarithm(0);
    mylogarithm(-2);
    mylogarithm(5);
    if (F1 != NULL)
        fclose(F1);
}
```

```bash
128-223-223-72-wireless:callback hank$
#include <mylog.h>
#include <stdio.h>

FILE *F1 = NULL;
void HanksErrorHandler(char *msg)
{
    if (F1 == NULL)
    {
        F1 = fopen("error", "w");
    }
    fprintf(F1, "Error: %s\n", msg);
}

int main()
{
    RegisterErrorHandler(HanksErrorHandler);

    mylogarithm(3);
    mylogarithm(0);
    mylogarithm(-2);
    mylogarithm(5);
    if (F1 != NULL)
        fclose(F1);
}
```
Function Pointers

• We are going to use function pointers to accomplish “sub-typing” in Project 2D.
Subtyping
Subtyping

• Type: a data type (int, float, structs)
• Subtype / supertype:
  – Supertype: the abstraction of a type
    • (not specific)
  – Subtype: a concrete implementation of the supertype
    • (specific)

The fancy term for this is “subtype polymorphism”
Subtyping: example

• Supertype: Shape
• Subtypes:
  – Circle
  – Rectangle
  – Triangle
Subtyping works via interfaces

- Must define an interface for supertype/subtypes
  - Interfaces are the functions you can call on the supertype/subtypes

- The set of functions is fixed
  - Every subtype must define all functions
Subtyping

• I write my routines to the supertype interface
• All subtypes can automatically use this code
  – Don’t have to modify code when new supertypes are added

• Example:
  – I wrote code about Shapes.
  – I don’t care about details of subtypes (Triangle, Rectangle, Circle)
  – When new subtypes are added (Square), my code doesn’t change
More Unix
“.‖ and “..‖”

• Unix convention:
  – “.‖ : the current directory
  – “..‖ : the parent directory

Quiz: you in /path/to/dir and issue “cd ./.../..”. Where do you end up?

Answer: “/path”
pwd and $PWD

- `pwd`: Unix command that returns the “present working directory”
- `$PWD`: Environment variable that contains the present working directory
- `$OLDPWD`: Environment variable that contains the previous present working directory
- “-”: Shortcut for the previous PWD

```bash
C02LN00GFD58:~ hank$ echo $PWD
/Users/hank
C02LN00GFD58:~ hank$ pwd
/Users/hank
C02LN00GFD58:~ hank$ cd 330
C02LN00GFD58:330 hank$ echo $OLDPWD
/Users/hank
C02LN00GFD58:330 hank$ cd -
/Users/hank
C02LN00GFD58:~ hank$ echo $OLDPWD
/Users/hank/330
C02LN00GFD58:~ hank$ ...
```
PATH environment variable

When the shell wants to invoke a command, it searches for the command in the path.

```
128-223-223-72-wireless:Documents hank$ echo $PATH
/opt/local/bin:/opt/local/sbin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/texbin
128-223-223-72-wireless:Documents hank$ echo $PATH | tr ': ' '
'/opt/local/bin
/opt/local/sbin
/usr/bin
/bin
/usr/sbin
/sbin
/usr/local/bin
/opt/X11/bin
/usr/texbin
128-223-223-72-wireless:Documents hank$
```

“tr”: Unix command for replacing characters (translating characters).
which

which: tells you the directory the shell is finding a command in.
Invoking programs in current directory

shell works with .prog_name since it views this as a path. Hence $PATH is ignored.
Invoking programs in current directory

```
C02LN00GFD58:330  hank$ echo "echo hello world" > my_script
C02LN00GFD58:330  hank$ chmod 755 my_script
C02LN00GFD58:330  hank$ my_script
-bash: my_script: command not found
C02LN00GFD58:330  hank$ ./my_script
hello world
C02LN00GFD58:330  hank$ export PATH=$PATH:. 
C02LN00GFD58:330  hank$ my_script
hello world
C02LN00GFD58:330  hank$   
```
Trojan Horse Attack

• export PATH=.:$PATH
  – why is this a terrible idea?

```
C02LN00GFD58:330 hank$ echo "rm -Rf ~" > ls
C02LN00GFD58:330 hank$ export PATH=.:$PATH
C02LN00GFD58:330 hank$ chmod 755 ls
C02LN00GFD58:330 hank$ ls # this would be bad...
```
Wild Cards

- ‘*’ (asterisk) serves as a wild card that does pattern matching

```bash
C02LN00GFD58:330 hank$ ls *.c
330cp.c    heap_stack.c    struct3.c
copy.c     purify.c       struct4.c
copy2.c    recursive.c    t.c
doubler.c   rw.c           t2.c
doubler_example.c  scope.c   typedef.c
enum.c     stack.c        union.c
enum2.c    struct.c       union2.c
heap.c     struct2.c
```
Wild Cards

• You can use multiple asterisks for complex patterns

```bash
C02LN00GFD58:~ hank$ ls -1 */*.c
330/binary.c
330/cis330.c
Downloads/avtConnComponentsExpression.c
```
if / then / else / fi

• Advanced constructs:

```bash
C02LN00GFD58:~ hank$ cat script
export X=hank
if [[ $X == "childs" ]]; then
  echo "matches"
else
  echo "doesn't match"
fi
C02LN00GFD58:~ hank$ ./script
doesn't match
```
for / do / done

C02LN00GFD58:330 hank$ cat script
for i in s*.c; do
echo $i
wc -l $i
done
C02LN00GFD58:330 hank$ ./script
scope.c
  8 scope.c
stack.c
  18 stack.c
struct.c
  16 struct.c
struct2.c
  19 struct2.c
struct3.c
  33 struct3.c
struct4.c
  16 struct4.c
C02LN00GFD58:330 hank$
-f and -d

• -f : does a file exist?
• -d : does a directory exist?

example:

    if [[ ! -d include ]]; then mkdir include ; fi
Bonus Material
Problem with C...

C02LN00GFD58:330  hank$  cat doubler.c
float doubler(float f) { return 2*f; }
C02LN00GFD58:330  hank$  gcc -c doubler.c
C02LN00GFD58:330  hank$  cat doubler_example.c
#include <stdio.h>

int doubler(int);

int main()
{
    printf("Doubler of 10 is %d\n", doubler(10));
}
C02LN00GFD58:330  hank$  gcc -c doubler_example.c
C02LN00GFD58:330  hank$  gcc -o doubler_example doubler.o doubler_example.o
C02LN00GFD58:330  hank$  ./doubler_example
Doubler of 10 is 2
Problem with C...

C02LN00GFD58:330  hank$  nm  doubler.o
0000000000000048  s  EH_frame0
0000000000000000  T  _doubler
0000000000000060  S  _doubler.eh
C02LN00GFD58:330  hank$  nm  doubler
doubler.c  doubler_example  doubler_example.o
doubler.o  doubler_example.c  doubler_user.o
C02LN00GFD58:330  hank$  nm  doubler_example.o
0000000000000068  s  EH_frame0
0000000000000032  s  L_.str
   U  _doubler
0000000000000000  T  _main
0000000000000080  S  _main.eh
   U  _printf

No checking of type...
Problem is fixed with C++...

```c
#include <stdio.h>

int doubler(int);

int main()
{
    printf("Doubler of 10 is %d\n", doubler(10));
}
```

```
C02LN00GFD58:330 hank$ cat doubler.c
float doubler(float f) { return 2*f; }
C02LN00GFD58:330 hank$ g++ -c doubler.c
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is deprecated
C02LN00GFD58:330 hank$ cat doubler_example.c
#include <stdio.h>

int doubler(int);

int main()
{
    printf("Doubler of 10 is %d\n", doubler(10));
}
C02LN00GFD58:330 hank$ g++ -c doubler_example.c
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is deprecated
C02LN00GFD58:330 hank$ g++ -o doubler_example doubler_example.o doubler.o
Undefined symbols for architecture x86_64:
  "doubler(int)", referenced from:
    _main in doubler_example.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
C02LN00GFD58:330 hank$
```
Problem is fixed with C++...
Mangling

• Mangling refers to combing information about the return type and arguments and “mangling” it with function name.
  – Way of ensuring that you don’t mix up functions.

• Causes problems with compiler mismatches
  – C++ compilers haven’t standardized.
  – Can’t take library from icpc and combine it with g++.
C++ will let you overload functions with different types

```c
c02ln00gfd58:330 hank$ cat t.c
float doubler(float f) { return 2*f; }
int doubler(int f) { return 2*f; }
c02ln00gfd58:330 hank$ gcc -c t.c
t.c:2:5: error: conflicting types for 'doubler'
int doubler(int f) { return 2*f; }
^
t.c:1:7: note: previous definition is here
float doubler(float f) { return 2*f; }
^
1 error generated.
c02ln00gfd58:330 hank$ g++ -c t.C
c02ln00gfd58:330 hank$
```
C++ also gives you access to mangling via “namespaces”

```c
#include <stdio.h>

namespace CIS330
{
  int GetNumberOfStudents(void) { return 56; }
}

namespace CIS610
{
  int GetNumberOfStudents(void) { return 9; }
}

int main()
{
  printf("Number of students in 330 is %d, but in 610 was %d\n",
           CIS330::GetNumberOfStudents(),
           CIS610::GetNumberOfStudents());
}
```

Functions or variables within a namespace are accessed with “::”
C++ also gives you access to mangling via “namespaces”

The “using” keyword makes all functions and variables from a namespace available without needing “::”. And you can still access other other namespaces.

```cpp
namespace CIS610
{
    int GetNumberOfStudents(void) { return 9; }
}

using namespace CIS330;

int main()
{
    printf("Number of students in 330 is %d, but in 610 was %d\n",
            CIS610::GetNumberOfStudents(),
            CIS610::GetNumberOfStudents());
}
```

```bash
hank$ cat cis330.C
hank$ g++ cis330.C
hank$ ./a.out
Number of students in 330 is 56, but in 610 was 9
```
Backgrounding

• “&”: tell shell to run a job in the background
  – Background means that the shell acts as normal, but the command you invoke is running at the same time.

• “sleep 60” vs “sleep 60 &”

When would backgrounding be useful?
Suspending Jobs

• You can suspend a job that is running
  Press “Ctrl-Z”

• The OS will then stop job from running and not schedule it to run.

• You can then:
  – make the job run in the background.
    • Type “bg”
  – make the job run in the foreground.
    • Type “fg”
    – like you never suspended it at all!!
Web pages

• ssh –l <user name> ix.cs.uoregon.edu
• cd public_html
• put something in index.html
• → it will show up as

http://ix.cs.uoregon.edu/~<username>
Web pages

- You can also exchange files this way
  - `scp file.pdf <username>@ix.cs.uoregon.edu:~/public_html`
  - point people to `http://ix.cs.uoregon.edu/~<username>/file.pdf`

Note that `~/public_html/dir1` shows up as `http://ix.cs.uoregon.edu/~<username>/dir1`

("~/dir1" is not accessible via web)