What do you think?

Work together!

Common valgrind errors are ‘invalid write’ and ‘invalid read’

How might you interpret these? What sorts of coding errors might lead to these messages?

CSE 374 Lecture 13

Review Items

Scoping

- Scopedemo.c
- Info locals
- Info variables

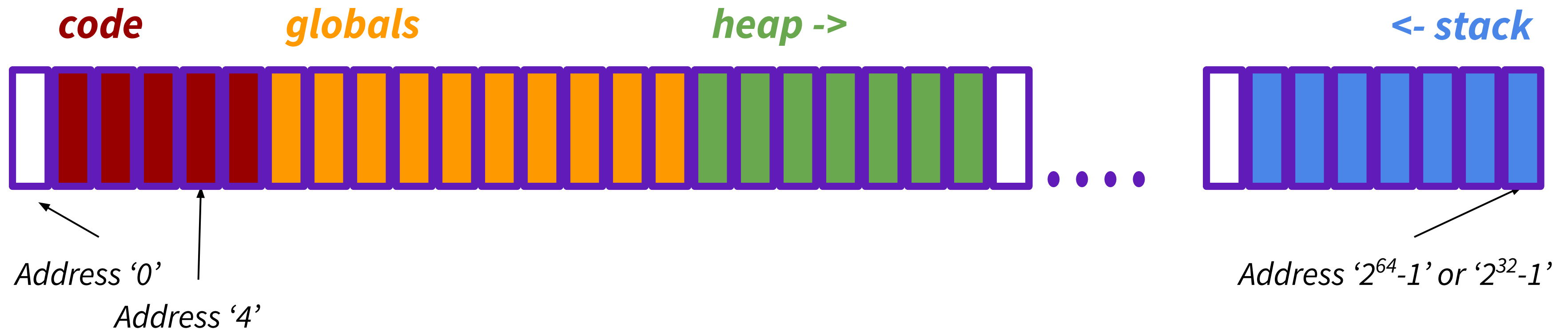
Code structure

- Square1.c demo
  - Declare many times, define once
  - Should declare before you use
- Function divisions
  - Module divisions
- Commenting

Review Items

Scoping

- Scopedemo.c



Review Items

Scoping

- Scopedemo.c
- Info locals
- Info variables

Code structure

- Square1.c demo
    - Declare many times, define once
    - Should declare before you use
- Function divisions
    - Module divisions
- Commenting

Source File Structures

```
// includes for functions & types
defined elsewhere
#include <stdio.h>
#include "localstuff.h"
// symbolic constants
#define MAGIC 42
// global variables (if any)
static int days_per_month[ ] = { 31,
28, 31, 30, …};
// function prototypes
// (to handle "declare before use")
 void some_later_function(char, int);
// function definitions
void do_this( ) { … }
char *return_that(char s[ ], int n)
{ … }
int main(int argc, char ** argv) { … }
```

Preprocessor

Pre-processes your C code before the compiler gets to it.

- ➔ Follows commands prefaced by ‘#’
- ➔ Includes content of header files
- ➔ Defines constants and macros
- ➔ Conditional compilation (not covered right now)

File inclusion

- ➔ #include <foo.h>
  - ◆ Searches for foo.h in “system include” directories (/usr/include, etc)
- ➔ #include “foo.h”
  - ◆ Starts by searching in current directory (allows coder to break project into smaller files)
- ➔ Include include file’s preprocessed contents
- ➔ Recursively include all the includes from original file
- ➔ Use gcc -l dir1 to tell gcc to look for include in dir1

Demo magic.c

Datatypes in C

- 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 *(converting 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("Value of mean : %f\n", mean );
}
```

`(int*)malloc(sizeof(int))`

Implicit casting

- When necessary the compiler automatically converts from one type to another (more general) type
  - Promotes to integers, then to larger integers, then to floating point
  - During arithmetic
  - R-value converted to L-value

For details:
https://www.oreilly.com/library/view/c-in-a/0596006977/ch04.html

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
}
```

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
- Like Java class, except no methods
- Access a field S.f
- If S* Ps then *Ps.f
  - shortcut S->f

```
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-tags

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;
}
```

Struct-tags

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;
```

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?)

(Demo: point.c)

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!!*

Typedef struct ex.

```
typedef int int32;  // use int32 for portability
typedef struct point {  // type tag optional (sortof)
   int32 x, y;
} Point2d;   // Point2d is synonym for struct point
typedef Point2d * ptptr;  // pointer to Point2D
Point2d p; // var declaration
ptptr ptlist; // declares pointer
```

Linked Lists

Points to the List

Last node doesn’t point to next

```
// A single list node that stores an
integer as data.

typdef struct IntListNode {
  int data;
  struct IntListNode* next;
} IntListNode;
```

```
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;
}
```