More Standard Library and Program Organization

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics to Discuss

- Environment Variables
- Variable numbers of parameters
- Nonlocal jumps
- Program structure
  - Abstract Data Types
  - Generic ADTs

Environment Variables

- Your execution environment supports variables
  - We can access these from the shell

```
$ echo $HOME
/home/jerry
$ NAME=linda
$ echo $NAME
linda
```

- The shell uses these a lot for configuration and to customize behavior
  - `HOME, PATH, SHELL, PS1, EDITOR`, …
- But, these aren't just available to the shell, all programs have environment variables.

Using the Environment

- The standard library lets us get/set these variables
  - We need our good friend, `stdlib.h`
  - Then, we can gen environment variable values

```
char *str = getenv( "HOME" );
printf( "HOME = %s\n", str );
```

  - And, we can change them

```
setenv( "HOME", "/some/other/place", 1 );
```

Variable to set.

new value.

Should I overwrite?

- These changes are local to the program, they go away when it exits.

Variable Numbers of Arguments

- We can define functions that take a variable number of arguments
  - printf() needs this
  - The notation looks like this:

- Typically, you'll have:
  - One or more named arguments (at least one is required)
  - Followed by a variable number of unnamed arguments

Processing Variable Arguments

- We can handle variable numbers of arguments … but we'll need some help.

```
#include <stdarg.h>
```

- We'll need to do four things
  - Create a variable of type `va_list`, to store vararg state
  - Call `va_start()`, to start processing variable arguments
  - Call `va_arg()` for each argument, to extract it with the right type
  - Cll `va_end()` when we're done

Counting Arguments

- There's no automatic mechanism to tell how many arguments we got
- We'll need to build-in a way to do that
  - Like printf(), we could use one of the named arguments to figure it out
  - Or, we could use a mechanism to mark the last argument
- Let's try both of these …

Adding Multiple Arguments

```
int addIntegers( int n, ... )
{
  va_list ap;

  va_start( ap, n );

  int total = 0;
  for ( int i = 0; i < n; i++ )
    total += va_arg( ap, int );

  va_end( ap );

  return total;
}
```

State for vararg processing.

Start processing after n.

Get the next int argument.

All done.

```
int sum = addIntegers( 5, 17, 42, 98, 3, 27 );
```

```
int sum2 = addIntegers( 29, 17, 42, 98, 3, 27 );
```

It's our job to use the function properly.

Printing Multiple Strings

```
void printStrings( char *first, ... )
{
  va_list ap;
  va_start( ap, first );

  printf( "%s\n", first );

  char *str;
  while ( ( str = va_arg( ap, char * ) ) ) {
    printf( "%s\n", str );
  }

  va_end( ap );
}
```

We must have at least one named parameter.

Keep going until we hit the NULL.

```
printStrings( "one", "two", "three", "four", NULL );
```

A null to mark the end.

Max of Multiple Values

```
double maximum( int n, double v1, ... )
{
  va_list ap;
  va_start( ap, v1 );

  double largest = v1;

  for ( int i = 1; i < n; i++ ) {
    double v = va_arg( ap, double );
    if ( v > largest )
      largest = v;
  }

  va_end( ap );
  return largest;
}
```

How many values?

Force the caller to give us at least one.

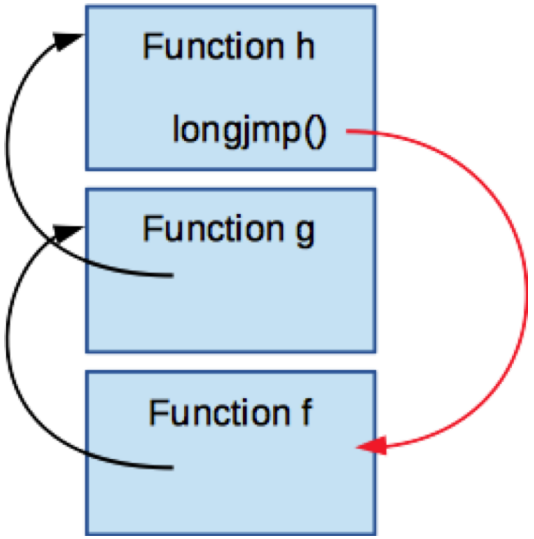
Handle the first value as a named argument.

Handle the rest through varargs.

```
max = maximum( 6, 32.1, 15.32, 2.99, 45.2, 13.62, 7.0 );
```

Nonlocal Jumps

- A goto will let you jump anywhere
  - … as long as it's in the same function.
  - Maybe an OK mechanism for handling errors
- C also supports a long jump
  - To jump out of a function back to one of its callers
  - Made for handling errors
  - Kind of like an exception in Java



Nonlocal Jumps

- Supported by the setjmp.h header.
- Before you can longjmp, you must take a picture of the place you want to jump back to.
  - This is called your program's environment
  - Where it's executing
  - And where the top of the stack is
- The environment must be stored in instance of jmp_buf

```
jmp_buf env;
```

Taking a Picture

- The setjmp() macro saves the state of the environment.

```
setjmp( env );
```

- Initially, it returns zero.
- You need to check this (for later).

```
if ( setjmp( env ) == 0 ) {
  ..
```

Hitting Rewind

- If something goes wrong, calling longjmp() will take you back to that picture.

```
if ( something_went_wrong )
  longjmp( env, 1 );
```

Some int you can use to indicate what went wrong.

- You can allocate the environment statically
- Or pass it to the function for it to use.
- But, you can only longjmp() **up the stack.**

Waking up in setjmp()

- The longjmp() takes you right back to the matching setjmp() call

```
if ( something_went_wrong )
  longjmp( env, 1 );
```

```
Some other function.
```

```
jmp_buf env;
if ( setjmp( env ) == 0 ) {
   Call some other function.
```

- So it looks (almost) like you're right back where you started.

Waking up in setjmp()

- After a longjmp(), the setjmp() won't return zero.
  - It returns the in you you passed to longjmp()

```
if ( something_went_wrong )
  longjmp( env, 1 );
```

```
Some other function.
```

```
jmp_buf env;
if ( setjmp( env ) == 0 ) {
   Call some other function.
} else {
   Error handling code.
}
```

I'll return 1 this time.

Using setjmp()/longjmp()

- We can use this kind of like an exception.

```
void middleFunction( jmp_buf *eenv )
{
  int sum = readValues( eenv );
  printf( "Sum: %d\n", sum );
}
```

```
void topFunction()
{
  jmp_buf eenv;

  if ( setjmp( eenv ) == 0 ) {
    middleFunction( &eenv );
  } else {
    printf( "Invalid input!\n" );
  }
}
```

Plan ahead for where to go if something fails.

Passing the error-handling environment among functions.

An error in here could jump past this middle function.

Using setjmp()/longjmp()

- Here's where we notice the error.

```
int readValues( jmp_buf *eenv )
{
  int sum = 0;
  int m, v;

  while ( ( m = scanf( "%d", &v ) ) == 1 )
    sum += v;

  if ( m != EOF )
    longjmp( *eenv, 1 );

  return sum;
}
```

Jump back to the error-handling code.

Skipping the printout in the middle function.

Cleaning Up

- What if we had some resources to free?

```
int readValues( int *list, int capacity, FILE *fp, jmp_buf *eenv )
```

```
void middleFunction( jmp_buf *eenv )
{
  FILE *fp = fopen( "input.txt", "r" );
  if ( !fp )
    longjmp( *eenv, 1 );

  int capacity = 100;
  int *list = (int *)malloc( capacity * sizeof( int ) );

  int vcount = readValues( list, capacity, fp, eenv );

  for ( int i = vcount - 1; i >= 0; i-- )
    printf( "%d\n", list[ i ] );

  fclose( fp );
  free( list );
}
```

Now, this function reads from a file … into an array.

A jump past this function would leak these resources.

We need something like a finally block for this.

Cleaning Up

- We can catch and re-throw to free resources.
- With no changes to the other functions.

```
void middleFunction( jmp_buf *parentEnv )
{
  FILE *fp = fopen( "input.txt", "r" );
  if ( !fp ) longjmp( *parentEnv, 1 );
  int *list = (int *)malloc( 100 * sizeof( int ) );

  jmp_buf midEnv;
  if ( setjmp( midEnv ) == 0 ) {
    int vcount = readValues( list, capacity, fp, &midEnv );
    ... print the list …
    fclose( fp );
    free( list );
  } else {
    fclose( fp );
    free( list );
    longjmp( *parentEnv, 1 );
  }
}
```

Our own environment, to catch the jump.

Tell the readValues() function to go here on an error.

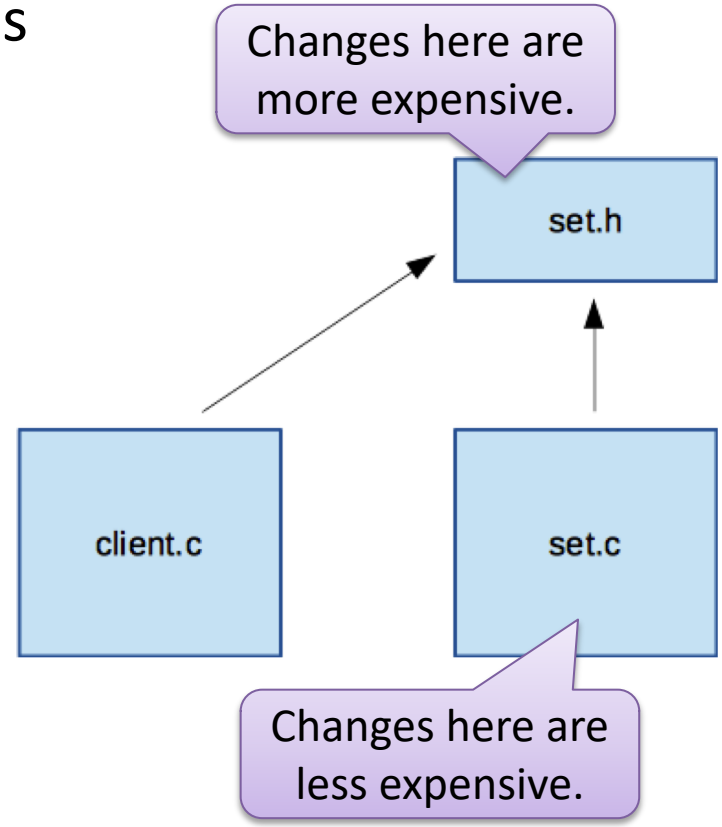
Clean up our part, and jump on up the stack.

Designing a Component

- We're going to look at a few alternatives for designing and implementing a component.
- It will support a set of (integer) values.
  - makeSet() : to initialize the representation
  - insertValue() : to add a value to the set
  - containsValue() : to report whether a value is in the set.
  - destroySet() : to free any resources used by the set.

Designing a Component

- The set component will have a header to advertise its interface.
- And an implementation file (to implement that interface)
- *Client code* is anything else that uses this component.
- We'd like to put as little as possible in the header.
  - Why? If something's in the header, we can't change it without recompiling (all) the client code.
  - Imagine if someone had to fix a bug in stdio.h!



Hiding the Implementation

```
void makeSet();
void insertValue( int val );
bool containsValue( int val );
void destroySet();
```

set.h

```
#include "set.h"

struct Node {
  int val;
  struct Node *next;
};
static struct Node *head;

void makeSet()
{
   head = NULL;
}

void insertValue( int val )
{
  for ( struct Node *n = head; *n; n = n->next )
    ...
```

set.c

Hidden Representation

We could change this representation without breaking client code.

Client Code

```
#include "set.h"

int main()
{
  makeSet();

  insertValue( 2 );
  insertValue( 3 );
  insertValue( 5 );
  insertValue( 7 );
  insertValue( 11 );

  for ( int i = 0; i <= 12; i++ )
    if ( containsValue( i ) )
      ...

  destroySet();
}
```

Initialize the set

Perform some operations with it.

Clean up when we're done.

Abstract Object

- The textbook would call this an abstract object.
  - You can use this object without being sensitive to how it's implemented.
- But, here, we can only have one set.
  - We'd like client code to be able to make as many sets as they want.
  - We need to give client code a type that represents a set.

Exposing the Implementation

set.h

```
struct Node {
  int val;
  struct Node *next;
};

typedef struct {
  struct Node *head;
} Set;

void makeSet( Set *s );
void insertValue( Set *s, int val );
bool containsValue( Set *s, int val );
void destroySet( Set *s );
```

Here, use this type for each set you need.

Every function takes a (pointer to the) set it's supposed to operate on.

set.c

```
#include "set.h"

void makeSet( Set *s )
{
  s->head = NULL;
}

...
```

Client Code

```
#include "set.h"

int main()
{
  Set s;
  makeSet( &s );

  insertValue( &s, 2 );
  insertValue( &s, 3 );
  insertValue( &s, 5 );
  insertValue( &s, 7 );
  insertValue( &s, 11 );

  for ( int i = 0; i <= 12; i++ )
    if ( containsValue( &s, i ) )
      ...

  destroySet( &s );
}
```

Make an instance of set.

Initialize it, passing the address of the uninitialized set we just made.

Perform some operations with it.

Clean up when we're done.

A Bad Idea

```
#include "set.h"

int main()
{
  Set *s;
  makeSet( s );

  insertValue( s, 2 );
  insertValue( s, 3 );
  insertValue( s, 5 );
  insertValue( s, 7 );
  insertValue( s, 11 );

  for ( int i = 0; i <= 12; i++ )
    if ( containsValue( s, i ) )
      ...

  destroySet( s );
}
```

All these functions want a pointer to a Set, so that's what I'll give them.

This will compile … but I bet it will crash when you run it.

No More Abstraction?

- To let client code make instances of Set, we had to let it see our representation.
- This is bad
  - Maybe our design is super secret (less important)
  - Maybe this makes the header larger, which will slow compilation for all code using it (less important)
  - Maybe we want to be able to change our design or fix our implementation later (more important)
- Can we let client code use the Set type … while keeping its contents secret?

Incomplete Type

- Yes. We can give what's called a *incomplete type*.
  - We can tell the compiler about a struct, without saying what's in it.

```
struct ListStruct;
typedef struct RectangleStruct Rect;
```

There's a structure named this.

There's a structure named this, and here's a type name for it.

- This permits name-only use of the struct.
  - You can work with pointers to it.
  - … but you can't do anything that depends on its size or its contents.

Re-Hiding the Implementation

- We can use an incomplete type to insulate client code from the implementation.
- But, client code can only use this type as an *opaque pointer*.
- We'd call this design an *Abstract Data Type* (ADT).
- Now, the header might look like:

```
typedef struct SetStruct Set;

Set *makeSet();
void insertValue( Set *s, int val );
bool containsValue( Set *s, int val );
void destroySet( Set *s );
```

Here, use this type when you need a Set.

But, I'm not telling you anything about it. You have to ask me to make one for you.

Looking Inside

```
#include "set.h"

struct Node {
  int val;
  struct Node *next;
};

struct SetStruct {
  struct Node *head;
};

Set *makeSet()
{
  Set *s = (Set *) malloc( sizeof( Set ) ) ;
  s->head = NULL;
  return s;
}
...
void destroySet( Set *s )
{
  ...

  free( s );
}
```

set.c

In the implementation file, we provide the type we provide the full type definitions.

I'll have to give you a dynamically allocated instance (so, more expensive than stack allocation).

And, you'll need my help when you're done with this object.

Client Code

```
#include "set.h"

int main()
{
  Set *s = makeSet();

  insertValue( s, 2 );
  insertValue( s, 3 );
  insertValue( s, 5 );
  insertValue( s, 7 );
  insertValue( s, 11 );

  for ( int i = 0; i <= 12; i++ )
    if ( containsValue( s, i ) )

  destroySet( s );
}
```

Please make one of those Set objects I've heard about.

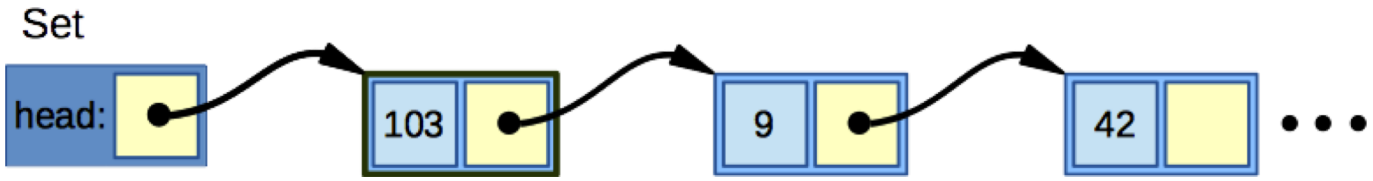
The rest of this is the same, except s is now a pointer.

A Generic, Abstract Data Type

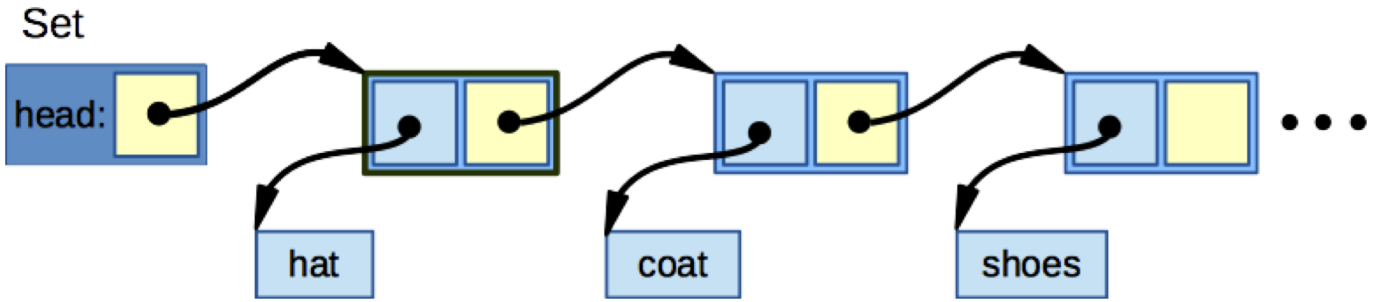
- All we have is a set of ints
- What if we wanted a set of something else?
  - We'd have to make another component
  - … but much of its code would be the same
  - If we were clever, we could make this code generic.
- We would call this a *Generic ADT*

A Generic, Abstract Data Type

- In the implementation, we'd have to be able to store any type of data in our set.
- Previously, we stored a fixed-sized value right inside each node.

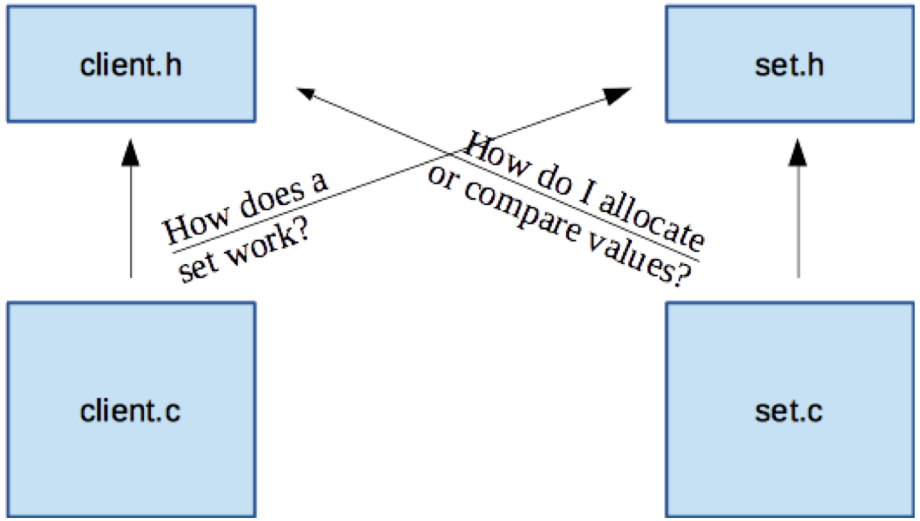


- Now, we'll use a generic pointer to each value, stored elsewhere in memory.



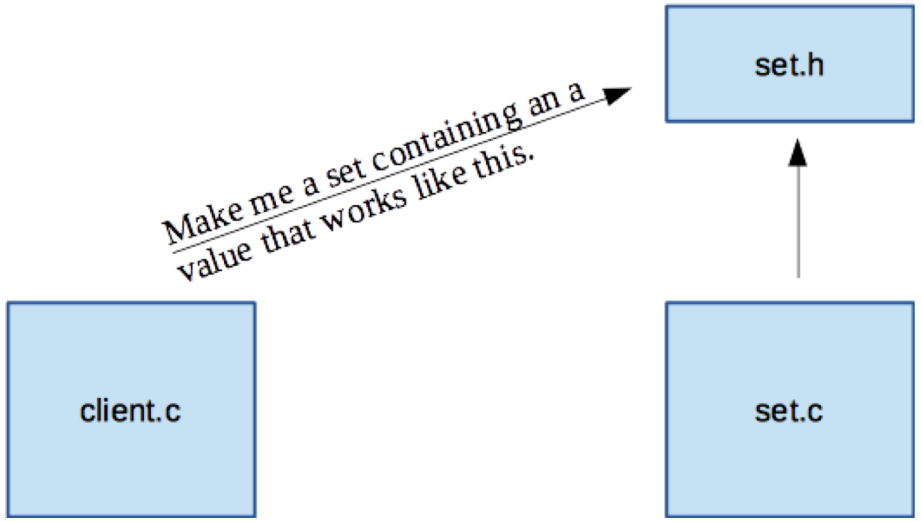
Circular Dependency

- How will our set know
  - How what value type it's supposed to contain?
  - Or how to compare values of that type?
- Could it just use code from the client for this?
- No. This is a terrible idea.
- It creates a *circular dependency* in our code.



Avoiding Circular Dependency

- Lots of design cleverness involves eliminating circular dependency.
- Our set needs to provide a mechanism for telling it about the value it's working with.
- Kind of like what qsort() does
- … it provides a way to tell it something about the type of values it's sorting.



Generic Implementation

```
#include "set.h"

struct Node {
  void *val;
  struct Node *next;
};

struct SetStruct {
  size_t vsize;
  bool (*same)(void const *v1, void const *v2);

  struct Node *head;
};

Set *makeSet(size_t vsize, bool (*same)(void const *v1, void const *v2))
{
  Set *s = (Set *) malloc( sizeof( Set ) ) ;
  s->head = NULL;

  s->vsize = vsize;
  s->same = same;

  return s;
}
```

set.c

I could hold any value.

I have to remember how large values are, and how to compare them.

You have to tell me once, when you make each set.

Client Code

```
#include "set.h"

bool sameInt( void const *v1, void const *v2 )
{
  return *(int const *)v1 == *(int const *)v2;
}

int main()
{
  Set *s = makeSet( sizeof( int ), sameInt );

  insertValue( s, (int []){ 2 } );
  insertValue( s, (int []){ 3 } );
  insertValue( s, (int []){ 5 } );
  insertValue( s, (int []){ 7 } );
  insertValue( s, (int []){ 11 } );

  for ( int i = 0; i <= 12; i++ )
    if ( containsValue( s, &i ) )
      ...

  destroySet( s );
}
```

Here's how to compare two ints.

Make a set and tell it how to work with ints.

We have to pass values by address now (using array syntax to get the address of just one literal int).