Lecture Outline - C coverage

Reading segments

- *First reading is prefaces, Introduction 1, 2, 3, especially 4.*
- *Second is 5 & 6, skim 6.5 - 6.9*
- *Third is 7.*

Introduction

Standard C has backwards compatibility with “classic” or K&R C - causes some problems

- *Classic C does not use function prototypes, and function declarations do not need to include argument type information*
  - prototype idea was developed in early C++, applied to C
  - int foo(double x)
  - int foo(x)
    double x;
    { body }
- *the int type is default if a type is undeclared*
- *calling an undeclared function can produce difficult to find errors*
  - sqrt example -
- *recommendation - set your C compiler options to be strict C99 and require function prototypes*
  - You should use gcc 5.1.0 (available on CAEN lab machines)
  - -std=c99 -pedantic-errors -Wmissing-prototypes -Wall
  - But we will be using C89, because it is closest to K&R but with two valuable options from C99:
    - // comments are allowed
    - you can put a declaration can be anywhere in a block, not just at the beginning of a block
    - no other C99 features beyond C89 should be used in this course.

Relation to C++

- *no classes, no member variables or functions, no i/o operators (all library i/o done through function calls).*
- *Assumes procedural programming paradigm, but object-based programming done by careful programming, using struct types and functions that have a pointer argument to a struct type variable or allocated memory; heavy reliance on programming by convention*

Several key concepts to cover - make sure they are clear

- *basics of writing lines of code itself should be familiar.*

Headers, prototypes, and the linker

Intro example: scary C

- *<beginning of file>*
  int main(void)
  {
  int i = 2;
  double x;
  x = sqrt(i);
  / what is the value of x? */*
  }

- *will it compile?*
- *will it link (build an executable)?*
- *will it execute without crashing?*
- *will it produce a correct result?*

What's in a header file?

- *mostly function prototypes - in C*
- *double sqrt(double);*

function prototypes

Why supplied?

- Makes it easier to get the code right!

How does the compiler use a function prototype?

- It tells the compiler what conversions to make when pushing arguments onto the function call stack, and where to find the returned value (if any) and what conversions might be needed there.

Basic Syntax concepts

Declaration vs definition

- *roughly, declaration tells the compiler information about a symbol or identifier, and definition asks the compiler to commit memory space or its contents.*
- *A definition can also serve as a declaration if there is no previous declaration. If there was a previous declaration, then the declaration and definition have to agree with each other.*

clear with functions

- prototype is a declaration - tells the compiler how to set up the function call
- actual function code (includes the body) is the definition - tells the compiler to make arrangements to set aside the memory space, and what to put in it.

clear with struct type in C

- the structure declaration just says how to lay out memory space, but no actual memory space is set aside.
  - that doesn't happen until a variable is defined of that type
- *sometimes an issue with variables*
  - e.g. extern declaration versus the "defining declaration"
- *often, a declaration and a definition appear at the same point*
  - most variable declarations are also definitions.
  - can define and declare a function at the same point.
    - but we will often declare a function separately from defining it
- *in C89, must declare variables at beginnings of blocks only { }*
  - C99 allows them anywhere, like C++

- **Scope**
  - *where in the code does an identifier have a certain meaning?*
    - an identifier refers to a piece of memory - in what context does that identifier refer to a specific piece of memory?
  - *file scope*
    - translation unit = all headers as #included + the code in the .c file - this is what is actually compiled
    - file scope from the point of declaration through the rest of the translation unit
      - variables and functions at file scope have external linkage by default.
  - *global scope - if external linkage is used properly*
    - known throughout the program - all separately compiled modules can refer to it.
    - function names by default have global scope - external linkage.
    - "global variables" are normally given external linkage, though can be given just file scope also,
  - *block scope*
    - { } - curly braces define a scope, variables can declared at the beginning of any block.
      - use to reduce scope of variables in a complex function
    - why is returning a pointer (or reference) to a local variable always a mistake?
  - *function scope - within a function*
    - function parameters are in the scope of the first function block

- *struct (or class in C++) scope*
  - a name in a struct (or class) declaration is different from the same name outside
  - {
    int x = 1;
    int y = 2;
    struct Point {int x; int y;} p;
    p.x = 3; /* the x and y are different variables than the previous x and y */
    p.y = 4;
    }
- *hiding - if same names, the compiler uses the definition in the closest, innermost, scope*
  - A common rule - name in the inner scope "hides" or "shadows" name in an outer scope.
  - happens automatically and silently, so duplicating variable names inside an enclosing scope is a bad idea
  - one reason why global variables should at least have a unique and distinctive name - make them harder to hide by accident
  - example

```
#include <stdio.h>

void foo(int x);
int x = 1;

int main(void)
{
        printf("%d\n", x);
        foo(x + 1);
        printf("%d\n", x);
}
void foo(int x)
{
        printf("%d\n", x);
        while(1) {
                int x = 3;
                printf("%d\n", x);
                break;
                }
        printf("%d\n", x);
}

/* what does the following do?
void foo(int x)
{
        printf("%d\n", x);
        while(x <= 3) {
                int x = 4;
                printf("%d\n", x);
                }
        printf("%d\n", x);
}
*/
```

- **Lifetime**
  - *When does the memory space for a variable exist - when is it set aside or reserved, and when does it go away or become free for other use?*
  - *"Automatic", stack, function-local lifetime*
    - variables declared in a function block
      - memory set aside on the function call stack when the block is entered; freed up when block is exited
      - refering to it afterwards is undefined - space could have been recycled for some other data
      - Note: function parameter variables are in the top-level block in the function
        - only difference from local variables is where they get their initial values
  - *static lifetime*
    - static local variables, global variables
    - special static area of memory set aside and initialized when program starts
    - not part of regular function call stack, so contents are preserved regardless of function calls
    - when program terminates, static memory space is freed up
  - *dynamically allocated lifetime*
    - programmer controls lifetime by allocating/deallocated memory space explicitly in code
      - using malloc/free or new/delete
    - memory space stays reserved as long as programmer wants it to; recylcled when programmer says it can be
    - normally all recovered when program terminates, but common custom to "clean up" after yourself.

Types

why types?

- *tell compiler how much space it requires*
- *tell compiler what can be done with it*
- *seems trivial, but actually vastly important - some types can be declared incompletely - important - used in project 1*
  - because pointers to "objects" are always of the same size no matter what they point to, you can declare or define a pointer without having to say anything about the kind of thing it points to.
    - you just can't dereference it, or access data with it, until you do.
- *allows decoupling - can have a variable without compiler yet knowing everything about it.*

Built-in types

- *built into the C language - corresponds generally to what the CPU hardware works with directly*
  - integer numbers - short, int, long, also includes char - basically a one-byte number
  - floating point numbers - float, double
  - addresses - pointers - regardless of what they point to
  - sizes of them (number of bits) depends on the hardware being compiled for
    - Standard C sets only very rough restrictions or meanings on them.

sizeof operator - a compile-time operator that evaluates to the number of bytes occupied by a type

- *sizeof(int) typically 4*
- *sizeof(char) 1, **by definition** - should't ever sav sizeof(char) for this reason*
- *sizeof(x) is synonymous with sizeof whatever type x has.*
- *sizeof(expression) is synonymous with sizeof whatever type the expression has*
  - e.g. if x is a double, and y is an int, then sizeof(x+y) == sizeof(double)
- *sizeof is compile time - can never produce a result based on information present only at run time.*
  - char * str;
    str = foo(); /* str now points to a C-string */
    len = sizeof(str); /* will ALWAYS be sizeof(char *) typically 4! - contents of the C-string are irrelevant!

integer types can be signed or unsigned

- *In this course never declare an integer to be unsigned explicitly.*
- *Unsigned ints are dangerous -*

- unsigned int i = 5, j = 3;
- (i - j) /* what you expect */
- (j - i) /* what is this? */

▼ *Unsigned ints do nothing to make the code more expressive:*

- e.g.
  unsigned i;
  for(i = 0; i < n; i++)
- doesn't matter that i was unsigned! But if you do arithmetic on i you may be sorry!

▼ *Traditional use of unsigned ints - to express sizes which have to be positive, so the sign bit is used to double the maximum possible value*

- Very important on 16 bit machines!

▼ *Standard Library has a typedef for an integer type best suited to express sizes*

- size_t
- many functions that take or return sizes do so with size_t
- is often unsigned, but don't bet on exactly what it is.

▼ *In this course, to interface with Standard Library, either declare size_t variables or cast to/from ordinary (signed) int type. Use extreme caution with size_t because it might be unsigned.*

▼ tricky:

- size_t len = strlen(s);
  x = len - max; /* WATCH OUT */

▼ safer

- int len = (int) strlen(s);
  x = len - max;

▼ **in C, the char type is actually a kind of int - a one-byte int**

▼ *can freely convert between numerical types and chars .*

- char c1, c2;
  c1 = 'a';
  c2 = 42;
  c1 = c1 + c2;
  int diff = c1 - c2;

▼ *character codes were assigned so that the difference between two characters corresponds to sorting order - alphabetical order*

▼ c1 - c2 will be:

- negative if c1 comes before c2
- 0 if they are the same character (and the same case)

- positive if c1 comes after c2

- The reason why strcmp returns negative, 0, positive - chars are subtracted and difference compared to zero, starting at the beginning of the string and stopping if the difference is non-zero and returning it.

float & double types

floating point is approximate - numerical computation can be tricky

- e.g. variance calculation using two mathematically equivalent formulas
- see courses about this
- articles on the web about floating point - quite complex

- *no reason to use float unless space is at a premium - not enough bits for accurate numeric work - double is enough for most, but not all uses*
- *C89 math library assumes doubles for arguments and returned values - compiler autoatically converts floats to doulbles in function calls*

when can you compare two floating point values for equality?

only if nothing possible to right of decimal point.

- integers (within range) are always represented exactly, but only if assigned from an integer value - don't assume an integer value will result from non-integer calculations!

any other case is probably going to fail - trust nothing

- my trauma from Intel's extended precision register

- Sample code - try it and see what you get ...

```
/*
Demonstration that you can't compare floating-point (float or double) values and expect them
to be equal, even if they are equal mathematically.
*/
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
	float x = 1. / 1000.; // approximately one-thousandth
	float y = x * 1000000.; // should be one thousand, right?

	if(y == 1000)
		cout << "y == 1000" << endl;
	else
		cout << "y != 1000" << endl;

	cout << "y = " << setprecision(12) << y << endl;

	return 0;
}

/* example output - will depend on compiler, library, and machine
```

```
y != 1000
y = 1000.00006104

*/
```

- **enumerated types**
  - *A convenience facility - allows you to give names to a certain set of integer values, specifically listed or "enumerated"*
    - enum Colors_e {RED, YELLOW, BLUE, GREEN};
      - RED will be 0, YELLOW 1, etc.
      - integer values assigned automatically by the compiler, starting with 0, adding 1 to the previous to get the next, so you don't have to worry about them being consistent.
      - you can override if you want:
        - enum Colors {RED, YELLOW = 42, BLUE, GREEN = 6};
          - RED is 0, YELLOW is 42, BLUE is 43, GREEN is 6
    - Can use this instead of numerical values to make code more readable and expressive.
    - Name of the type is "enum Colors_e" - use to declare a variable of that type
    - can then use enumerated values to write code that expresses more clearly than raw integers would
      - enum Colors_e thing_color = RED;
      - if(thing_color == BLUE) { }
      - switch(thing_color) {
        - case RED:
        - case BLUE:
  - *C doesn't enforce that an enum is a different type at all, but will let you mix integer and enums freely, even to the extent of making a mess of the whole idea.*
    - thing_color = 99; /* what? */
  - *Don't do arithmetic to determine an enum value - this undermines the purpose of the enum type and results in obscure and error-prone code. Often a switch statement is a good choice.*
    - Bad:
      ```
      State_e new_state = old_state + 1;
      // what does this mean? Can it go out of range?
      ```
    - Good:
      ```
      State_e new_state;
      switch(old_state) { // obvious what is happening here
           case START:
                new_state = INITIALIZE;
                break;
           case INITIALIZE:
                new_state = OPEN_CONNECTION;
      ```

```
            break;
        case OPEN_CONNECTION:
            new_state = START_TRANSMISSION;
            break;
    // etc
```

- *I/O with enums: since enums aren't a first-class type, it gets treated as an integer. So it gets written out as the integer value, not its symbolic name. To read in an enum value, read the integer, check that it is a possible value for the enum type (clumsy to do) and then assign it to a variable of the enum type with a cast.*

The preprocessor and macros

macros

- *#define N 23*
- *BUILDS SYMBOL TABLE string "N" defined as string "23"*
- *everywhere the token “N” appears, "23" is substituted*
- *double-quoted strings are ignored - won’t go into strings - inconvenient in places, but only sensible thing to do.*

*#define SQUARE(x) (x) * (x)*

- why the parens? consider SQUARE(z+1)

macro pros

saves typing

- #define MAX(A, B) ((A) > (B)) ? (A) : (B)

results of macro expansion are inline, faster than a function call

macro cons

- can result in incomprehensible code
- can essentially define your own computer language
- can be cranky, since text substitution is dumb -knows nothing about types
- also, macro definitions can be hidden in includes of includes, meaning that reader can be confronted with one whose source is unknown

preprocessor is very important in C

- *often see macros defined*

but is heavily deprecated (== official insult) in C++

- *macros are not type safe*
- *macros interfere with comprehensibility*
- *inline functions will be just as fast - define instead*
- *function templates are much more flexible and type-safe*
- *using macros unnecessarly in C++ is a sign of bad taste, bad style, ignorance, etc.*

using preprocessor macros for conditional compilation

- *e.g. different compilers/architectures*

- *common in library implementations*

▼ **using macros as "include guards"**

- *still heavily used in C++*
- *#ifndef/#define/#endif*

▼ **assert macro - example of handy use of macro where rewriting your code makes some sense**

▼ *usage:*

- defined in <assert.h>
  #include <assert.h>

▼ assert(expression that is supposed to be true if everything ok);

- e.g. assert(i < n);

▼ if expression is false, program halts, see message on stderr/cerrr like

- assertion i < n failed in zot.c line 23

- excellent programming practice to sprinkle your code with asserts to catch programming errors quickly and clearly - e.g. before a bad pointer causes an uninformative crash.
- define NDEBUG before the #include to turn off - no code results - assert code literally disappears
  #define NDEBUG
  #include <assert.h>

▼ so leave asserts in during development; when debugging complete, turn them off to eliminate overhead.

- NOTE: Don't use for run time errors (like out of memory or bad input data) - NO PROTECTION IF TURNED OFF for deployment! Programming errors/debugging only!

▼ *a stripped down definition for illustration - always use the supplied one, in <assert.h> or <cassert> because there is some implement specific stuff involved.*

- #ifdef NDEBUG

  #define assert(condition) ((void) 0)

  #else
  #define assert(condition) ((condition) ? ((void) 0) :
  __assertion_failed(#condition,__FILE__, __LINE__))

  #endif

- note the leading-underscore names: part of the implemetation; other compiler/libraries will differ. However, the file and line number symbols, which begin and end with double underscores are Standard - you can use them yourself if you want. They are replaced during preprocessing by the file name and the line number where they appaer.

▼ **A case of a messed-up macro - a good example of why they are not a good idea**

- *sometimes macros used instead of a function for speed -*

- *the code is in-line instead of a function call, but called as if a function*
- *example*
  - while(*startp && !(isalpha(*startp))) {
  - was translated as:
  - while (*startp && ! ((__ctype + 1)[*startp] & (0x00000001 | 0x00000002)))
  - then gave warning that subscript was a char, not an int!
- *but how could I avoid this? isalpha is *supposed* to be call with a char or an int - it's not my fault if the implementers did this*
  - my claim is that the macro is incorrect - should have cast the subscript
- *- so macro differs from a function - a function can hide the details of the implementation from the caller, while a macro can't*
- *why macros are hated in C++ they too easily interfere with program organization and clarity.*

Some Small but Important Issues

- **Order of evaluation in an expression is undefined unless precedence, associativity require it.**
  - *can't depend on whether f or g will be called first - code must not be sensitive to it.*
    - x = f(a) + g(b);
    - foo(f(a), g(b));
  - *what values of i are given to foo?*
    - i = 3;
    - foo(i++, i);
    - could be either 3 and 3 or 3 and 4 - can't depend on it.
  - *special rule for increment/decrement operators:*
    - never use both the variable and the incremented/decremented variable in the same expression or statement.
      - stay out of trouble!
- **Naming conventions**
  - *See the handout: C Coding Standards*
  - *Don't start names with underscores*
    - Such names are reserved for implementation symbols - too easy to screw up and collide with one!
      - I used to think I was being paranoid until a student had a very confusing error due to exactly this problem!
    - Exact rule: first character is an underscore followed by upper case letter, or a second underscore, is reserved for the implementation.
    - Very hard to tell exactly how many underscores you have, so best not to use any leading underscores. Not allowed in this course, though you see some people who like to live dangerously.
  - *Macro names should be all upper case*
    - almost universal convention, so follow it!
  - *Common for typedef names to end with "_t" - I like to use this*
    - e.g. size_t is in standard library
  - *Very common to use initial uppercase of user-defined types or typedef names - you should do this*
    - struct Thing, Record_t
    - note C, C++ std. lib is almost all lower-case names, so makes it easy to tell your types from library types.

- *Basic names: either upperlower or underscore to divide names into words*
  - FindMaximumValue
  - find_maximum_value /* my preference */
- *Follow guru advice - make the names long enough to be comprehensible*
  - if you have to write or read a comment to tell what a variable is for, try a longer, more descriptive name
  - names used purely locally can be single letter
    - i for for loop array subscripts
    - p for a temporary pointer

- **Function parameters**
  - *A function with no arguments must be declared as (void) - in C, empty parentheses mean something different than in C++ - backwards compatible with "Classic" C - no argument list type checking!*
    - In C:
      - void foo(void); /* foo takes no arguments and returns nothing
      - void foo(); /* foo might take arguments as in pre-Standard C but compiler will not check them.
    - In C++
      - void foo(); /* takes no arguments and returns nothing and Compiler checks for this! */
  - *Parameter names in function prototypes are optional and are ignored if present*
    - use meaningful names for documentation purposes
- **Standard header files for C always end in .h**
  - *<stdio.h>*
- **use of goto**
  - *why banned - almost*
    - spaghetti code
  - *better ideas which make goto easy to avoid*
    - if-else with blocks
    - iteration constructs: for, while, do-while - uses blocks
    - use of functions as a code organization technique

Linkage

Overview: source file, preprocessing, translation unit, compilation, object file, linker, executable

- *See separate notes - Multiple Sources, Linker, global variables*

internal, external

- *"static" keyword*

the One Definition Rule

- *error if the linker finds more than one equally applicable definition*
- *error if the linker fails to find a definition for something that is referred to*
  - no definition of an external variable
  - no definition of a function that is called
    - normally no problem if the function is never called
    - most linkers will reduce the size of the executable by stripping out code that is never called.

Pointers, Arrays, Function pointers, structures

struct type

- *what is it - declared layout of memory, defined block of memory*
- *the "user-defined type" in C*
  - where "user" means "user of the language" - the programmer
  - the *only* truly user-defined type - everything else is basically built-in.
- *In C, name of user defined type always starts with struct*
  - following declares a type named "struct Point"
    - struct Point {
      double x;
      double y;
      };
  - why the ending ';'? You can declare a variable of that type in the same statement.
    - struct Point {
      double x;
      double y;
      } p1, p2;
    - can declare a structure type anywhere, but most normally done at file scope or in a heade file, and then used in definitions later
  - can use a typedef to give it a one-word name
    - typedef struct Point Point_t;
- *compiler must lay them out is same order as declared, (lowest to highest addresses) but not necessarily contiguous - often filler bytes*
- *note alignment issues!!*
  - compiler must ensure that all variables start on proper address boundaries for their type - depends on CPU architecture, but typically:
    - char can be anywhere
    - int must be a multiple of e.g. 4 bytes
    - double must be a multiple of e.g. 8 bytes
- *example:*
  struct S {
  char c;
  int i;
  double d;
  };
  - on one compiler/architecture this takes 16 bytes, not 1+4+8 = 13

- sizeof(struct S) usually greater than sizeof(char) + sizeof(int) + sizeof(double)
  - sometimes a lot!! more
- *NEVER guess at the size of a struct type - ALWAYS use sizeof operator!!*
  - note that sizeof(char) is one, by definition - using sizeof is redundant
- **Typedef - how does it work?**
  - typedef <already known type> <new synonym name for already known type>
  - doesn't really create anything, just gives a synonymous name for already known types - compilers often just substitute it in, and then ignore it thereafter - can be inconvenient
  - But normally affects the syntax parsing - the typedef'd name is treated as a unit.
  - means that complex declarations can be a lot easier to get right if built up out of typedefs

- **Pointers**
  - *"pointer" - an address, or a variable containing an address*
    - int * ptr = 0;
    - int i = *ptr;
    - *ptr = 42;
  - *by definition, a "null" pointer - an address represented as 0 (zero), points to nothing at all*
    - by custom and convention, a null pointer is used to mean an "empty" pointer - "nothing there"
    - trying to derefence a zero/null pointer, or make use of the value at a 0 address, is always an error
      - what the error results in depends ... might not crash - depends on the machine/OS, etc.
      - most machines use address zero for some special purpose, and may get confused or upset if you try to store something there.
  - *the macro NULL is customarily used in C to represent a pointer value of zero*
    - #define NULL 0 /* a common standard definition in both C and C++ */
    - #define NULL (void*) 0 /* another definition that attempts to distinguish between integer zero and the zero address
    - but in fact, 0 is just as good as NULL - means exactly the same thing - NULL just contributes some clarity
      - in C, using NULL is a long honored custom.
    - but in C++, since macros are avoided, convention is NOT to use NULL
      - I will apply this in this course - show you are tuned in by using NULL only in C, not in C++
      - we don't like to use NULL instead of 0 in C++, and actually have a much better choice starting with C++11.
  - *What about pointers to different types of data?*
    - We talk about “representation” of the address - what is the format in terms of bits?
    - on most modern machines, all kinds of data (and code) in memory resides at addresses that have the same representation - e.g. 32 bit positive binary numbers.
    - in olden times, some machine architectures had addresses that had a different representation for different kinds of data in memory
      - e.g. word addressed machines had a special address representation for accessing individual characters, or for code in memory versus data.
      - a result of trying to optimize the hardware cost - every bit more in the data pathway cost a lot of money before very large scale integrated circuits
    - C is committed to being compatible with these architectures - so a C compiler for that architecture has to know how to transform pointers from one representation to another.

- C developed in the context of early byte-address machines, so it is most “comfortable” with that family of architectures.
- e.g. the null pointer might not actually be all bits zero, so if you set a pointer to zero, the compiler has to know how the null pointer looks for that architecture.
- on most modern machines, the null pointer really is all bits zero.

▼ *what does the type of a pointer mean?*

- so why do pointers have "types"?

▼ in certain cases, the compiler must know what is at the address -

- *ptr - dererence gives you a thing of the pointed-to type - what is it? the pointer type says.
- doing pointer arithmetic - compiler needs to know the size of the pointed-to thing

▼ *pointers to const data*

▼ basic use of const - to say that a variable should not be modified - compiler will flag it as an error if your code tries to do so:

- const int x = 5;
  /* other code */
  x = 6; /* error! */

- pointer to const means pointer can't be used to modify the pointed-to data:
  int x = 5;
  int * p = &x; /* pointer to non-const int */
  const int * cp = &x; /* pointer to const int */

  p = 6; / OK */
  cp = 7; / error! */

▼ const in C is not quite the same as const in C++

▼ consider const int x = 42;

▼ in C++, the compiler "knows" x is a constant and what its value is at compile time

▼ can thus use it anywhere you could use a literal constant

- char ary[x]; // same as char ary[42]
- switch (i) {
  case x: // same as case 42:

▼ in C, x is still a variable, just one that is marked as read-only after initialization, but the compiler doesn't "know" that it is actually a constant - it is still a variable!

- note gcc incorporates a bunch of non-standard extensions; use -pedantic-errors with -std=c89 to get actual c89 enforcement
- char ary[x]; // not same as char ary[42] - ary is a C99 variable-length array
- switch (i) {
  case x: // not the same as case 42: - treating x as constant is gcc extension

- This is why #define is still used for constants in C - since the preprocessor substitutes in the literal contant, compiler is happy with it.

- this is a "bug" in the C Standard - probably why gcc "corrected" it

- in C++, Stroustrup wanted to avoid using preprocessor mactors as much as possible, so made compiler aware that an const variable intiialized with a compile-time constant was in fact equivalent to a literal constant. New C++11 constexper goes even vurther.

▼ Because of limitations, const in C is not used very much compared to C++, but it can be useful to:

- make clear whether input parameter pointers are pointing to read-only data or not.

- make coding more bug-proof - compiler helps you avoid modifying stuff that isn't supposed to be modified

- Some use of this in Project 1

- Important concept: Don't say things are const that are not conceptually constant! Nothing but misery and kludgy code down that path!

▼ *pointer parameters*

- no reference parameter type

- how do you get more than one value back from a function?

▼ what if it is a pointer value you need back? Pointer to pointers

- looks scary, but draw picture, make up addresses, keep it straight

-
```
        int i = 1;
        int j = 2;
        int k= 3;
        int * kp = &k;
        printf("%d, %d, %d, %d\n", i, j, k, *kp);

        i = foo(&j, &kp);

        printf("%d, %d, %d, %d\n", i, j, k, *kp);

}

int foo(int * ap, int ** bp)
{
        *ap = 5;
        **bp = 7;
        *bp = ap;
        return 4;
}
```

▼ *pointer arithmetic*

- incrementing/decrementing pointers, and pointer arithmetic in general, usually only makes sense if you are pointing into an array

▼ +1 means point to the next thing of that type, -1 to the previous thing of that type

- actual no. of bytes added depends on the type
- char * pc;
- int * pi;
- pc++ means add 1 to address in pc, point to next char
- pi++ means add 4 (typically) to address in pi, point to next integer

▼ difference between two pointers is a signed integer value for the number of elements between the two pointers:

- char a[10]; int b[10]
  char * pc1 = &a[0]; char * pc2 = &a[2];
  pc2 - pc1 is 2 (two characters, actual difference in addresses is 2)
  pc1 - pc2 is -2
  int * pi1 = &b[0]; int * pi2 = &b[2];
  pi2 - pi1 is 2 (two ints, actual difference in addresses is 8 typically)
  pi1 - pi2 is -2

▼ actual type for difference between two pointers is defined as ptrdiff_t in <stddef.h> (same place as size_t is defined.

- could be long, or long long on 64 bit machines

▼ *void * type means pointer to uncommitted type*

- can't dereference it - compiler doesn't know what kind of thing is at that address!
- just a raw address
- introduced in Standard C to represent directly the idea of a raw address - no claim is being made about what is at that address.
- can convert to/from pointers of other types, but watch out!
- void * does have a set size - e.g. sizeof(void *) is usually 4 on 32-bit machines.

▼ can have pointers to void pointers, and can dereference them, do pointer arithmetic on them:

- void * a[5]; /* array of five void pointers */
  void ** p = &a[1]; /* p points to the void pointer in the second cell of the array */
  p++; /* now points to the void pointer in the thrid cell of the array */
  *p is the contents of the void pointer where p points - a void *
  **p - can't do - trying to dereference a void pointer
- in other words, a pointer to a void pointer is not a void pointer.

▼ Some small differences between C and C++

▼ in C, void * was made backwards compatible with how char * was the original raw address type in classic C

- on byte-addressed machines, a raw address was the same as a pointer to characters
- so sizeof(void) was made equal to sizeof(char) == 1 by definition

- so you can do pointer arithmetic on void *
  - void * p = some address
    p++ is that address plus one
- in C++, this was tightened up
  - sizeof(void) is undefined, so can't do pointer arithmetic on a void *

- *void pointers and casting - can go to/from void pretty freely (but note integers)*
  - **implicit** conversion - the compiler will let you do an assignment or copy of a value from one type to another
    - might produce a warning - either helpful or annoying.
    - Function parameters are copies of argument value in the call, so implicit conversions will be done in a function call following same rules as assignment
  - **explicit** conversion - you write a cast expression that specifies what type you want to convert a value to
    - you are "casting it in the role of ..."
  - Is a representation change involved?
    - On most modern machines, pointers have the same representation regardless of data type - just a plain address
    - In this case, all the cast does is tell the compiler to treat the address as a pointer to the other type.
    - But if a representation change is involved, the compiler may have to generate machine instructions to do it.
  - A demo of what's legal when converting to/from void pointers. Try this with your C compiler

```
#include <stdio.h>

int main(void)
{
        struct S1 {
                char c;
                int i;
                int j;
                } s1;

        struct S2 {
                char c;
                double d;
                } s2;

        void * void_ptr = 0;

        struct S1 * S1_ptr = &s1;
        struct S2 * S2_ptr = &s2;

        int i = 3;
        int * int_ptr = &i;
```

```
    double d = 3.14;
    double * double_ptr = &d;
    char c = 'x';
    char * char_ptr = &c;

/* Start with simple conversions. The compiler will let you convert one kind of object into
another if they have a known and well defined relationship - like numeric types.
So the following are legal "implicit" conversions. */

    i = s2.d;       /* but copies the integer part only, loses rest */
    s2.d = i;
    i = c;
    s2.d = s1.c;  /* chars are actually just a one-byte int in C */

/* But the compiler doesn't know the meanings of user-defined (struct) types,
so how could it meaningfully convert one into the other? Even the cast doesn't
make sense, so it is disallowed; the following are all illegal. */

    s1 = s2;

    s1 = (struct S1) s2;

    i = (int) s1;

/* The following are illegal for much of the same reason -
pointers are not interchangeable if they point to different kinds of objects. */

    S1_ptr = S2_ptr;

    S1_ptr = int_ptr;

    int_ptr = S1_ptr;

/* Even when the pointed-to objects have legal "implicit" conversions, this doesn't
mean that the pointers are implicitly convertible. Among other issues, note
that a double might have to reside at a certain set of addresses
due to alignment issues, that int's don't necessarily share. So all of the following
are illegal in Standard C. */

    int_ptr = double_ptr;
    double_ptr = int_ptr;
    int_ptr = char_ptr;

/* Likewise, integers and address are different kinds of things, so even though
addresses are integral values, the following are all illegal. */

    int_ptr = 360;

    i = int_ptr;

    void_ptr = 360;

/* HOWEVER, you can coerce the compiler to convert the pointer type with a cast, but that doesn't
mean that the result is meaningful - you are forcing the compiler to agree with your idea that the
pointed-to
data come be meaningfully interpreted as the other kind of object.  The following are all legal, but
it is up to you whether they are actually meaningful or correct. */
```

```
        int_ptr = (int *) 360;

        S1_ptr = (struct S1 *)S2_ptr;

        S1_ptr = (struct S1 *)int_ptr;

        void_ptr = (void *) 360;

/* void pointers in C are special because you can freely assign other pointers to and from
without the use of a cast  - but again, you better be right!  It is often a good idea to
include the cast anyway to let the reader know what you are doing more clearly. */

        void_ptr = int_ptr;

        S1_ptr = void_ptr;

        void_ptr = S1_ptr;

        int_ptr = void_ptr;

        return 0;
}
```

- **Incomplete type declarations - also called "forward" declarations**
  - *tell the compiler only that a certain type will be used*
    - struct Point;
    - structure declations is most common in C, class declarations in C++.
  - *then can declare pointers of that type, but can not dereference them or do pointer arithmetic with them because compiler doesn't know what it is or how big it is.*
    - likewise, cannot declare a variable of the incomplete type - compiler doesn't know how big it is, so can't set aside stack space for it.
    - pointers are allowed because pointers are always the same size, no matter what they point to, so compiler can deal with a pointer declaration to a incomplete type with no problem.
  - *then when necessary, provide the complete declaration to the compiler to allow full use of the type*
    - Examples:

```
struct Thing; /* we will be a using a Thing struct */

struct Thing * ptr;  /* ptr points to a Thing struct - whatever it is */

ptr = get_Thing();  /* we can work with a pointer to a Thing struct now */
print_Thing(ptr);    /* we can give it to another function now */

ptr->i = 13;  /* error - do not know yet that Things have an "i" member */

struct Thing t; /* error - do not know yet how big, or what is in, a Thing struct */

/* now later  in the code */

struct Thing {     /* the complete declaration */
        int i;
        char c;
        };

struct Thing t; /* no problem, t can be allocated now */

ptr = &t;

ptr->i = 13;  /* no problem, complete declaration is known */
```

  - *Using incomplete types in Project 1*
    - By putting an incomplete type declaration in the header file for a container (like Ordered_container), we allow the client code to use the containers (through a pointer), but without knowing what is *inside* the container. A simple, but powerful form of encapsulation. The implementation file (Ordered_array.c, Ordered_list.c) has the complete declaration required for the actual container code.
      - technique is also called an "opaque type"

- **Arrays (built-in arrays)**
  - *in C99, size of arrays can be defined at run time, but for Standard C89, arrays are given a size when declared and so the size is fixed at compile time.*
    - it is easy to simulate dynamically sized arrays on the heap.
  - *considered evil in C++ - too easy to make errors, too awkward - e.g. use std::vector<> or std::array<> instead, but built-in arrays are still there in C++, and are still important, especially with C-strings, arrays of characters.*
    - K&R say that name of array is same as address of first cell, but this is not quite true
  - *name of array carries overall size information, but only in the scope of the original declaration!*
    - example

```
int main(void)
{
        double ary[5];   /*array of 5 doubles occupying (typically) 40 bytes */
        printf("%d\n", sizeof(double));   /* outputs 8 (typically)*/
        printf("%d\n", sizeof(double *));   /* outputs 4 (typically) */
        printf("%d\n", sizeof(ary[0]));     /* outputs 8  */
        printf("%d\n", sizeof(&ary[0]));   /* outputs 4  */
        printf("%d\n", sizeof(ary));        /* outputs 40 */
        foo(ary);
}

void foo(double ary [ ] )     /* void foo (double * ary) is synonymous */
{
        printf("%d\n", sizeof(ary[0]));     /* outputs 8  */
        printf("%d\n", sizeof(&ary[0]));   /* outputs 4  */
        printf("%d\n", sizeof(ary));        /* outputs 4 ! */
}
```

  - *better way to put it: when array name is passed to a function, or used in an expression, the name of the array "decays" to a pointer of element type pointing to the first cell of the array*
    - ary becomes double * - the address of ary[0]
    - ary + 3 is equivalent to ary[3];
    - call: foo(ary); ditto
    - definition of [] operator
      - left hand side is pointer to something of type T
      - right hand side is an integer
      - calculate address lhs + rhs * sizeof(T)
      - dereference this address to obtain value there (or change value there)
    - left hand side can be array name, or a pointer variable - works the same
    - 2D arrays - ?

- *can use for run-time sized arrays*
  - int * a = malloc(n * sizeof(int));
  - a[0] is first
  - a[n-1] is last element
  - an array of pointers?
  - int ** a = malloc(n * sizeof(int *));
  - *a[0] is whatever the pointer in the first cell points to

- **C-strings**
  - *Called "C-strings" to distinguish from the C++ Std. Lib. string class.*
  - *only thing C compiler knows about "strings" are string literal constants*
    - when compiler sees "abc" it arranges for a, b, c, \0 to be stored in memory somewhere
    - everywhere "abc" appears, compiler replaces it with address where the array starts.
    - so the type of "abc" is char * or const char *.
  - *everything else concerning strings is in the C-string library functions and i/o functions*
    - lots of them - don't re-invent the wheel!
    - see the course web site handouts for a list of the most useful.
  - *idiomatic to use char * pointers to process C-strings rather than array notation*
    - array notation is a mathematical shorthand best for number arrays - you need to be comfortable with using pointers directly
  - *example - copy a C-string from a source to a destination - the "essence of C" - how does it work?*
    - void strcpy(char * dest, char * src)
      {
      while(*dest++ = *src++);
      }

Function pointers

- **the name of a function has type “function taking parameters of T1, T2, etc and returning T3”**
  - *can be treated as the address of where the function ends up in memory*
  - *can arrange to store a function’s address in a pointer variable, and then call the function using the pointer variable, or “function pointer”*
  - *but for compiler to generate the code for a call to the function using the function pointer, it needs to know what the return type and parameter types are - compare to function prototypes.*
    - do this by declaring the function pointer type.
- **syntax of a function pointer declaration:**
  - *return_type (*variable_name)(type1, type2 etc);*
  - *int (*fp)(int, double);*
  - *fp is a pointer to a function that returns an int and takes an int and a double as parameters.*
    - example

```
int foo(int);
int main(void)
{
        int i = 42;     /* to use in the call */
        /* declaration */
        int(*fp)(int);
        /* assignment */
        fp = &foo;
        fp = foo; /* implicit conversion */
        /* calling a function using a function pointer */
        i = (*fp)(i);
        i = fp(j); /* also permitted */
}
```

- **function pointer cast - declaration, no variable name, enclosed in parentheses**
  - *general cast syntax: the type name in parentheses - e.g. (int *) - do the same for function pointer cast*
    - (return_type (*)(type1, type2 etc))
- **function pointer typedef - the typedef name shows up as the variable name in the declaration**
  - *typedef return_type (*typedef_name)(type1, type2)*
  - *typedef_name fp; /* function pointer of that type */*
- **main use of a function pointer - to help create generic code**
  - *one part of code can decide what function should be called, and pass a pointer to it as a parameter*
  - *other part will use the function pointer to call whatever function was selected elsewhere*
  - *or, function pointers could be stored in an array, in a struct, wherever*

▼ *simple example of using function pointers at run time to vary behavior*

- /* Demonstration of function pointers in a trival context. */

```
#include <stdio.h>

/* the function pointer type */
typedef int (*int_getter_t)(int *);

/* function prototypes */
void do_the_work(int_getter_t);
int get_odd_int(int *);
int get_even_int(int *);


/* main asks the user which function to use, sets a pointer correspondingly,
and then hands it to the do_the_work function to use.
*/
int main(void)
{
        int_getter_t fp;       /* the function pointer */
        char c;

        while(1) {
                printf("Do you like odd numbers? Enter y, n, or anything else to quit: ");
                /* the whitespace in the format string says to skip leading whitespace and then read a
character */
                scanf(" %c", &c);

                if(c == 'y') {
        /*              fp = get_odd_int;           one syntax */
                        fp = &get_odd_int; /*       an equivalent syntax */
                        }
                else if(c == 'n') {
        /*              fp = get_even_int; */
                        fp = &get_even_int;
                        }
                else
                        break;

                /* call using the function pointer as an argument */
                do_the_work(fp);
                }

        printf("Done!\n");
        return 0;
}

/* this function "does the work" using a function passed as a parameter */
void do_the_work(int_getter_t fp)
{
        int i, result;

        /* call the function using the pointer */
/*      result = fp(&i);                one syntax */
        result = (*fp)(&i);    /*       an equivalent syntax */
```

```
        printf("You entered %d\n", i);
        if (result) {
                printf("You chose wisely!\n");
                }
        else {
                printf("You are inconsistent!\n");
                }
        return;
}

/* these functions ask the user to supply an integer;
they return the integer value using the pointer parameter,
and then return either 1 (true) or 0 (false) depending on
whether the number is odd or even versus even or odd.
*/

int get_odd_int(int * ip)
{
        printf("Enter an integer: ");
        scanf("%d", ip);
        if (*ip % 2)
                return 1;       /* true if odd */
        return 0;
}

int get_even_int(int * ip)
{
        printf("Enter an integer: ");
        scanf("%d", ip);
        if (*ip % 2)
                return 0;
        return 1;       /* true if even */
}
```

- **Casting between a function pointer and void***
  - *C has to provide for the possibility that addresses of data and addresses of machine code may have different representations.*
    - E.g. there is an architecture in which machine code is word addressed, but data is byte-addressed, with byte address = 4*word address
  - *void* is a pointer to data (or data objects), not a pointer to machine code (functions).*
  - *There is no such thing as a pointer to a function of any type - no function pointer analog to void* for data types. Can't use void* for this purpose - conflicts with possibility of different representations*
    - Example:
      - int (*fp) (double); /* a function pointer
      - void* get_function_address(char * function_name); /* similar to the dlfcn.h POSIX package for dynamically-loaded library access */
      - fp = get_function_address("foo"); /* warning or disallowed - can't convert between void* and a function pointer */
  - *What to do if we really need a generic function pointer?*

- Use a function pointer cast - tell the compiler we know more than it does about what is going on - namely that the void* address really is the address of a function of that type:
- fp = (int (*) (double)) get_function_address("foo"); /* cast the void* to the function pointer type */
- This is legal C, but as with all casts, we are telling the compiler to accept our belief that this is valid and will work. If we are wrong, we are on our own, and the results we get will be undefined.

Casts of function pointers for generic code

In C, generic code that can work for different data types has to be done in terms of void, usually involving generic functions that have void* parameters and return type.*

- You often need to given the generic code a pointer to a function that it can call using void* data, even if the function has to work in terms of actual data types.

Although you can cast a function pointer of one type to another type, that does not result in any parameter or return data type conversions.

- Recall that C has to support different representations of pointers to different types of data.
- Recall that the type of a function is used by the compiler to generate the code for setting up the call - this involves checking on the number and types of the parameters, and converting the argument values and return types if necessary.
- A function pointer contains only the address of where the function code is in memory. The type of the function pointer tells the compiler how to set up the call.
- So the compiler forbids you from assigning a function pointer to a function whose type does not match - this usually gives an error of "pointer types are incompatible". If this was allowed, there could be a mismatch in the calling code, resulting in undefined results - what happens will depend on details in the architecture and implementation.

For example:

- char (*fp) (int, double); /* a pointer to function with type "function returning char that has an int and a double as parameters"
- int foo(double, int) { blah blah} /* a function with type "function return int that has a double and an int as parameters"
- fp = foo; /* not allowed because the function types don't match, so a call of foo through fp results in undefined behavior. */
- char c = fp(int_var, double_var); results in contents of call stack and returned value that is different from what foo expects

This rule also applies to pointer parameters and return values, even if pointers usually have the same representation, because C has to support different representations of pointers.

- Applies even if void* pointers are involved, which the compiler is normally happy to implicitly convert void* to/from any other pointer type.

For example:

- void* (*fp) (void*, void*); /* a pointer to function with type "function returning void* that has an void* and a void* as parameters"
- char* foo(double*, int*) { blah blah} /* a function with type "function return char* that has a double* and an int* as parameters"

- fp = foo; /* not allowed because the function types don't match */
- void* result = fp(&double_value, &int_value); /* not allowed even though the call stack values and returned value are compatible */

▼ This is counter-intuitive if we are working on an architecture where pointers have the same representation for all data types - we know that the pointers will get represented on the call stack and returned value in a valid way.

- But C has to assume that e.g. char*, double*, int*, and void* might actually have different representations, so the rule applies and the assignment is not allowed.

▼ Upshot is that function pointers are limited to storing the address only of a matching function type. But this really interferes with writing generic code, which involves calling functions with void* parameters or returned values. What do we do? Two possibilities:

▼ 1. Wrap the actual function to be called with another function with matching parameter types - the wrapping function performs the conversions explicitly or implicitly:

- void* (*fp) (void*, void*); /* a pointer to function with type "function returning void* that has an void* and a void* as parameters"
- char* foo(double*, int*) { blah blah} /* a function with type "function return char* that has a double* and an int* as parameters"
- void* wrapper(void* p1, void* p2) /* explicit cast version*/
  {return (void*) foo((double*) p1, (int*) p2);}
- void* wrapper(void* p1, void* p2) /* implicit cast version*/
  {return foo(p1, p2); }
- fp = wrapper; /* function pointer type and function types match, no problem */
- char* result = (char*) fp((void*)&double_value, (void*)&int_value); /* call through fp with explicit casts to/from void* */
- void* result = fp(&double_value, &int_value); /* let compiler perform implicit conversions to/from void* */
- This method fully complies with the Standard, but the wrapping function is a nuisance bit of code to have to write. Also, note that the casting to/from void* will only be correct if we in fact are supplying the correct pointers to the data.

▼ 2. Use a function pointer cast to tell the compiler that we think it is OK to call the function even though the types are incompatible:

- void* (*fp) (void*, void*); /* a pointer to function with type "function returning void* that has an void* and a void* as parameters"
- char* foo(double*, int*) { blah blah} /* a function with type "function return char* that has a double* and an int* as parameters"
- fp = (void* (*) (void*, void*))foo; /* function pointer cast - tell compiler to treat foo as if it had same type as fp */
- char* result = (char*) fp((void*)&double_value, (void*)&int_value); /* call through fp with explicit casts to/from void* */
- void* result = fp(&double_value, &int_value); /* let compiler perform implicit conversions to/from void* */

- This is legal C, but as with all casts, we are telling the compiler to accept our belief that the call will function correctly. If we are wrong, we are on our own, and the results we get will be undefined.

- **Conclusion: The type system helps us write code that the compiler can guarantee will produce correct machine code - “type safety”. However, sometimes this type-safety conflicts with what we need to, and we need to bypass the type-safety. This is especially true for generic code.**
  - *C++ supports generic programming much more safely with templates.*

Memory allocation

- **Why do it? - To directly control the lifetime of a variable.**
  - *variables occupy space in memory - "exists" when it is valid to refer to the contents of that piece of memory reserved for it*
  - *the lifetime of a variable is the period of time when that variable's memory is reserved and it is valid to refer to it.*
  - *lifetime of automatic (local) variables is from when the function is called to when it returns*
  - *lifetime of static variables is from program startup to program termination*
    - resides in a fixed place in memory
  - *dynamically allocated variables have a program-contorlled lifetime: you reserve a piece of memory, keep the data in it, and you decide when you are done with it. Pass around pointer to it to whoever needs to refer to it.*
- **in C++ you have new/new[]/delete/delete[]**
- **in C you have malloc and free, and a couple of unnecessary variants.**
  - *malloc allocates memory, free returns it*
  - *declared in stdlib.h (misnamed!)*
  - *in this course, don't use anything except malloc - other variants work, but not necessary.*
- **getting a specified number of bytes of memory**
  - *void * malloc(size_t number-of-bytes);*
    - gtd to be alligned for anything you might want to put in it.
    - NULL is returned if memory not available
    - good programming practice to check, because it you don't, you can get unhelpful crash on protected-memory systems, or down in flames on unprotected systems
    - example code

```
#include <stdlib.h>/* for malloc declaration */
#include <stdio.h> /* for I/O */
. . .
        /* allocate an array of ints with size specified by int num_cells */

        int * p;
        p = malloc(num_cells * sizeof(int));
```

```
        /* obsolete version:
        The cast  is old custom from pre-ANSI days; unnecessary, and error-prone!
        p = (int *)malloc(num_cells * sizeof(int)); */

        if(!p) {
                printf("Out of memory!\n");
                exit(1); /* exits program with Unix signal for a failed execution */
                }
        /* use p, e.g., as if it were an array: */
        p[3] = 42;
. . .
        /* when finish, free the memory */
        free(p);
        /* don't refer to it after freed! results are undefined!!! */
        p[3] = p[2];   /* DANGER! DANGER! DANGER! HEAP CORRUPTION LIKELY! */
```

- GOTCHA
  - if you forget to do #include <stdlib.h>, then HORRIBLE THINGS might happen
    - int malloc(int) might be the resulting assumed declaration
    - if int is too small .... ow!
    - leaving the cast off would help detect it! - cast assign a pointer to an int implicitly!

- **when done, return memory with free()**
  - *void free (void * ptr)*
  - *ptr must point to a piece of memory previously allocated by malloc (same address as malloc returned)*
- **How does this work?**
  - *See simple example in K&R Ch. 8*
  - *malloc/free maintain a data structure to manage the memory, and these optimized for speed*
    - bookkeeping info (e.g. links between the blocks in the data structure) are typically stored right next to the block you're given - you could break them by writing outside the block.
    - once block of memory is free'd, the memory manager is free to write its bookkeeping info into the block, clobbering what is there.
    - turns out memory allocation is a very common substantial bottleneck as it is ...
    - for speed, no checks for correctness of pointer given to free, just updates the data structure unconditionally
      - very fast, very efficient, but completely stupid
- **DON'Ts**
  - *Don't allocate memory unnecessarily - this is a slow and error-prone process and should be avoided if possible.*

- `E.g. the "bookkeeping" in the memory manager takes a huge amount of time compared to allocation on the function-call stack.
- If anything wrong, you may well "corrupt the heap" and cause unpredictable, confusing errors or crashes - lots of ways to do it!

▼ *Don't forget to free the block when it is no longer needed*

- a memory leak

▼ *Don't lose the pointer value (address) - must keep it stored somewhere*

- if you need to re-use the pointer variable, save the value in another somewhere
- you need to give the block address to free in order to release the memory!

▼ *Don't scribble outside the allocated block*

- too easy to do with array subscripts, pointer arithemetic
- if you write outside the block, you might clobber the bookkeeping information in the memory pool.
- Tricky because often, actually allocated block is bigger than you asked for - minimum size because of alignment issues can give you a bit of extra space that you didn't ask for. So bug doesn't show up right away, only when data gets a little bigger, or you change machines, etc.

▼ *Don't free using an address you didn't get from malloc*

- if you give an invalid address to free, it will update the bookkeeping information in a bogus way

▼ *Don't free using the same address more than once*

- if you free more than once, free will update the bookkeeping information in a bogus way

▼ *Don't make use of the block (or never use the pointer value) after it has been freed*

- state of data in a freed block is undefined
- previous data might still be there but it might not - as soon as the block is recycled it can be changed, and by freeing the block you've made it available for recycling
- depending on the implementation, part of the data might be scribbled on right away by the memory manager for its book-keeping.
- Tricky, because often the data will still appear to be there, so the bug might no show up right away - only eventually, or when you change C implementations, etc.

▼ **DOs**

- *If you can do what you need with a local "automatic" variable, do so instead of allocating memory.*

▼ *As soon as you write "malloc", stop, ask, and decide "where is the free going to be?"*

- Then be sure to write that code!

▼ *Organize your code so that memory is allocated and deallocated in only a few clear places, and decide explicitly who is responsible for freeing the allocated memory and make it clear in comments (at least) - don't leave it up for inconsistency and confusion.*

- One approach - A "creation" function for a struct that allocates memory for it and any of its members, and a "destroy" function that does all of the deallocation. Then you only have to get it right in one place, and then remember only to always use the create and destroy functions.

- *Use a tool to check on memory allocation correctness! (e.g. purify)*

I/O

review of stream concept

- *record-oriented input - input divided into records of fixed length or delimited in some way - e.g. lines*
  - cf. old card-reader input in FORTRAN
  - approach for ascii text files - read the characters comprising the record into a string, then can use sscanf to parse it with library I/O
  - approach for binary data - read it unconverted into a block of memory, take it apart in various ways (e.g. cast pointer to a structure type).
- *C library also supports "binary I/O" - directly read/write streams or blocks of bytes of arbitrary size directly as bit patterns.*
  - e.g. output value of an int containing decimal 10 as a value, you get 00001010 as the last eight bits (depending on endianess).
  - We won't use binary I/O in this course.
- *C I/O library supports idea that input is a stream of individual characters*
- *Library will parse this stream for you in almost all situations*
  - unread characters remain in the stream to be read in the next input operation.
  - important property - direct support for type-ahead from the console
- *rare that you have to parse it yourself character by character - e.g. you are writing a parser or compiler of some sort*
  - so before you write an elaborate function, check for what's in the library
    - e.g. fgets and fputs are handy for read/write of a whole line to/from a C-string.
  - DON'T REINVENT THE WHEEL/RECODE THE STANDARD LIBRARY
    - a total waste of time, and your code is probably neither as good nor as reliable.
    - basic I/O functions are declared in <stdio.h>

Files and Streams:

- *Information kept in an implementation defined struct type, named FILE, declared in <stdio.h>*
- *Note: stdin, stdout, stderr are actually global variables of type FILE * defined and opened for you during program startup by code in the Standard Library.*
- *FILE* fopen(char * filename, char * mode)*
  - find the named file and open it in the specified mode (all needed for this course):
    - "r"
    - "w"

- other modes exist, but aren't needed for this course.

- return a pointer to the FILE struct if can find the file and open it, NULL if not.

- what if no file of that name? Long experience says it should depend on mode:

  - "r" - fails to open - NULL is returned - can't read something that isn't there.

  - "w" - new empty file is created; existing file is overwritten, failure only if the file can't be created for some reason (e.g. name is illegal, disk drive full, etc).

- *int fclose(FILE *)*

  - return EOF if an error, 0 if OK

- **variable argument lists in printf/scanf**

  - *basic form*

    - prototypes are int printf(char *, . . .); int scanf(char *, . . .);

    - ... - push any number of arguments onto the stack, doing standard promotions

      - e.g. float to double

    - only information about the number and the types of the arguments is the information in the control string

      - results *undefined* if the control string does not match the number and type of arguments - except if too many arguments, the extra ones are ignored.

  - *why they are unsafe*

    - compiler is not required to check that control string agrees with the arguments, and no way to tell at run time either

      - so you can get overflows or garbage I/O easy as pie!

      - newer compiler incorporate features of a favorite tool, "lint", and check this as much as they can

      - note that format string can be created at run time - doesn't have to be a constant -

    - C++ does things in a way that is usually more verbose, but is type safe.

Basic functions for stream I/O

- **use int `fprintf(FILE *, char *, ...)` or `int printf(char *, ...)` to output values of various types**

  - *printf(format, whatever) is equivalent to fprintf(stdout, format, whatever)*

  - *printf idea - cntrol string says how to convert from untyped values on the stack to ascii characters, put in the stream*

    - note: %f for double, because both float and double get converted to double on the stack

  - *returned value is number of bytes written - not particularly useful, usually ignored.*

- **use `fputs(char *, FILE *)` to output a whole C-string**
  - *writes to the file all the characters in the supplied string up to but not including the null byte.*
- **use `int fgetc(FILE *), int getc(FILE *), int getchar(void)` for single-character input**
  - *getchar() is the same as fgetc(stdin)*
  - *reads and returns the next character from the stream. If an error or end of file, returns #defined EOF value*
    - regardless of the kind of character the next character is - will read whitespace, e.g.
      - Note that everything can be read as a character.
    - Always use the Standard symbol EOF to check the returned value - don't assume you know what it is.
    - use `ferror(FILE *)` or `feof(FILE *)` to determine if the return of EOF is due to a "hard" I/O error or just an end of file.
    - functions return true if the state of the stream is an error or eof.
- **use `int fscanf(FILE *, char *, ...)` or `int scanf(char *, ...)` for "scanning" input for different types of values**
  - *scanf(format, whatever) is equivalent to fscanf(stdin, format, whatever)*
  - *the idea of scanf - control string says how to parse the input, convert ascii char sequences to values. The arguments are always a pointer to where to put the value. The characters in the stream are read and processed one at a time to determine the value to be stored.*
    - int scanf(char *, ...);
    - return number of values successfully read, or a value #defined in the library as EOF
      - EOF is often -1, but use the EOF symbol!
      - despite the name, EOF does not always mean "end of file!"
    - initial and intervening whitespace is almost always skipped when reading with scant
    - %d - look for a decimal integer, skip initial whitespace
    - %c - grab the next character regardless of what it is
      - " %c" - a space before the '%' - a space in the control string means to skip any amount of whitespace at that point. - not mentioned in K&R
        - actually scanf format strings can be used to do very elaborate parsing, but this is rarely used and so probably not worth learning - you are better off learning how to use perl and regular expressions instead.
    - %s - read in a string, and store as a C-string: skip initial whitespace, start reading and storing characters until whitespace or EOF, then put the null byte in at the end
      - %10s - don't read and store any more than 10 characters (need 11 char destination for the null byte at the end!)

- note that " %s" or " %10s" with leading whitespace is redundant - don't look ignorant by using it. %s really does skip leading whitespace, and doesn't need "backup" for it to happen!

- If you use a #define for the maximum input length, how can you get a corresponding format string? Best is two #defines, one for the number of chars to read, the other to declare arrays with - help prevent some typing errors.

  - Simplest: three #defines, kept together, change first two to change the maximum size allowed:
    ```
    #define MAXINPUTLENGTH 31
    #define INPUTFORMAT "%31s"
    #define INPUTARRAYSIZE MAXINPUTLENGTH + 1
    . . .
    char buffer[INPUTARRAYSIZE];
    . . .
    scanf(INPUTFORMAT, buffer);
    ```

  - Fanciest - using Standard Library functions that do "I/O" into C strings to create a format string at run time. sprintf, sscanf (and C++ stringstreams) are sometimes very handy, so good to know about them. Requires only one #define, so best for single point of maintenance. But it takes run time to generate the strings, so you should arrange to do it only once if possible.
    ```
    #define INPUTARRAYSIZE 32
    /* a global variable, authorized in project 1 specs */
    char g_format_string[10];

    int main(void)
    {
        /* do just once at startup */
        sprintf(g_format_string, "%%%ds", INPUTARRAYSIZE-1);
    . . .
    }

    /* in later code or other functions */
    . . .
         char buffer[INPUTARRAYSIZE];
    . . .
        scanf(g_format_string, buffer);
    ```

  - Craziest - Use macros to create the format string - esoteric and verbose, but it works; less useful to learn about because we don't like macros, especially in C++
    ```
    #define MAXINPUTLENGTH 31
    #define INPUTARRAYSIZE MAXINPUTLENGTH + 1
    /* we need both of the below because of how macros get processed */
    #define STRINGIFYHELPER(x) # x
    #define STRINGIFY(x) STRINGIFYHELPER(x)
    /* two different higher-level macros */
    /* Compiler concatenates a series of string literals into a single string
    literal */
    #define SAFESCANF(array_name) scanf("%" STRINGIFY(INPUTLENGTH) "s",
    array_name)
    #define SAFESCANF2(array_name, length) scanf("%" STRINGIFYHELPER(length) "s",
    array_name)

    . . .
    char buffer[INPUTARRAYSIZE];
    . . .
    /* The following three lines all turn into:
    scanf("%" "31" "s", buffer);
    which gets turned into
    scanf("%31s", buffer);
    */
    scanf("%" STRINGIFY(MAXINPUTLENGTH) "s", buffer);
    ```

```
SAFESCANF(buffer);
SAFESCANF2(buffer, MAXINPUTLENGTH);

/* for further explanation, see demo code in C_examples directory */
```

- %lf - look for something that can be a double value (might not have a decimal point) skipping initial whitespace, and store it where the argument points
- %f - ditto but for float (unusual, but note inconsistency with printf)
- %c and %s always succeed (unless EOF encountered, or your hardware is broken, but %d, %f, etc might not - if can't parse input as a number, stop, process no more input specifications in the format string. NOTE: no requirement that "all" of the input has to be parsed as a number.
  - example - a123 12x45
  - value returned is the number of successful conversions performed
  - can tell if scanf failed by seeing if returned value is different from number of values expected
  - if scanf tried to read past the end of the file, then EOF returned

- **use `char * fgets(char * s, int n, FILE * f)` to read an entire line (or part of a line) into a C-string.**
  - *The integer n is the size of the char array in bytes starting from the supplied char * s.*
  - *If succeeds, always gives you a valid C-string that fits into the array.*
    - test returned pointer value - if non-NULL, read was successful.
  - *Read characters from the stream f and store them into the array until one of the following happens:*
    - n-1 characters have been read and stored. A null byte is stored in the last cell of the array, and s is returned.
    - A newline character has been read and stored. A null byte is stored in the next cell of the array, and s is returned. You can tell if a whole line got read if the last character in the string is a '\n' character.
    - End of file was encountered after reading and storing at least one character. A null byte is stored in the next cell of the array, and s is returned.
    - End-of-file was encountered before any characters were read, or a "hard" I/O occurred; NULL is returned and the contents of s are undefined. If NULL is the returned value, use feof() to determine if the cause is end-of-file or a "hard" I/O error condition.
- **Important idea: if any possibility of input conversion failure, or end of file, must check the return value before the data gets used - always make sure it happens in this order:**
  - *#1. do the input operation*
  - *#2. check for success*
    - if success, and ONLY if success,
      - #3. use the data

        - #4. do the next step, which might involve going back to #1
    - if fails,
        - #5. do something appropriate
            - if end of data, stop reading and do the next step
            - if data was bad, it is an error, deal with it
- *People often write code that does #3 before #2, and sometimes even #3 before #1.*
- *C++ makes this a little easier to get right than C, but still a common mistake*
- *See C Coding Standards for more discussion.*

Union type

- **like struct, except that members occupy same memory space, size is dictated by the member requiring the largest memory space**
  - *union U {*
    char c;
    int i;
    double d;
    };
  - *c, i, and d share the same set of bytes*
  - *obviously, only one value can be stored there at a time*
- **used only when memory is at a premium**
  - *in one compiler, struct S has sizeof = 16, union U has sizeof = 8*
- **how tell what's there? Need another way to tell - either in your code's organization, or a separate "type" indicator that you carry along and set/query to see what's there**
  - enum Thing_type {C, I, D};
    struct Thing {
    enum Thing_type theType;
    union U theValue;
    };

    thing.theValue.d = double_var;
    thing.theType = D;
    ...
    if (thing.theType == D)
    double_var = thing.theValue.d; /* contains a double */
- **Demo of difference between structs and unions with the same member types - try it on your machine!**
  - /* demonstration of the difference between an union and a struct */

    #include <stdio.h>

    int main(void)
    {
    struct S {
    char c;
    int i;
    double d;
    } s;

    union U {
    char c;
    int i;
    double d;
    } u;

    /* The union is a lot smaller, even though the same number

```
        and types of members are declared. */
        printf("struct S has size %d\n", sizeof(struct S));
        printf("union U has size %d\n", sizeof(union U));


        s.c = 'x';
        s.i = 123;
        s.d = 3.14159265;
        /* all three values of the struct are stored separately */
        printf("s contains: %c, %d, %f\n", s.c, s.i, s.d);

        /* demo that only the last value stored in the union is well-defined */
        u.c = 'x';
        u.i = 123;
        u.d = 3.14159265;
        /* only the d value makes sense */
        printf("u contains: %c, %d, %f\n", u.c, u.i, u.d);

        u.d = 3.14159265;
        u.c = 'x';
        u.i = 123;
        /* only the i value makes sense */
        printf("u contains: %c, %d, %f\n", u.c, u.i, u.d);

        u.i = 123;
        u.d = 3.14159265;
        u.c = 'x';
        /* only the c value makes sense */
        printf("u contains: %c, %d, %f\n", u.c, u.i, u.d);


        return 0;
}
```