95-789

Lecture 8

Mid-Semester Review

Richard J. Orgass
Heinz School
Carnegie Mellon University

Carnegie Mellon

Class Topics Covered

- Week 1
  - Organizing, Code Warrior, compiling and executing
  - No questions on exam
- Week 2
  - elementary ideas
  - input/output
  - include files
  - control structures
  - data files
- Week 3
  - Functions
  - Random Numbers

Class Topics Covered -- 2 --

- Week 4
  - Recursion
  - Arrays
  - Copying structures
  - Quicksort, Treesort3
- Week 5
  - Pointers
  - Strings
  - Command line arguments
- Week 6
  - Classes
  - Storage Allocation

Tokens

- Five kinds of tokens
  - identifiers
  - keywords
  - literals
  - operators
  - other separators
- White Space ignored except as token separators
  - blanks
  - horizontal and vertical tabs
  - new lines
  - form feeds
  - comments

Comments

- Two styles
  - single line comments
  - multi-line comments
- Single line
  - // to end of line is a comment
- Multi-line
  - /* to next occurrence of */
  - Multi-line comments do not nest.

Identifiers

- Arbitrary long sequence of letters and digits
  - underscore (_) is a letter
  - First character must be a letter
  - Underscore (_) may not be first character
- Some C++ implementations use only the first 31 characters of identifiers, ignoring the rest. Not standard compliant

- "Spelling" conventions
  - names of
    - classes
    - functions
    - constants
    - enumerated types
  - start with upper case letter
  - all others with lower case letter

Reserved Keywords

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

Identifiers containing a double underscore (__) are reserved for use by C++ Implementations.

Identifiers starting with a single underscore (_) should be avoided.

Reserved Symbols

- Operators and punctuation (one character)

```
!   %   ^   &   *   (   )   -   +   =   {   }   |   ~
[   ]   \   ;   '   :   "   <   >   ?   ,   .   /
```

- Multi-character operators

```
->  ++  --  .*   ->*  <<  >>  <=  >=  ==  ~=  &&
||  *=  %=  +=   -=   <<= >>= &=  ^=  |=  :=  ::
```

- Preprocessor tokens

```
# ##
```

Standard Header Files

- From ANSI C
  - ranges of fundamental types, etc.
    - float.h
    - limits.h
    - stddef.h
  - Basic library functions
    - stdarg.h
    - stdlib.h
- C++ Specific
  - new.h
- Other libraries and headers are available
  - They do not determine the semantics of programs
    - stdio.h
    - iostream.h
    - string.h

Declarations and Definitions

- Declaration introduces names into a program
- Declaration is a definition unless
  - it declares a function without specifying the body
  - contains the extern specifier and no initializer or function body
  - it is the declaration of a static data member in a class declaration
  - it is a class name declaration
  - it is a typdef declaration
- Declarations can be repeated, definitions cannot

Declarations and Definitions -- II

- Examples of definitions

```
int a;
extern const c = 1;
int f(int x) {return x+a;}
struct S { int a; int b; };
enum { up, down };
```

- Examples of declarations

```
extern int a;
extern const c;
int f(int);
struct S;
typedef int Int;
```

C++ Types

- Two kinds of types
  - Fundamental
  - Derived

- Numerical limits for fundamental types
  - Include file limits.h
    - #include <limits.h>

Fundamental Types

- Character types
  - three different types
    - char
    - unsigned char
    - signed char
  - Value is integer representation of the ISO character code of the symbol

- Integer types
  - three different types
    - short int
    - int
    - long int
  - Can be signed or unsigned

Fundamental Types -- II

- Floating Types
  - Three types
    - float
    - double
    - long double
  - Must be signed
  - Characteristics in file float.h
    - #include <float.h>

- Integral Types
  - types char and int of all sizes
  - enumerations
- Arithmetic Types
  - integral types
  - floating types

Derived Types

- Arrays of objects of a given type
- Functions that take arguments of given types and have a given return value
- Pointers to objects or functions of a given type
- References to objects or functions of a given type
- Constants which are values of a given type

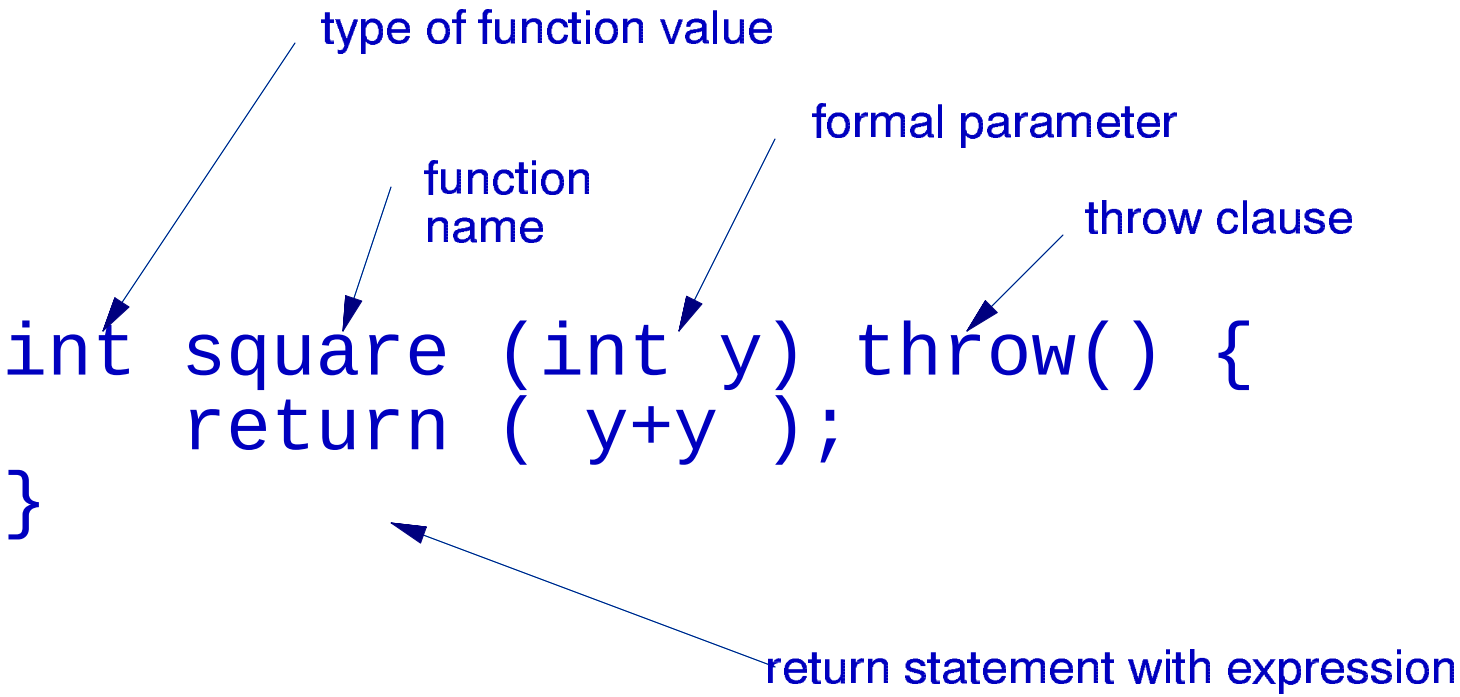
Derived Types -- II

- Classes containing
  - a sequence of objects of various types
  - a set of functions for manipulating these objects
  - set of restrictions on access to these objects and functions
- Structures
  - classes without default access restrictions
- Unions
- structures capable of containing objects of different types at different times.
  - **DO NOT USE!!**

Math Library Functions

```
ceil(x)
cos(x)
exp(x)       [exponential function, e**x]
fabs(x)
floor(x)
fmod(x, y)  [remander of x/y as flot, % for ints]
log(x)
log10(x)
pow(x, y)
sin(x)
sqrt(x)
tan(x)
```

Function Definition



Function call

```
int x;
int sqr;
sqr = square (x);
```

Function Example

```
#include <iostream.h>
int square ( int ) throw();
int main() {
    for ( int x = 1; x <= 10; x++ )
        cout << square ( x ) << "  ";
    cout << endl;
    return 0;
}

// Function Definition

int square ( int y ) throw() {
    return y * y;
}
```

Enumerated Types

```
enum Months { Jan = 1, Feb, Mar, Apr, May, Jun,

   Jul, Aug, Sep, Oct, Nov, Dec };
```

Usual first element of enum has value 0.
Assigning value to first element makes that the starting value.

Storage Classes

- auto
  - allocated on stack at block entry
  - released at block exit
  - no programmer action required
- register
  - Instruction to compiler suggesting that a register be used to store this value
  - Used before optimizing compilers did register assignments
  - Advice to compiler
- extern
  - data value or function is in another compilation unit
- static
  - value and location remain fixed throughout execution
- Note: volatile is a type qualifier not storage class

Iterative Factorial Calculation

```
int factorial (int i) throw() {
    answer = 1;
    for (int counter = i; counter >= 1;
            counter-- )
        answer *= counter;
    return answer;
}
```

Recursive Factorial Calculation

```
#include <iostream.h>
#include <iomanip.h>

unsigned long factorial (unsigned long) throw();

int main() {
    for (int i = 0; i <= 10; i--)
        cout << setw(2) << i << "! = " << factorial(i)
          << endl;
    return 0;
)

unsigned long factorial( unsigned long number) throw() {
    if (number <= 1)
        return 1;
    else
        return number*factorial(number-1);
}
```

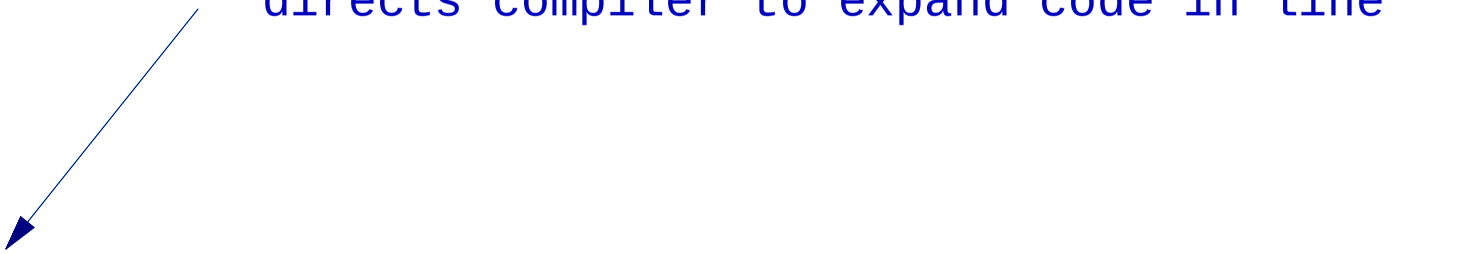
Fibonacci Numbers

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

Calculating Fibonacci Numbers

```
// Recursive Fibonacci function
#include <iostream.h>

long fibonacci(long) throw();

int main() throw() {
  long result, number;
  cout << "Enter an integer: ";
  cin >> number;
  result = fibonacci(number);
  cout << "Fibonaci(" << number << ") = " << result << endl;
  return 0;
}

long fibonacci(long n) throw() {
  if (n == 0) || (n == 1)
    return n;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Inline Functions




```
inline float cube(const float s) throw() {
    return s*s*s;
}
```

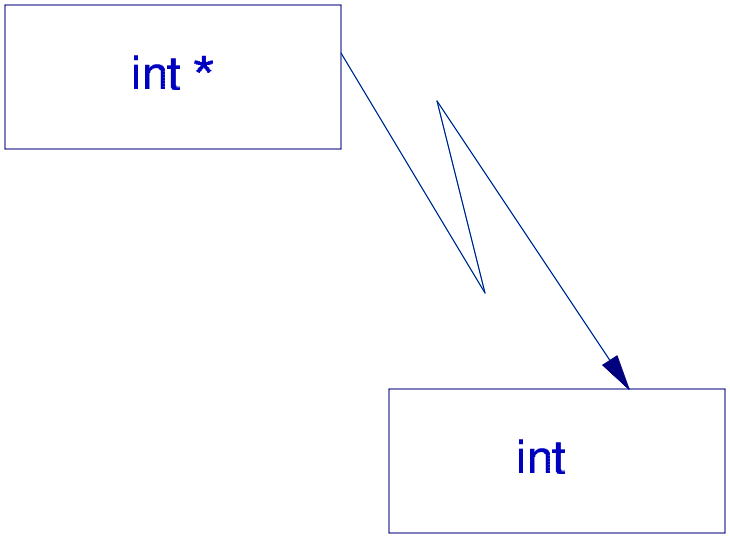
Default Arguments

```
int boxVolume(
  int length = 1, int width = 1, int height = 1) throw();
```

```
boxVolume();           // same as boxVolume(1, 1, 1)
boxVolume(5);          // same as boxVolume(5, 1, 1)
boxVolume(4, 8);        //same as boxVolume(4, 8, 1)
boxVolume(, 8, 12);     // same as boxVolume(1, 8, 12)
```

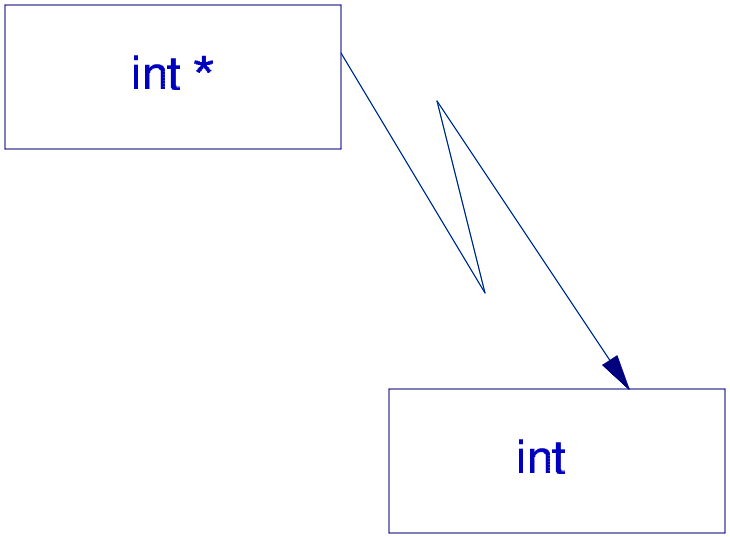
Pointer to Integer

- Pointer to Integer

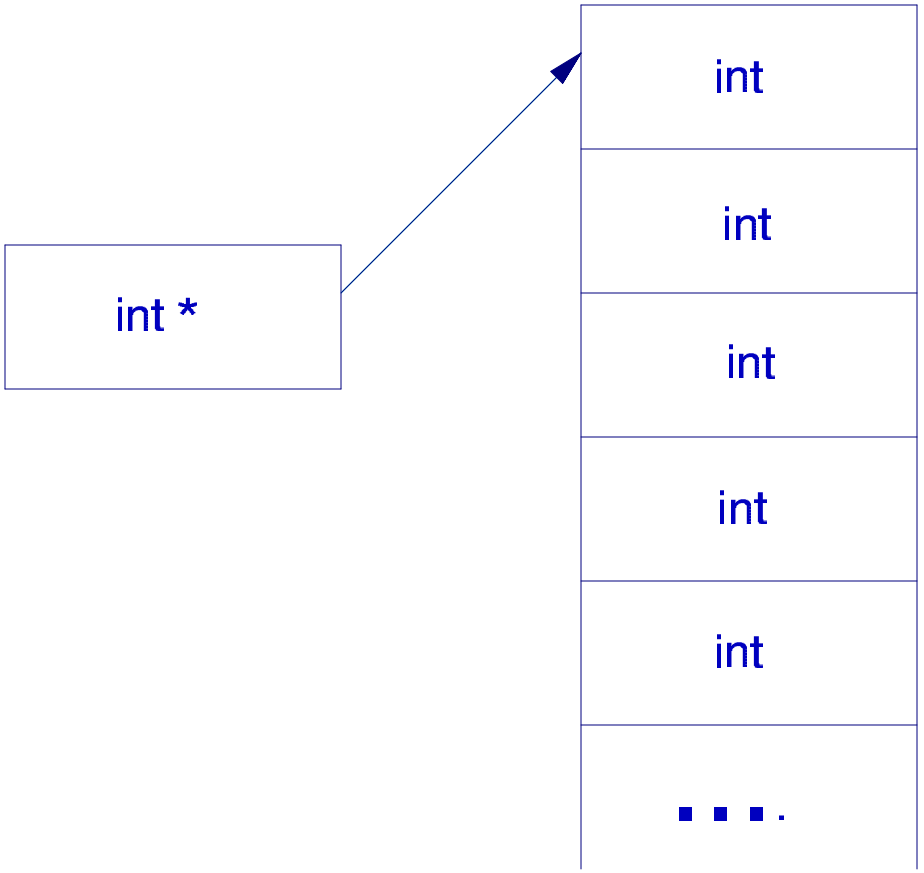


Pointer Ambiguity

- Pointer to Integer



- Integer Array



Strings

- Implemented as an array of char
  - name of array points to first element
    - See previous slide
  - no checking to see if string fits in allocated space
  - Example:

```
char c;
char *charPtr1 = &c;
char  charPtr2[10];
```

- How do you tell the difference between charPtr1 and CharPtr2?

String Functions in Library

- #include <string.h>
- char *strcpy(s, ct)
  - copy string ct into s and return s
- char *strncpy(s, ct)
  - copy at most n characters of string ct to s; return s.
  - pad with null characters if ct is too short
- char *strcat(s, ct)
  - concatenate string ct to the end of string s; return s
- char *strncat(s, ct, n)
  - same as strcat but for the first n characters; end terminate with null; return s
- int strcmp(cs, ct)
  - compare string cs to string ct; return < 0 if cs >ct or >0 if cs > ct
- int strncmp(cs, ct, n)

Reading Command Line Arguments

- int main(int argc, char **argv) throw();
  - the value of argc is the number of tokens on the command line
  - the first token on the command line is the program name
  - the array of strings argv is the tokens on the command line numbered from 0 to argc-1

Structs

```
struct Example {
                    int field1;
                    char field2;
                    float field3;
                    double field4;
                    unsigned field5;
                  }
struct Example *oneExample;
```

// Declares oneExample to be a pointer to an instance of struct Example.
// Storage for an instance of Example is NOT allocated.

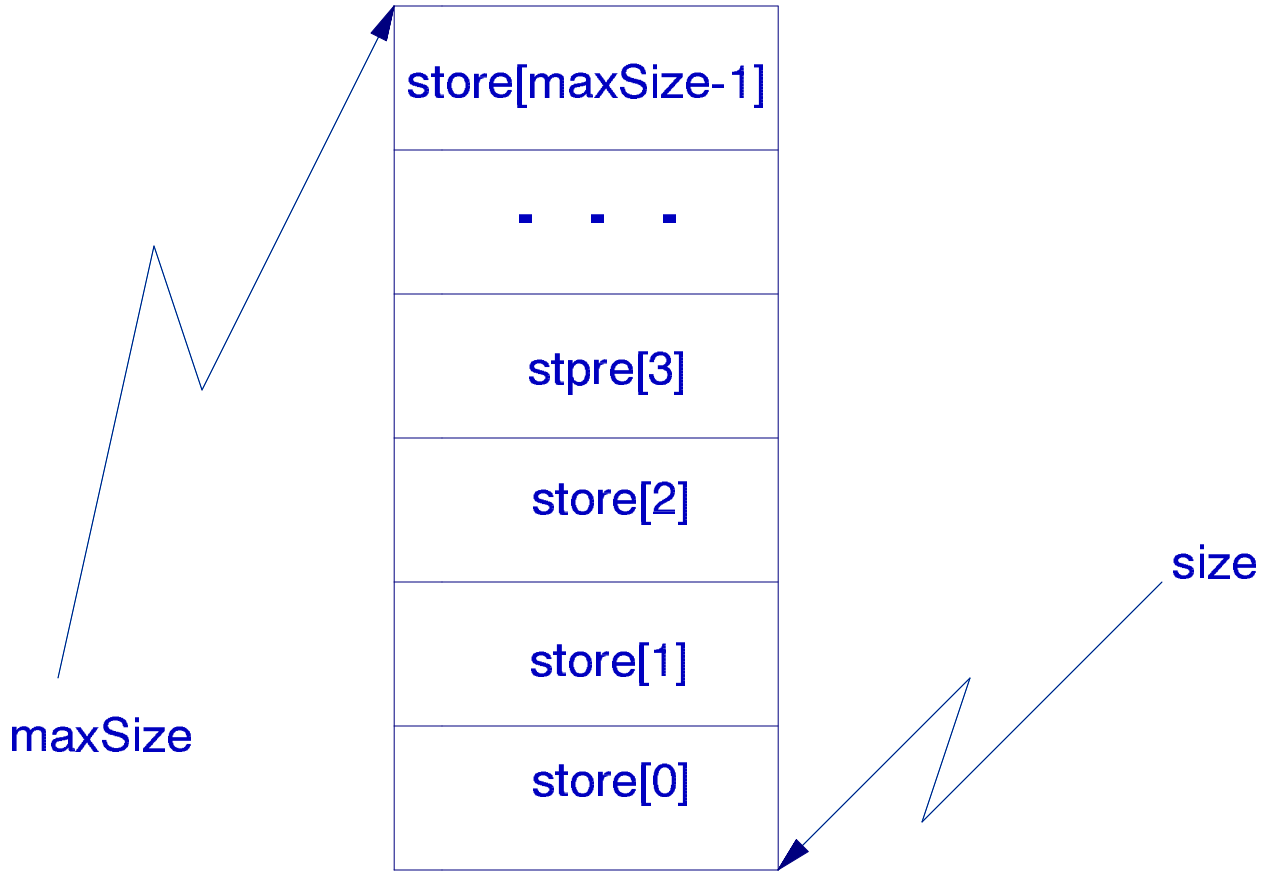
Structs -- 2 --

```
struct ExampleBody {
  int field1;
  float field2;
  char field3;
  char *field3;
}

typedef Example struct ExampleBody*;

// or

typedef struct ExampleBody {
  int field1;
  float field2;
  char field3;
  char *field3; } *Example;
```

Referencing Fields of Structs

```
struct ExampleBody {
  int field1;
  char field2;
  char field3;
}

struct ExampleBody foo;

// References to fields of Example Body

foo.field1;
foo.field2;
foo.field3;
```

Example: Stacks



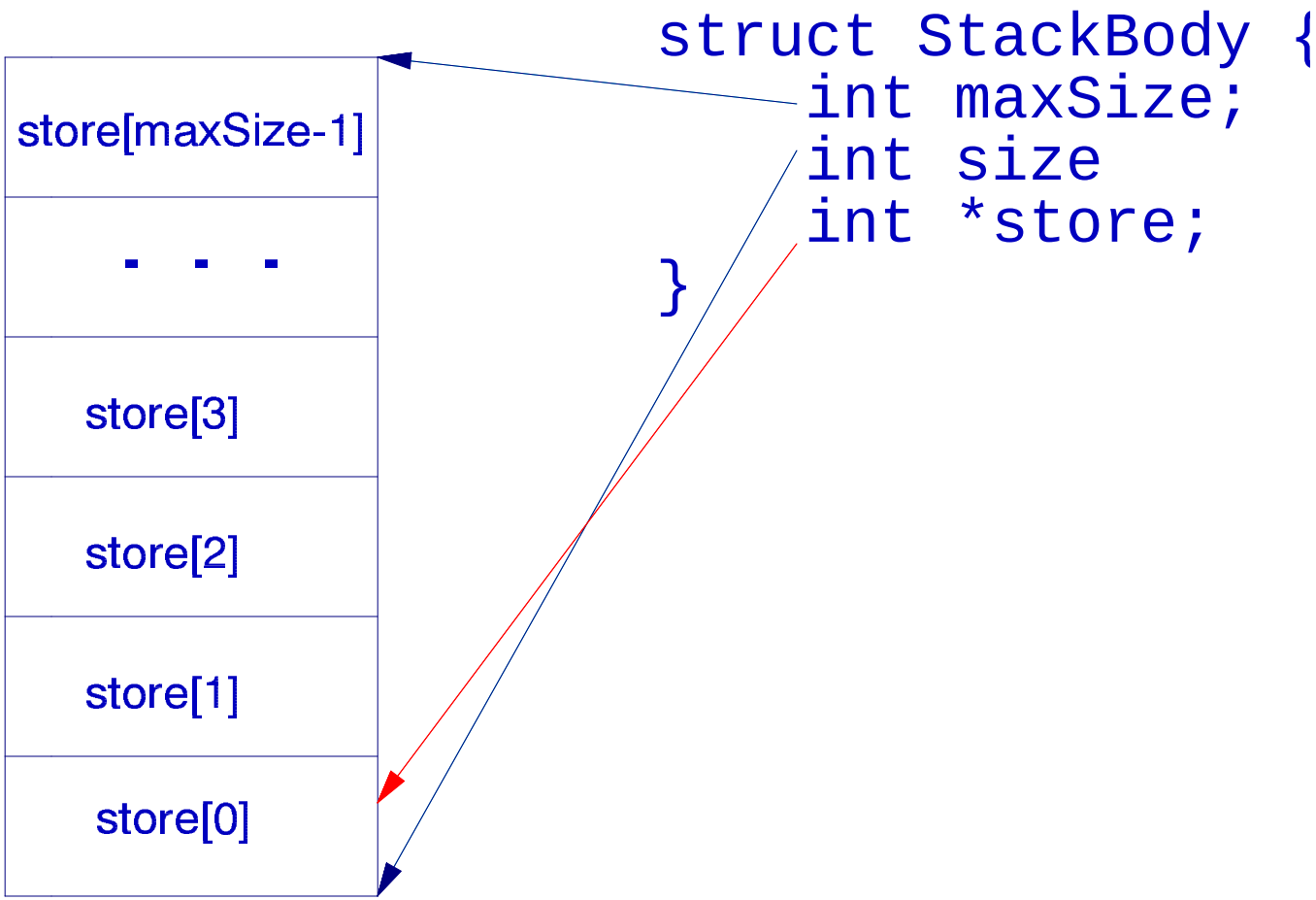
Fields of Structs via Pointers

```
struct ExampleBody {};
typedef struct ExampleBody *Example;

Example e = malloc(sizeof(struct ExampleBody));
// Allocate storage for an ExampleBody and
// set e to point to this storage.

// References are:

e->field1           (int)
e->field2           (char)
e->field3           (char *)
```

Stack Declaration (C)

```
// File Stack.h created by Orgass on 9/28/99

#ifndef _STACK_H_
#define _STACK_H_

// Declare Stack as a pointer type
typedef struct StackBody *Stack;

Stack new(int size);
// Create a new stack and return a pointer to the stack

void *dispose(Stack s);
// Destroy stack s and release all storage

void push(Stack s, int i);
// Push the integer i onto stack s, adding it to the
// top of the stack

int pop(Stack s);
// Remove the top element from stack s and return it
```

Example: Stacks



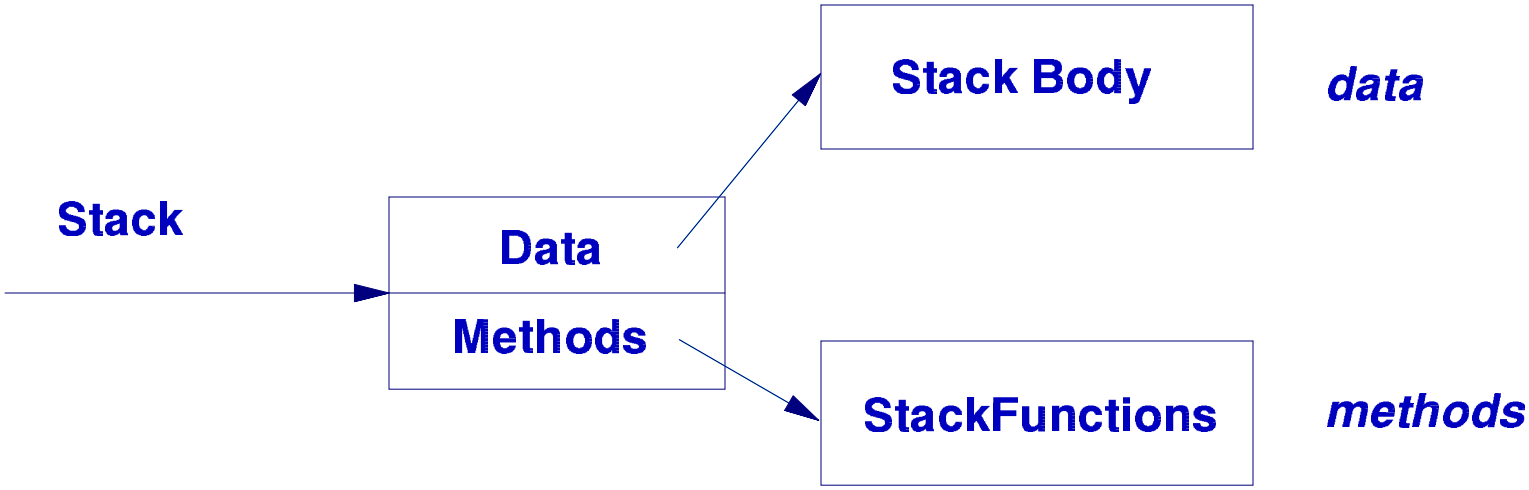
A struct of Functions

```
typedef void (*Push)(Stack, int);
typedef int (*Pop)(Stack);
typedef bool (*Full)(Stack);
typedef bool (*Empty)(Stack);

struct StackFunctions {
  Push push_;
  Pop pop_;
  Full full_;
  Empty empty_;
}
```

A struct of Functions -- 2 --

```
// Now initialize such a struct

#include <Stack.h>

struct StackFunctions methodBody;

methodBody.push_ = push;
methodBody.pop_ = pop;
methodBody.full_ = full;
methodBody.empty_ = empty;
```

New View of a Stack



Simplified view of C++ object implementation