

C Basics 1: Overview & C vs. Java

1. Overview of C and C vs. Java

- **history of C**
- **C standards**
- **C vs. Java characteristics**

2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling

C Basics 1: Overview & C vs. Java

9. Functions and Parameter Passing
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Overview of C

C was developed by Dennis Ritchie at AT&T's Bell Labs, starting in 1969.

A key goal was to provide a “high-level” programming language to support development of **portable, cross-platform** programs (up to that point, many programs were being written in machine-specific **assembly language**).

Ritchie also co-developed **UNIX** (along with Ken Thompson and others) starting around the same time.

By 1973, UNIX had been implemented primarily in C, becoming the first (largely) *portable* operating system.

Overview of C (contd.)

While C is much more abstract than assembly language, it lacks features found in modern “high-level” programming languages that are designed to support programming.

C programs are intended to be able to be made nearly as efficient as assembly language programs.

It is often said that a (properly designed) C program will be about as fast as is possible.

With a C program, what occurs at runtime is largely only what the program indicates will happen.

By contrast, in a *Java* program, every array access will be *bounds checked*, *garbage collection* routines will be run periodically, etc.

Overview of C (contd.)

While programmers new to C are often dismayed by the language failing to “hold their hands” and help discover their bugs, if a program is properly designed, automatic operations such as array bounds checking are simply a waste of CPU cycles.

C also differs from most modern high-level languages in allowing the user to directly access and manipulate memory, via **pointers**.

C remains the most commonly used language for programming with **system calls** and for **system programming** (coding of *operating systems* and **embedded systems** software).

C is also one of the most widely available languages, with *free-of-cost* compilers for virtually every computer architecture.

C Language Standards

In 1978, Brian Kernighan and Dennis Ritchie wrote the first book describing C: “**The C Programming Language**.”

This book became the *de facto* language standard, often referred to as **K&R C**.

The first standardized versions of C were ratified by ANSI in 1989 and ISO in 1990, commonly referred to as **C89** or **C90**.

A revised version of the ISO C standard was published in 1999, commonly referred to as **C99** (formal name **ISO/IEC 9899:1999**).

Among the features added in C99 were: variable-length arrays (array size determined at runtime), intermingling of declarations and code, allowing index variable declarations in `for` statements, new types (`long long int`, `_Bool`, `complex`), and inline functions.

C Language Standards (contd.)

Another version was ratified in 2011, and is known as **C11** (**ISO/IEC 9899:2011**).

Key extensions of general interest include: better Unicode support, generic function selector (`_Generic`), and language support for **multithreaded programs**.

The multithreading support involves several elements: an uninterruptible data type (`_Atomic`), thread specific/local storage, and library functions that provide thread operations (creation, mutexes, condition variables, etc.).

This standard also addresses special uses such as embedded processors and better defines various aspects of implementations.

C Language Standards (contd.)

The latest C standard was ratified by ISO in 2017 but not published until 2018 (**ISO/IEC 9899:2018**).

It is usually known as **C17**, but may also be referred to as **C18**.

It did not add any new features, but instead involved technical corrections to C11.

The next draft standard is known informally as **C2x**, and is expected to be voted on in 2023 (so would be **C23**).

C Language Standards (contd.)

Most C compilers, e.g., **GCC**, now support all C17 features.

However, which versions features are *enabled by default* can vary, so one may require options to compile code that uses them.

E.g., with GCC: **-std=c17**

GCC supported C standards as follows:

- C99 as of v. 4.5
- C11 as of v. 4.7
- C17 as of v. 8.1

C vs. Java

Java is a “**C family**” language and so shares much syntax with C, such as declarations, control constructs, operators, etc.

C is a purely **procedural language**, however, while Java is a purely **object-oriented language**.

C does *not have any support* for **OOP** concepts like **classes**.

In a procedural language like C, programs are structured in terms of **functions (subroutines)** operating on **data** held in **variables**.

The focus is on the *actions* that need to be taken to accomplish the goals of the program.

With OOP, programs are structured in terms of classes of objects and the operations that can be done to them.

C vs. Java (contd.)

	C	Java
Programming Paradigm	pure procedural	pure OOP
Composite Data Types	structs	classes
Inheritance	no	single: subclass <i>extends</i>
Encapsulation Mechanism	files	classes
Compilation Modularity	files	classes (files)
Memory Management (heap)	manual <code>malloc()</code> , <code>realloc()</code> , <code>free()</code> , etc.	garbage collection <code>new</code>

C vs. Java (contd.)

	C	Java
Pointers	explicit, manipulable: defined: <i>basetype*</i> <code>int *ip, i;</code> <code>ip = &i + 1;</code>	implicit, not manipulable called <i>references</i> all classes
Invalid Pointer	NULL (<code>void *</code>)0	null
Numeric Types	integer: <code>char, short, int, long, long long</code> real: <code>float, double, long double (C99)</code> other: <code>_Bool (C99)</code> sign: <code>signed, unsigned</code> minimum sizes only	integer: <code>byte, short, int, long</code> real: <code>float, double</code> (all integers signed) defined sizes

C vs. Java (contd.)

	C	Java
Boolean Type	<C99 no, use int's C99: <code>_Bool</code> (0/1 int) 0 is false 1 is standard true, but any $\neq 0$ considered true	yes: false, true
Characters	ASCII encoding single byte integer constants: <code>'a'</code> , <code>'\141'</code> C99: wide char's C11: Unicode	Unicode encoding two bytes constants: <code>'a'</code> , <code>'\u0041'</code>
Strings	non-builtin type: type <code>char*</code> (char array, requires <code>'\0'</code> sentinel) constants: <code>"abc"</code>	builtin reference type: <code>String</code> constants: <code>"abc"</code>
String Comparisons	string library functions e.g., <code>strcmp()</code>	String class methods e.g., <code>.equals()</code>

C Basics 1: Overview & C vs. Java

©Norman Carver

C vs. Java (contd.)

	C	Java
Implicit Type Conversion	extensive: <i>promotion & demotion</i> may not preserve values: unsigned int \rightarrow signed int	limited: promotion
Type Casting	yes: <i>(type) var/expr</i>	same
Arrays	access: <code>arr[i]</code> declaration: <code>int iarr[constant]</code> (C99 allows <i>expr</i>) static size no bounds checking	access: <code>arr[i]</code> declaration: <code>int[] iarr = new int[expr]</code> static size (vs. vectors) auto bounds checking

C Basics 1: Overview & C vs. Java

©Norman Carver

C vs. Java (contd.)

	C	Java
Main	<code>int main(int argc, char *argv[])</code> <code>int main()</code>	<code>public static void main(String[] args)</code>
Operators	assignment: <code>=</code> , <code>+=</code> , etc. arithmetic: <code>+</code> , <code>++</code> , etc. relational: <code>==</code> , <code>!=</code> , etc. logical/bitwise: <code>&&</code> , <code>&</code> , etc.	same same same similar
Selection Constructs	<code>if</code> , <code>else</code> , <code>switch</code>	same
Looping Constructs	<code>for</code> , <code>while</code> , <code>do-while</code>	same
Global Variables	yes (external definitions)	no, but: <code>public static</code> members
Automatic Variable Initialization	generally no: global and <code>static</code> only	yes (to 0 or null)

C Basics 1: Overview & C vs. Java

©Norman Carver

C vs. Java (contd.)

	C	Java
Subroutines	functions	class methods
Subroutine Values	any type plus void	same
Parameter Passing	call-by-value: but pointer params act like call-by-ref	call-by-value: but reference params act like call-by-ref
Generic Functions	use untyped pointers: <code>void*</code> C11: generic selector <code>_Generic</code>	generics system (SE 5.0+)
Function Overloading	no	yes (methods)
Operator Overloading	no	no

C Basics 1: Overview & C vs. Java

©Norman Carver

C vs. Java (contd.)

	C	Java
Function Arguments (Callbacks)	yes: <code>hofunc(void (*h)(int));</code>	SE 8.0+: lambdas (previously only messy alternatives to achieve effect)
Variadic Functions	yes: (<code>stdarg.h</code>) <code>printf(char *fmt, ...);</code>	SE 5.0+
Optional or Default Parameters	no	no
Preprocessor	yes	no
Header Files	yes	no
Namespaces	no, single space (for <i>ordinary identifiers</i>)	yes: <code>package, import</code>
Exception Mechanism	no	yes: <code>try, catch</code>

C vs. Java (contd.)

	C	Java
Concurrency: Processes	via system calls (<code>fork()</code> , etc.)	limited: <code>ProcessBuilder</code> class
Concurrency: Threads	via system calls (<code>Pthreads</code>) C11: yes	yes: <code>Thread</code> class <code>java.util.concurrent</code> package

C Basics 2: C Program Elements

1. Overview of C and C vs. Java
2. **C Program Elements**
 - **comments**
 - **header includes**
 - **function prototypes**
 - **global variables**
 - **main**
 - **program termination**
 - **compilation**
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings

C Basics 2: C Program Elements

7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Program Layout

A typical layout for a **C source file** is:

1. **comments**
2. **header includes** (for library and external user functions)
3. **defines** for constants/macros (if used)
4. **prototypes** (for functions defined in this file)
5. **global variable** declarations (if used)
6. **main** (if in this file)
7. function definitions

(Some people prefer to put `main` last, possibly avoiding the need for some prototypes.)

Program Layout (contd.)

```
// A partial example C program

#include <stdlib.h>
#include <stdio.h>

#define MAX_LINE_LENGTH 100

//Prototypes:
char *get_line(FILE *fpntr);

//Global variables:
int flag = 0;

int main(int argc, char *argv[])
{
    ...
    return EXIT_SUCCESS;
}

char *get_line(FILE *fpntr)
{
    ...
}
```

Comments

The traditional syntax for C comments is that they start with “/*” and end with “*/”, possibly extending over multiple lines:

```
/* This is a multi-line comment, which will be totally
   ignored by the C compiler, but don't forget to
   terminate it or you will be sorry!           */
```

GCC extensions have also allowed *single line comments*, and these were added to C99:

```
// These comments can be multi-line only by repeating the
// comment double slashes, but there is much less chance
// of forgetting to terminate the comment!
```

Header includes

The core C language is very small, with much of the functionality required to write all but the simplest programs being supplied in the **C Standard Library**.

Key libraries (standard library components):

- **stdlib** – set of important functions like `exit()`, `malloc()`, etc.;
- **stdio** – I/O functionality, to read/write to terminal or files;
- **string** – functions for manipulating C strings;
- **math** – trigonometric functions, etc.

Header includes are required when calling functions from any of these libraries, for example:

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

Function Prototypes and Header includes

A **prototype** declares the function name, parameter types, and return type (parameter names are optional):

e.g., `char *get_line(FILE *)`

The C compiler must have seen either a function's prototype or the function's definition before a function is called.

This applies to all functions, including *library functions* and even the C Standard Library functions.

Prototypes are often gathered into **header files** (.h files), which are include'd via a **preprocessor directive**.

Placing prototypes for functions defined in a file near the top of the file makes it easier to see what is defined in the file.

Global Variables

Global variables are variables whose **scope** is not limited to a **lexical environment** such as a function body.

In C, global variables are created by declaring a variable *outside* of the body of `main` or any function.

Technically, these are variables with “*external definitions*.”

By default such variables have **file scope**, meaning they can be referenced in all functions defined subsequently in the file.

However, through a process called **linkage**, global variables' scope can be extended to an entire, multi-file program.

main

A C program's functions can be distributed among a *set* of source files, and these files are able to be separately compiled.

There must be exactly one `main` among the files, which will be the **entry point** for the executable.

`main` has an `int` return type, which is the program's Linux/UNIX **exit status**.

`main` can be set up to accept **command-line arguments** using the `argc/argv` mechanism or not:

- `int main(int argc, char *argv[])`
- `int main(void)`
- `int main()`

main (contd.)

Command-line arguments mechanism:

- `argc` – *argument count*, the number of arguments in `argv`;
- `argv` – *argument vector*, array of arguments as *strings*;
- `argv[argc]` is `NULL` (a **sentinel**).

By convention, Linux/UNIX passes the program name/path as the first element of `argv`, `argv[0]`.

This means that `argc` will be *one greater* than what you think of as the number of arguments.

main (contd.)

For example, consider the command:

```
prog -n25 test.txt
```

The program `prog` takes *two arguments*, but `argc` will be 3, and `argv` will be a 4-element array:

"prog"	"-n25"	"test.txt"	NULL
<code>argv[0]</code>	<code>argv[1]</code>	<code>argv[2]</code>	<code>argv[argc]</code>

(Each `argv` element is a string, and strings are arrays, so what is stored in each `argv` element is actually a **pointer** to the array of `char`'s making up the string.)

Program Termination

C programs are generally terminated by calling `exit()`, either from inside `main` or from inside any function:

```
#include <stdlib.h>
```

```
void exit(int status);
```

The parameter `status` is what is returned from `main`, i.e., the program's *exit status*.

(Actually, only the *low order* eight bits of `status` are returned, meaning `status` should take on values between 0 and 255.)

Program Termination (contd.)

stdlib.h defines two constants that should generally be used to make the exit status obvious and portable:

- EXIT_SUCCESS (zero for Linux/UNIX)
- EXIT_FAILURE (one for Linux/UNIX)

Another method for terminating a C program is by calling `return` inside of `main`.

It is purely a matter of style whether one uses `return` or `exit()` when terminating a program from inside of `main`.

Note, however, that `exit()` is a function while `return` is not:

```
exit(EXIT_SUCCESS);
```

vs.

```
return EXIT_SUCCESS;
```

Program Termination (contd.)

Even if your `main` calls a function that will call `exit()`, most C compilers will want to see `return/exit()` at the end of `main` to guarantee termination and a status value.

In addition to terminating the program, `exit()` can also first call any “cleanup” functions that have been *registered* via calls to `atexit()` or `on_exit()`:

```
int atexit(void (*function)(void))
int on_exit(void (*function)(int, void *), void *arg)
```

The notation “`void (*function)(void)`” means `atexit` requires **function arguments** that take no parameters and have void returns:

```
void done(void){ printf("Done.\n"); }
...
atexit(done);
...
```

Compilation

Before a C program can be executed, it must be converted from **source code** (text) to a binary, **executable** file format.

The format will be specific to both an operating system and a **processor architecture**.

The process of converting the C source file(s) to an executable file is generally referred to as **compilation**, but the **compiler** proper is just one of the tools used in the transformation.

The transformation is actually accomplished in a sequence of stages, each producing a new file format.

The GNU C compiler, **GCC**, carries out the required sequence of steps by default, automatically invoking the other tools that are required (options can be supplied to limit the actions taken).

Compilation (contd.)

The transformation from source code to executable involves four stages:

1. **Preprocessing:** the GNU C **preprocessor**, `cpp`, executes any preprocessor directives in a C source (`.c`) file, producing an intermediate (`.i`) file (C source that doesn't need preprocessing);
2. **Compilation Proper:** `gcc` compiles the C statements in a `.i` file, resulting in an **assembly language** (`.s`) file (still text);
3. **Assembly:** the GNU **assembler**, `as`, converts an assembly language `.s` file into **binary, machine code** format, resulting in an **object code** (`.o`) file;
4. **Linking:** the **linker/loader**, `ld`, takes one or more `.o` object files, resolves **references** among the files and to library functions, producing a single **executable** file.

Compilation (contd.)

Linux executables use a format called **ELF (Executable and Linkable Format)**.

An executable file contains machine code for the program (and static libraries) in a form that can be loaded into memory and executed.

Linking allows a large C program to be broken into *multiple* source (.c) files (*“modules”*) that can be compiled *independently of one another*, into separate object (.o) files.

The linker takes the set of object (.o) files for the complete program, combines their code, then converts references to functions and globals in another object file (or a library) to the appropriate memory address in the single resulting executable file.

Compilation (contd.)

Most of the C Standard Library functions are in the main C library file, **libc**, which is *automatically linked* (e.g. during compilation).

The functions in the *math* component, however, are in a separate library file, **libm**, which is not automatically linked.

Thus, use of standard library math functions not only requires including **math.h**, it will also require telling GCC to have the linker link with **libm**, which is done by including the option *“-lm”*:

- *“-l”* is the option to specify an additional library to link;
- *“m”* is the shorthand for **libm**;

C Basics 3: The Preprocessor

1. Overview of C and C vs. Java
2. C Program Elements
3. **The Preprocessor**
 - **#include and header files**
 - **#define and constants/macros**
 - **conditional directives**
 - **stringification and concatenation**
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O

C Basics 3: The Preprocessor

8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

The Preprocessor

A somewhat unique feature of C is the **preprocessor**, which is run before the actual C compiler and *rewrites* the C source code file before the C compiler runs.

The preprocessor executes various **preprocessor directives** that are embedded in the C source code file.

Preprocessor directives are used for:

- *including* text from **header** and other source code files
- *defining constants* and simple **macros**
- *conditionally* taking actions such as including code

Preprocessor directives begin with a "#":
#include, **#define**, **#ifdef**, etc.

The Preprocessor (contd.)

Whitespace is generally allowed on the line before the "#" and even between the "#" and the directive name.

However, some older C compilers require the "#" to be in column 1 of a line!

Since preprocessor statements are *not C statements*, they are *not to be terminated* by a ";":

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

This is a *common source of compiler errors* for beginning C programmers.

#include and Header Files

The C compiler must have seen either a function **prototype** or the function's definition before a function can be called.

This applies even to functions from the *C Standard Library*.

Typically, *prototypes* for the functions in a library are gathered into a **header file** (.h).

In order to use functions from the library, a source file must *include* the library's header file.

This is done using a preprocessor `#include` directive:

```
#include <library.h>
```

The `#include` directive literally causes the (header) file's contents to be inserted, so the contents is part of the source file when the C compiler runs.

#include and Header Files (contd.)

Having `#include` statements for the correct *header files* is a source of problems for many beginning C programmers.

Correct *header includes* will be required to call *C Standard Library functions* and *system calls*.

Correct header includes will also be required to use various standard **constant symbols** such as `NULL`.

If your code lacks appropriate header includes, you will get compiler warnings or errors that a constant or function is *undefined* or *implicitly defined*.

Even you get only *warnings*, you need to figure out which header includes your code is missing to ensure proper compilation!

#include and Header Files (contd.)

The necessary header includes are listed in the **man page** for a function or library call.

E.g., for `printf()` the man page says:

SYNOPSIS

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

```
int printf(const char *format, ...);
```

This shows that you should have the following header include line in your code in order to use the `printf()` library function:

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

#include and Header Files (contd.)

The *angle bracket* notation, `<...>`, tells the preprocessor to look for the file `stdio.h` in the *standard location* (e.g., `/usr/include`).

If you want to include your own header file, use something like:

```
#include "lab1.h"
```

Information about C Standard Library header files can generally be found in a man page, e.g., `"man stdio.h"`

You will usually want to always include the following two header files in code for this class: (to have standard constants)

- `stdlib.h`
- `unistd.h`

#define and Constants/Macros

The preprocessor `#define` directive is used to define **constants** and **macros**.

The preprocessor uses these `#define` definitions to perform *textual replacement* prior to the C compiler processing the source file.

E.g., a constant:

```
#define TRUE 1
```

The preprocessor will replace instances of the word `TRUE` in the source code with `1`, so the compiler will see only `1`.

It is common convention to make preprocessor constants *uppercase*.

#define and Constants/Macros (contd.)

Preprocessor constants are often used to define program attributes that might need to change:

```
#define BUFFER_SIZE 512
...
char buff[BUFFER_SIZE];
```

The `#define` directive can be used to define simple **macros** as well as a constants.

(In fact a preprocessor “constant” is really a preprocessor *macro* that does *not* take any *arguments*.)

#define and Constants/Macros (contd.)

Macros look like functions in that they have parameters:

```
#define MAX(a,b) (a > b ? a : b)
```

Now “`MAX(i1,i2)`” in the source would be replaced by the code:

```
(i1 > i2 ? i1 : i2)
```

It is important to understand that this replacement takes place during the preprocessor stage, so the expansion gets compiled into the executable.

This means there is not the runtime overhead of calling a function.

#define and Constants/Macros (contd.)

Writing completely reliable macros can require care.

For example, since `a` and `b` could be expressions (e.g., “`i+3`”), ensuring correct evaluation order requires `MAX` be defined as:

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

In fact, this version still has issues since it may evaluate each argument twice, causing unexpected results when the arguments have *side effects* (e.g., “`i++`”).

Obtaining single evaluation of the arguments requires this: (note: uses GCC `typeof` extension to C)

```
#define MAXEXT(a,b) \
    ({ typeof(a) _a = (a); \
      typeof(b) _b = (b); \
      _a > _b ? _a : _b; })
```

Conditional Directives

The preprocessor supports a set of “if-else” directives: `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` and `#endif`.

These directives can be used for *conditional inclusion* of code, hence for **conditional compilation**.

Example of compiling for UNIX vs. Windows (from Wikipedia):

```
#ifdef __unix__ // typically defined by UNIX compilers
# include <unistd.h>
#elif defined _WIN32 // typically defined by Windows compilers
# include <windows.h>
#endif
```

Conditional Directives (contd.)

Note the use of the “defined” operator, which returns “true” if a preprocessor constant/macro is defined (at this point in the code).

`#ifdef __unix__` is equivalent to `#if defined __unix__`.

Preprocessor constants can be `#define`'d in source code files and/or in header files.

They can also be defined using the “-D” option to GCC:

```
gcc -Wall -DDEBUG -oprogram prog.c
```

or

```
gcc -Wall -DDEBUG=2 -oprogram prog.c
```

Conditional Directives (contd.)

This can be used to conditionally include debugging code:

```
#ifdef DEBUG
    printf("x: %d\n",x);
#endif
```

One can even define different debug levels:

```
#if DEBUG == 2
    printf("completed\nx: %d\n",x);
#elif DEBUG == 1
    printf("completed\n");
#endif
```

Token Stringizing

Sometimes you need to put double quotes around text containing double quotes, and the **stringizing operator** `#` can make this more clear:

```
#define STR(arg) #arg
...
printf(STR(\nargv[1]: "%s"\n\n),argv[1]);
```

Stringizing involves more than putting double-quotes around the argument: embedded quotes must be backslash-escaped, etc.

E.g., stringizing `p = "foo\n";` results in `"p = \"foo\\n\";`.

Backslashes that are not inside string/char constants are not duplicated: `\n` itself stringizes to `"\n"`.

Token Stringizing (contd.)

Sometimes you may want to turn preprocessor constants into string constants, e.g.:

```
#define PORT 3060
...
getaddrinfo(hostname,"PORT",...); //need "3060" for port arg
```

(Won't work because macros inside double quotes are not expanded!)

Stringizing macros requires two levels of macros, because stringized arguments are not themselves macro-expanded:

```
#define STR(arg) #arg
#define XSTR(macro) STR(macro)
...
#define PORT 3060
...
getaddrinfo(hostname,XSTR(PORT),...); //get "3060" for port arg
```

Token Concatenation

Concatenating text onto a preprocessor constant is not allowed:

```
#define TYPE pid
...
TYPE_t process1; //do not get pid_t
```

However, the **token pasting operator** `##` can do it:

```
#define TYPE(stem) stem ## _t
...
TYPE(pid) process1;
```

Since arguments are not macro-expanded, cannot do:

```
#define TYPE(stem) stem ## _t
#define PTYPE pid
...
TYPE(PTYPE) process1;
```

Token Concatenation (contd.)

GCC manual has this example:

```
struct command {
    char *name;
    void (*function) (void); };

#define COMMAND(NAME) { #NAME, NAME ## _command }

struct command commands[] = { COMMAND (quit),
                              COMMAND (help),
                              ...
                              };
```

Avoids repeating names:

```
struct command commands[] = { { "quit", quit_command },
                              { "help", help_command },
                              ...
                              };
```

String Concatenation

While it is not possible to combine token stringization and token concatenation, C automatically *concatenates string constants* that appear sequentially:

```
printf("Name: %s\n","Norman" " " "Carver");

#define FIRST Norman
#define LAST Carver
printf("Name: %s\n",XSTR(FIRST) " " XSTR(LAST));
```

C Basics 4: Identifiers and Data Types

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. **Identifiers and Data Types**
 - **identifier naming**
 - **integer types**
 - **floating point types**
 - **constants**
 - **sizes/ranges of types**
 - **the sizeof operator**
 - **type conversions and casts**
5. Pointers and Arrays
6. Strings

C Basics 4: Identifiers and Data Types

7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Identifier Naming

The term **identifier** refers to **tokens** (words) used in programming languages to denote functions/methods, variables, and constants.

Programming languages limit the range of possible symbols for use in identifiers and may include **naming conventions**.

Identifiers in C must meet the following requirements:

- use only **alphanumeric** characters (a-z, A-Z, 0-9) plus the **underscore** (`_`);
- do *not begin* with a *digit* (0-9);
- do *not begin* with *two underscores*;
- do *not begin* with an *underscore followed by a capital letter*;
- are *not keywords*;
- are **case sensitive**;

Identifier Naming (contd.)

Java includes quite extensive naming conventions for classes, methods, source files, etc. (e.g., a class must be capitalized, a method/variable not, use “**camel case**”).

The C standards do not specify such conventions.

C identifiers are typically written *all lowercase*, with words *separated by underscores*: e.g. `process_array_row_order`

Preprocessor constants and macros are generally written *uppercase*: e.g., `MAX_LINE_LENGTH`

Compiler and system-related special symbols start and end with double underscores or a single underscore followed by a capital letter: `__STRICT_ANSI__`, `_POSIX_C_SOURCE`, `_Exit()`.

A few system calls also begin with underscore(s): `_exit()`, `__clone2()`

Integer Data Types

C has five **standard signed integer types**.

Each type has a full name, but abbreviations may also be used:

- signed char;
- signed short int / short / short int / signed short;
- signed int / int / signed;
- signed long int / long / long int / signed long;
- signed long long int / long long / long long int / signed long long (C99).

(The signed *keyword* is required only for signed char, so it is not typically included when specifying the other types.)

Integer Data Types (contd.)

C also has **unsigned** integer types, for representing only non-negative integers.

There are six **standard unsigned integer types**:

- _Bool (C99);
- unsigned char;
- unsigned short int / unsigned short;
- unsigned int / unsigned;
- unsigned long int / unsigned long;
- unsigned long long int / unsigned long long (C99).

The *standard signed integer types* and *standard unsigned integer types* together constitute the **standard integer types**.

Integer Data Types (contd.)

The “char type” is treated specially for historical reasons.

There are *three distinct* types: char, signed char, unsigned char.

It is *left up to the implementation* whether the char type is equivalent to signed char or to unsigned char.

This can vary by compiler and by computer architecture, so can cause **portability** problems.

Whether char is signed or unsigned does not matter when representing ASCII characters, but can when representing integers.

A key example is when dealing with the C I/O library symbol EOF, which is an int, and typically negative.

Boolean Data Type

C originally had no Boolean data type, integer values (typically int's) were instead used:

- 0 value is considered False;
- 1 or any value $\neq 0$ is considered True;

C99 added the integer _Bool type, an integer type with only values 0 and 1.

Provides some improvement in code readability and reliability.

(CPUs cannot manipulate single bits, so no storage savings.)

Boolean Data Type (contd.)

C99 also added the header `<stdbool.h>`, defining these *macros*:

- `bool` – expands to `_Bool`;
- `false` – expands to constant 0;
- `true` – expands to constant 1;
- `__bool_true_false_are_defined` – expands to 1;

In C99 code, it is common to `#include <stdbool.h>` and then use `bool`, `false`, `true` instead of directly using `_Bool` with values 0/1.

Floating Point Data Types

There are three **real floating types**:

- `float`;
- `double`;
- `long double` (C99);

C99 also has three **complex types**, which together with the *real floating types* constitute the **floating types**.

The `char` type, *signed integer types*, *unsigned integer types*, and *floating types* are collectively called the **basic types**.

Numeric Constants

Decimal integer constants are represented as sequences of digits *not starting with zero*: 123456

Suffixes may be used to indicate type:

- `u/U` – unsigned
- `l/L` – long
- `ll/LL` – long long

With *no suffix*, a decimal integer constant will be the “smallest” of the types `int`, `long int`, or `long long int` required.

Octal integers must be prefixed with a zero: 0711

Hexadecimal integers must be prefixed with `0x` or `0X`: `0xA12B`

Numeric Constants (contd.)

Floating constants must contain a decimal point: 123.456, 123., or .456

They may also contain an **exponent part**: 123.456e78

Suffixes may be used to indicate type:

- `f/F` – float
- `l/L` – long double

With *no suffix*, a decimal floating constant will be of type `double`.

Signs (+/-) may be indicated as a prefix for all numbers, and as an exponent prefix for floating constants: -123, -123.456e-78

Character Constants

Most **character constants** are represented by the character enclosed by *single quotes*: `'a'`

Character constants are `int`'s in C (but can be converted to `char`'s without loss of information).

The integer value of a character is the character's ASCII code.

The **backslash** character is also known as the **escape character** and is used in various **escape sequences**.

The single quote and backslash characters must be "escaped": `'\''` and `'\\'`

A number of ASCII "**control codes**" and other *non-printing characters* can be represented with special escape sequences.

Character Constants (contd.)

Here are some important examples:

- (horizontal) tab (ASCII 9): `'\t'`
- linefeed (ASCII 10): `'\n'`
- carriage return (ASCII 13): `'\r'`

Any ASCII character can be represented with an escape sequence that specifies the character's octal or hexadecimal code:

- (horizontal) tab (ASCII 9): `'\11'` or `'\x9'`
- linefeed (ASCII 10): `'\12'` or `'\xA'`

An important character because of its role as the **sentinel** in C strings is the **null character** (ASCII code zero): `'\0'`

Declarations and Constants

Example basic types declarations:

```
int x = 123456;
```

```
unsigned y = 1234u;
```

```
unsigned short ys = y;
```

```
long long z = 12345678911l;
```

```
double w = 1234.56e7;
```

```
char c = 'a';
```

Data Type Sizes/Ranges

Unlike Java, C specifies only *minimum sizes* for its types—e.g., `int`'s must be at least two bytes.

This approach has two important consequences:

- a type's size can vary from architecture to architecture
- different types need not be different sizes

While both a `short int` and an `int` can have an identical, two bytes representation, they are still considered *different types*.

C also requires that `long long` \geq `long` \geq `int` \geq `short`, etc.

The variability of size types can make it more difficult to write **portable code** (that can run on any computer architecture).

Data Type Sizes/Ranges (contd.)

The limits for types on the particular architecture are included in the header file `limits.h`, using symbols such as:

`INT_MAX`, `INT_MIN`, `UINT_MAX`

Typical type sizes on modern architectures:

<code>char</code>	1 byte
<code>short</code>	2 bytes
<code>int</code>	4 bytes
<code>long</code>	4 or 8 bytes
<code>long long</code>	8 bytes
<code>float</code>	4 bytes
<code>double</code>	8 bytes
<code>long double</code>	8, 12, or 16 bytes

The sizeof Operator

Writing **portable code** can require explicitly checking the sizes of types at runtime.

The **sizeof operator** can be used for that purpose:

```
int *iarr = malloc(sizeof(int)*10);
```

It returns the “size” of its argument *in bytes*.

The syntax is:

- `sizeof (type/variable/expression)`
- `sizeof variable/expression`

Note that `sizeof` can look like a *function call*, but is an *operator*.

The sizeof Operator (contd.)

A common confusion is how `sizeof` works with arrays (including “dynamic arrays” from `malloc()`).

If applied to a standard array variable (whose size is specified in the code), `sizeof` returns the *number of bytes* used by the array.

So the following might print 40:

```
int iarr[10];
printf("Size of 10 ints: %u\n", sizeof iarr);
```

However, “dynamic arrays” and array *function parameters* will be reported as the size (in bytes) of pointers/addresses,

So the following might print 4:

```
int *iarr = malloc(sizeof(int)*10);
printf("Size of pointer: %u\n", sizeof iarr);
```

The sizeof Operator (contd.)

C does not have any operator to allow you determine the *number of elements* in an array.

A common idiom to determine this is:

```
sizeof(iarr) / sizeof(iarr[0])
```

So the following would print 10:

```
int iarr[10];
printf("Elements: %u\n", sizeof(iarr)/sizeof(iarr[0]));
```

Be careful, though, because the following will *not* produce 10:

```
int *iarr = malloc(sizeof(int)*10);
printf("Elements: %u\n", sizeof(iarr)/sizeof(iarr[0]));
```

Type Conversions and Casts

A **type conversion** is the change of a value from one type into a value of another type.

C is very *flexible* about conversions, and there are many operators that will *implicitly/automatically* convert between types (e.g., assignments).

Implicit/automatic type conversion is referred to as **type coercion**.

Type conversions may also be *explicitly specified* by means of a **cast** operator (**type casting**): *(newtype)expression*

We can also distinguish between type **promotion** and **demotion**.

Promotion occurs when a conversion is from a type whose values are a *subset* of the target type, so the data value can **be preserved**.

Conversions and Casts (contd.)

With demotion, on the other hand, the data value *may not be preserved* because the target type is able to represent only a subset of the source type's values.

While conversions in C preserve values when possible, C will also do type demotions where values cannot be preserved.

This can lead to serious bugs, particularly when programmers do not understand how conversions occur.

One textbook on security has two full chapters devoted to what one needs to understand about C type conversions to avoid writing C code with vulnerabilities!

Unfortunately, C's rules are not always too clear or what happens may be left up to each implementation.

Conversions and Casts (contd.)

Here is one portion of the C99 standard dealing with conversions:

- When a value with integer type is converted to another integer type other than `_Bool`, if the value can be represented by the new type, it is unchanged.
- Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.
- Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

C Basics 5: Pointers and Arrays

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. **Pointers and Arrays**
 - **pointers**
 - **pointer operators**
 - **void***
 - **NULL**
 - **pointer problems**
 - **arrays**
 - **arrays and pointers**
 - **pointer arithmetic**

C Basics 5: Pointers and Arrays

6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Pointers

Pointer: a memory address (where a data object is stored).

Explicit pointer types and the ability to store and manipulate pointers (memory addresses) are an important and relatively unique feature of C.

Java **references** are pointers, but references cannot be explicitly manipulated in Java.

A pointer type is specified with a **base type** plus ***** (asterisk).

E.g., “`int*`” specifies a pointer to an `int` (an “int pointer”).

This means the address of a memory location that is the first (lowest numbered) byte of the storage for an `int`.

Pointers (contd.)

Pointer variable declarations:

```
int *ip; //ip contains a pointer to an int object
char *s; //s contains a pointer to a char object
```

(`char*` is most commonly used for a *string* rather than to create a pointer to a single `char`—see the Strings lecture.)

Notice that declarations generally are written with the `*` adjacent to the variable not the base type.

This emphasizes that to declare multiple pointer variables, the `*` must be repeated:

```
int *ip1, *ip2, *ip3;
```

Pointer Operators

Two **operators** are associated with pointers:

- * – **dereference operator**
- & – **address operator**

The *dereference operator* is used to retrieve the value stored at a pointer address:

```
int j = *ip;
```

I.e., it takes a `basetype*` and turns it into a `basetype`.

It can also be used on the LHS of an assignment to allow the *value* stored at a pointer address to be changed:

```
*ip = 2;
```

Pointer Operators (contd.)

The *address operator* is used to retrieve the address where a variable's value is stored:

```
int *ip = &i;
```

I.e., it takes a `basetype` and turns it into a `basetype*`.

Pointers Example

Pointer memory example:

```
int i = 1;
```

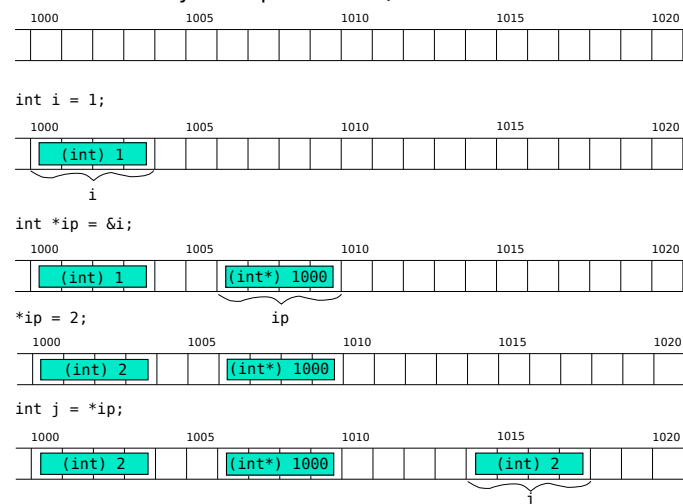
```
int *ip = &i; //ip is set to i's address (ip now "alias" for i)
```

```
*ip = 2; //2 stored in address pointed to by ip (since that  
//address was for i's value, i effectively changed)
```

```
int j = *ip; //j's value set to the value stored at  
//the address pointed to by ip, which was 2  
//same as doing j = i due to ip's value
```

Pointers Example (contd.)

Pointer memory example (boxes are bytes)



void*

It is possible to declare a pointer without specifying the type it points to.

This is done with the `void*` notation:

```
void *p; //p contains a pointer to an unknown type of object
```

A pointer of type `void*` is known as a **void pointer**.

Void pointers can be cast to any other pointer type.

NULL

One special pointer value is the **null pointer (constant)**.

The null pointer is a pointer value that “is guaranteed to compare unequal to a pointer to any object or function.”

In other words, it is a special, *invalid* pointer/address.

The constant **NULL** is most commonly used to explicitly represent the null pointer.

Note however that **NULL** must be provided by a *header include*, such as `stdlib.h`.

It is ultimately defined by a preprocessor macro in `stddef.h`, with an *implementation-dependent value*.

NULL (contd.)

In C, any integer value of 0 (zero) cast to a pointer is interpreted as the null pointer constant: i.e., `(void*)0 == NULL`

However, this does not mean that the value of the null pointer constant is actually (memory address) zero; implementations are free to use any value they desire.

This also does not mean that using 0 (zero) in your code where a pointer is required is good programming practice; use **NULL** to represent the null pointer constant!

The null pointer has some unique characteristics:

- “Conversion of a null pointer to another pointer type yields a null pointer of that type.”
- “Any two null pointers shall compare equal.”

Pointer Problems

Pointers can be a source of problems for beginning C programmers.

A common confusion is misunderstanding when a declaration results in memory being allocated to store a data object.

A declaration like “`double *dp`” allocates space to store a memory address (pointer), but it does *not allocate space to actually store a double*.

Thus, the following code fragment contains a **logic error**, because space has *not* been allocated to store the double 12.34:

```
double *dp;  
*dp = 12.34;
```

This code will not cause a compiler error, however; C will (attempt to) store the double 12.34 at whatever random address `dp` happens to have as its initial value.

Pointer Problems (contd.)

Depending on `dp`'s value, this might cause a **segmentation fault** or it might silently corrupt other data.

One way to fix the above code would be:

```
double *dp = malloc(sizeof(double));
*dp = 12.34;
```

This code allocates **dynamic memory** to store a double, setting `dp` to the address of that memory, before 12.34 is stored.

Pointer variables should generally be declared for one of two reasons only:

1. to use *dynamic memory* (as shown above)
2. to capture the result from a *function that returns a pointer*

Pointer Problems (contd.)

E.g., functions that return pointers:

```
char *strstr(const char *haystack, const char *needle)
struct hostent *gethostbyname(const char *name)
```

Example use:

```
char *last_name = strstr(full_name, " ") + 1;
(returns substring after first space in full_name)
```

If you are declaring pointer variables for some purpose other than the two listed above, think carefully about whether what you are doing is correct!

Arrays

Arrays are a **homogeneous composite/aggregate data type**: a data type that holds *multiple elements* all of *identical type*.

The C syntax for declaring arrays is:

```
int iarr[10];
(creates a 10-element int array named iarr)
```

C requires that the *square brackets* (“`[]`”) in an array declaration come *after the variable*; they cannot come after the type.

So the following are **illegal** C array declarations:

```
int[10] iarr;
int[] iarr = {1,2,3,4,5};
```

Arrays (contd.)

The C syntax for accessing array elements is:

```
for (int i = 0; i < 10; i++)
    iarr[i] = i * 2;
```

Array **indexing** is *zero-based*.

Array variables *cannot be assigned to* (cannot be **lvalues**), so **initialization** in definitions are useful/common.

Array initialization is done using **brace notation**:

```
int iarr[5] = {1,2,3,4,5};
(equivalent to: iarr[0] == 1, ..., iarr[4] == 5)
```

When initializing an array in the declaration, the array size need *not* be specified:

```
int iarr[] = {1,2,3,4,5};
(creates a 5-element array with initial values as above)
```

Variable-Length Arrays

Originally, the size of a C array had to be declared with a *constant expression* only, meaning the array size had to be *known at compile time*.

C99 added **variable-length arrays** (VLAs): array size can be specified with an expression that involves *variables* or *parameters*, so array size can be *determined at runtime*.

The introduction of VLAs into C means that some uses of **dynamic arrays** could be replaced with variable-length arrays.

Note: due to implementation and efficiency issues, VLAs were made *optional* in **C11**!

While VLA seems useful, it will likely be more efficient to use a constant-sized array of *maximum required size* than to use a VLA (unless the maximum size is huge).

Array Initializers

From C99 and on, **designators** of the form **[index]=** may be used in initializer lists, for partial or out-of-order initialization:

```
int a[6] = {[4] = 29, [2] = 15};
```

When using *designators*, unspecified elements are automatically initialized with the same values as objects that have **static storage duration** (zero/NULL).

So the above is equivalent to:

```
int a[6] = {0, 0, 15, 0, 29, 0};
```

A GCC extension allows a *range* of elements to be initialized to the same value using designators of the form **[first ... last] =**:

```
int widths[] = {[0 ... 9] = 1, [10 ... 99] = 2, [100] = 3};
```

With an unspecified array size, the length of the array when using designators is the highest value specified plus one.

Array Initializers (contd.)

The GCC manual says: “Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an enum type.”

For example:

```
int whitespace[256] = {[' '] = 1, ['\t'] = 1, ['\h'] = 1,
                    ['\f'] = 1, ['\n'] = 1, ['\r'] = 1};
```

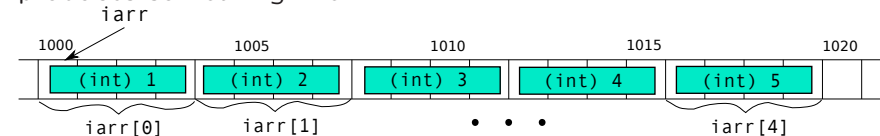
You can also write a series of *.fieldname* and **[index]** designators to specify a *nested subobject* to initialize (see **Structs** lecture):

```
struct point { int x, y; };
struct point parray[10] = {[2].y = yv2,
                          [2].x = xv2,
                          [0].x = xv0};
```

Array Storage Scheme

An array declaration allocates a *contiguous block of bytes in memory*, with array elements stored in successive bytes based on the data type.

For example, the declaration “`int iarr[] = {1,2,3,4,5};`” produces something like:



C effectively represents an array as simply the *memory address* of the *first byte* of the array memory block—i.e., as a *pointer*.

It uses the size of the array's data type to compute the **offset** for each element (i.e., **pointer arithmetic**—see below).

Array Bounds Checking

C does *not maintain array size information* at runtime (like Java).

This means that C *cannot* and does not do **bounds checking** on array accesses.

Unfortunately, many Java programmers come to rely on Java's array bounds checking to help test/debug their programs.

Since C doesn't do this, more effort must be put into designing programs to be correct from the start.

Of course, automatic array bounds checking is a waste of CPU cycles in programs that are correct, and one would not expect C to impose that unnecessary burden on correct programs!

Arrays and Pointers

As already noted, C arrays have a close connection to pointers: *"Except when it is the operand of the sizeof operator or the unary & operator, or is a string literal used to initialize an array, an expression that has type "array of type" is converted to an expression with type "pointer to type" that points to the initial element of the array object..."* (C99 standard)

In other words, an array is effectively represented as a pointer in most contexts.

In fact, an array variable can be used just like a *pointer* variable:

```
int iarr[10];
int *iptr = iarr; //Now iptr[i] == iarr[i]
```

As noted, however, unlike pointer variables, array variables *cannot* be assigned to (**lvalues** cannot be arrays).

Arrays and Pointers (contd.)

When a function has an *array parameter*, what will be passed in the function call is a *pointer (to the first element of the array)*.

This is why when (one-dimensional) arrays are used as *function parameters*, the array size need *not* be specified:

```
void process_array(int iarr[])
```

In fact, the above declaration could just as well be:

```
void process_array(int *iarr)
```

Pointer Arithmetic

Pointer arithmetic refers to explicit manipulation of pointers (memory addresses) with arithmetic operators, e.g.: `iptr + i`

Pointer arithmetic can be used an alternative to array notation: `*(iarr + i)` is equivalent to `iarr[i]`:

```
int iarr[10];
for(int i=0; i<10; i++)
    *(iarr + i) = i+100;
```

Note that in pointer arithmetic, C uses the size of the pointer variable's **base type** to determine the **offset**:
`ptr + n` means `ptr + sizeof(*ptr) * n`.

E.g., if `iptr` (of type `int*`) has value 1234, then then `iptr + 2` is 1242 (`1234 + 8`), assuming `sizeof(int)` is 4.

Pointer Arithmetic (contd.)

Array notation and pointer arithmetic can be used interchangeably.

Looping through argv using array notation:

```
int i = 0;
while (argv[i] != NULL)
    printf("%s\n", argv[i++]);
```

Looping through argv using pointer arithmetic:

```
char **arg = argv;
while (*arg != NULL)
    printf("%s\n", *arg++);
```

Pointers to Arrays

Arrays of pointers are common, e.g., arrays of `char*` (strings):
`char *strs[10];`

Much less common are **pointers to arrays**.

To declare such a pointer, we must add parens around the “*”:

- array of char: `char arr1[10]`
- pointer to array of char: `char (*arr2)[10]`
- array of pointers to char: `char *arr3[10]`
- pointer to array of pointers to char: `char *(*arr4)[10]`

Since an array effectively is a pointer, pointers to arrays are not necessary.

In fact, their notation and usage can be confusing.

Pointers to Arrays (contd.)

When C converts `arr1` to a pointer, both it and `arr2` (above) point to the first byte of a block of 10 bytes.

However, these pointers will have different types:

- `arr1` is `char*`: pointer to the first `char` in the block
- `arr2` is `char(*) []`: pointer to the entire block (array)

Note also that the `arr1` declaration allocates 10 bytes for the array, but the `arr2` declaration allocates memory for a pointer!

We can make `arr2` point to `arr1`:

```
arr2 = &arr1;
```

However, accessing array elements works differently:

```
arr1[i]    versus    (*arr2)[i]
```

Pointers to Arrays (contd.)

A potential advantage of using pointers to arrays is for readability:

- `int (*var) []` – makes it clear we have pointer to an array
- `int *var` – could be a pointer to one `int` or an array

Consider a function that returns an array via a parameter.

This can be done with pointers only, but that the parameter is an array can be made more clear by using pointer to array notation.

However, getting the syntax correct can be a bit tricky.

The next several slides show some example approaches.

Pointers to Arrays (contd.)

Using pointer to array (int's):

```
void create_arr(int (**arrptr) [5])
{
    static int arrnew[5] = {1, 2, 3, 4, 5};
    *arrptr = &arrnew;
    return;
}

int main(int argc, char *argv[])
{
    int (*arrget) [5] = NULL; //clear this is an array!
    create_arr(&arrget);
    ...
}
```

Pointers to Arrays (contd.)

Using pointers only (int's):

```
void create_arr(int **arrptr)
{
    static int arrnew[5] = {1, 2, 3, 4, 5};
    *arrptr = arrnew;
    return;
}

int main(int argc, char *argv[])
{
    int *arrget = NULL; //know is array only from name!
    create_arr(&arrget);
    ...
}
```

Pointers to Arrays (contd.)

Using pointer to array (char*'s):

```
void create_arr(char* (**arrptr) [5])
{
    static char *arrnew[5] = {"a", "b", "c", "d", "e"};
    *arrptr = &arrnew;
    return;
}

int main(int argc, char *argv[])
{
    char* (*arrget) [5] = NULL; //clear this is an array!
    create_arr(&arrget);
    ...
}
```

Pointers to Arrays (contd.)

Using pointers only (char*'s):

```
void create_arr(char ***arrptr) //joined the "triple-star club"!!
{
    static char *arrnew[5] = {"a", "b", "c", "d", "e"};
    *arrptr = arrnew;
    return;
}

int main(int argc, char *argv[])
{
    char **arrget = NULL; //know is array only from name!
    create_arr(&arrget);
    ...
}
```

Pointers to Arrays (contd.)

Finally, note that this functionality cannot be achieved using arrays only, even though they are effectively pointers:

```
void create_arr(int (*arrptr)[5])
{
    static int arrnew[5] = {1, 2, 3, 4, 5};
    *arrptr = arrnew; //would be assignment to array, not allowed
    return;
}

int main(int argc, char *argv[])
{
    int arrget[5]; //uninitialized array
    create_arr(&arrget); //&arrget is same address as just arrget,
                        //so not address that we want to change
    ...
}
```

C Basics 6: Strings

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. **Strings**
 - strings are char arrays
 - null char sentinel
 - comparing strings
 - the C String Library
 - strings vs. char's
7. Library I/O

C Basics 6: Strings

8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Strings

C does *not* have a *built-in* string type (like Java does).

Instead, a “**C string**” is:

- an *array* of `char` (characters)
- terminated by the **null char**, `'\0'` (a **sentinel**)

Because they are `char` arrays, and array variables are effectively pointers, *strings are effectively represented as pointers*.

String types can be denoted as either arrays or pointers:

`“char str[]”` or `“char *str”`

It is more common to see the `“char*”` type specification for function parameters.

Strings (contd.)

To allocate memory for a string of a particular length, we use notation like:

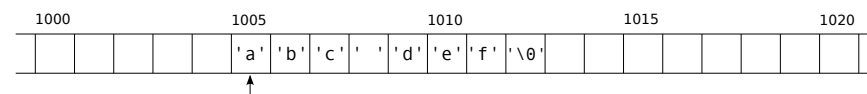
```
char str[4]; (can hold a three-char string + '\0')
```

String constants can be denoted with double-quote notation:

```
"abc def"
```

(same as in Java)

The constant `"abc def"` is effectively the *address* of a block of 8 bytes of memory, each byte holding a `char`:



Comparing Strings

Because strings are not a built-in type in C, you *cannot* use the **equality operator**, `==`, to test whether two strings are the same (contain the *same sequence of characters*).

Instead, you must use a string library function like `strcmp()`.

"`if (str1 == str2)...`" is *syntactically valid*, so it will not cause compiler errors, but it is *almost never what is wanted*.

What `==` will do is test whether the two strings are at exactly the *same memory address*—i.e., the *same string object*.

This is exactly what happens in Java with *reference types*, which is why you use `.equals()` or `.compareTo()` when comparing `String`'s.

The C String Library

Strings typically must be manipulated using library functions from the *C String Library*:

- `strlen` – length of a string (not counting final `'\0'`)
- `strcmp`, `strncmp` – *compare* two string to determine if *equal* or their *lexicographic order*
- `strchr` – determine if a `char` occurs in a string
- `strcat`, `strncat` – *concatenate* two strings (does *not allocate* dynamic memory for result)
- `strstr` – determine if a string is a *substring* of another string
- `strcpy`, `strncpy` – *copy* a string (does *not allocate* memory)
- `strdup`, `strndup` – *copy* a string (*does allocate* memory)
- `strtok` – *parse* a string into tokens/substrings

The C String Library (contd.)

Note that in order to use the C String Library functions, a program must contain the following header include:

```
#include <string.h>
```

One thing to note about the string library functions is that there is not a *substring* function.

Substrings are created using *pointer arithmetic* and `strcpy()`:

```
char full_name[] = "John Q. Smith";
char *last_name = full_name + 8; //uses full_name array

char new_last[6]; //create space to hold separate last name
strcpy(new_last, full_name + 8); //copy last name over

char middle_init[3];
strncpy(middle_init, full_name + 5, 2);
middle_init[2] = '\0'; //Must make middle_init valid C string
```

String Examples

Example code using strings:

```
//Make a copy of a string:
char str[20];
strncpy(str,"sample string",20);

//Read in a line from terminal:
char line[100];
fgets(line,100,stdin);

//Print out str and line:
printf("str: %s\nline: %s\n\n",str,line);

//Another way to make a copy of a string:
//(Could use strdup() instead of malloc()+strcpy().)
char *linecpy = malloc(strlen(line)+1);
strcpy(linecpy,line);

//Checking if str is "test":
if (strcmp(str,"test")==0)...
```

Strings vs. Characters

Beginning C programmers often have trouble understanding the difference between *strings* and *char's*.

Since a C string is a null-char-terminated *array* of `char`, `"A"` and `'A'` are not equivalent:

- `"A"` is of type `char*`, and is the two-element array: `'A', '\0'`
- `'A'` is of type `char`, and is the single byte/`char` `'A'`

Another source of confusion is the **empty string**, `""`:

- since it is a string, it must be a `char` array;
- it is the *one-element array* containing of just `'\0'`
- `""` is not the same as `'\0'`, however
- `""` and `'\0'` are of different types so not even comparable

Escape Sequences

We have seen that there are a number of **escape sequences** that can be used to represent characters.

These escape sequences can be included in strings to represent the corresponding characters.

They are heavily used to include non-printing characters such as newlines:

```
printf("The value of x is: %d\n", x);
```

C Basics 7: Library I/O

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
 - **C standard library I/O (stdio)**
 - **default I/O streams**
 - **buffering**
 - **unlocked functions**
 - **EOF**
 - **formatted I/O and string formatting**
 - **reading lines**

C Basics 7: Library I/O

8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Library I/O

Input/output (I/O) in C is accomplished using functions from the **stdio** component of the *C Standard Library*.

In order to use library I/O functions and associated symbols, a program must contain the following header include:

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

C **library I/O** is also referred to as **stream I/O** because an open file is referred to as a **stream**.

The **handles** for open files with library I/O are of type FILE* (a pointer to a FILE struct/structure).

A FILE struct contains information required for a *stream*, including its **file descriptor**, buffer (pointer), EOF/error indicators, etc.

Default I/O Streams

Three streams are automatically open for each process:

- **standard input** – symbol `stdin`;
- **standard output** – symbol `stdout`;
- **standard error** – symbol `stderr`.

By default, these three stream will all be associated with the *terminal*, so for example, input from `stdin` will come from the keyboard.

However, these three streams may be associated with other devices by the use of **pipng** and **redirection** in the *shell command* that invokes a program (e.g., “`ls | ./prog > save`”).

Library I/O Functions

Important library I/O functions:

- `fopen` – open a file
- `fclose` – close a stream/file
- `printf` – formatted output to `stdout`
- `fprintf` – formatted output to a stream
- `fscanf` – formatted input from a stream
- `getchar` – read a char from `stdin`
- `fgetc/getc` – read a char from a stream
- `putchar` – write a char to `stdout`
- `fputc/putc` – write a char to a stream
- `fgets` – read a line as a string from a stream
- `fputs` – write a string to a stream

Library I/O Functions (contd.)

Important library I/O functions (contd.):

- `fread` – binary input from a stream
- `fwrite` – binary output to a stream
- `fseek` – reposition read-write pointer/position in file stream
- `fflush` – flush buffer for a stream
- `feof` – check stream end-of-file status
- `ferror` – check stream error status

Library I/O Examples

Example of opening a file, reading a line, writing to file:

```
FILE *fptr = fopen("output.text","w");

char buffer[200];
printf("Enter line to save: ");
fgets(buffer,200,stdin);
int length = strlen(buffer);

fprintf(fptr,"Line is %d characters:\n%s\n",length,buffer);

fclose(fpnr);
```

I/O Buffering

A key property of library I/O is that I/O operations are **buffered** by default.

Being buffered means that reading/writing to disk/terminal is done in “chunks” for efficiency, even if the I/O functions are inputting or outputting individual characters.

C has three types of stream buffering:

- **fully buffered** – bytes are exchanged with the associated file/device in **blocks**
- **line buffered** – bytes are exchanged in **lines**, delimited by **newline chars** (`'\n'`)
- **unbuffered** – bytes are exchanged immediately

I/O Buffering (contd.)

By default, streams associated with *interactive* devices like *terminals* are set to be *line buffered*, while other devices (like files) are set to be *fully buffered*.

The buffering for a stream can be changed with `setvbuf()`:

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
```

- `mode` must be one of:
 - `_IONBF` – unbuffered
 - `_IOLBF` – line buffered
 - `_IOFBF` – fully buffered

I/O Buffering (contd.)

Buffering means that when an output function like `fprintf()` is called, data will first be transferred to a buffer associated with the stream, and only under the right conditions will data from the buffer be output to the target file/device.

One consequence of buffering is that *prompts* that do not end with a *newline* may not be printed on the terminal immediately, causing confused output from a program.

The library function `fflush()` can be used to flush buffered data:

```
int fflush(FILE *stream)
```

- for output streams: all buffered data written to the target
- for input streams: all unconsumed buffered data discarded

Unlocked I/O Functions

I/O functions like `fgetc()/getc()` and `fputc()/putc()` look like they would be very inefficient since they handle single chars at a time.

However, they are actually fairly efficient because standard buffering means that these functions simply move a single char between program memory and an I/O buffer, they do not require a disk access.

Unfortunately, the addition of **Threads** to Linux/UNIX resulted in the above functions becoming much slower than they once were, in order to be **thread safe**.

Thread safe here means that the functions can be executed **concurrently** by *multiple threads*, while behaving as if only a single thread of execution is accessing the *stream*.

Unlocked I/O Functions (contd.)

Here is how the **GNU glibc manual** describes the issue: “The POSIX standard requires that by default the stream operations are *atomic*. I.e., issuing two stream operations for the same stream in two threads at the same time will cause the operations to be executed as if they were issued sequentially. The buffer operations performed while reading or writing are protected from other uses of the same stream. To do this each stream has an *internal lock object* which has to be (implicitly) acquired before any work can be done.”

“The locking operations (explicit or implicit) *don't come for free*. Even if a lock is not taken the cost is not zero. The operations which have to be performed require memory operations that are safe in multi-processor environments. With the many local caches involved in such systems *this is quite costly*.”

Unlocked I/O Functions (contd.)

“So it is best to avoid the locking completely if it is not needed—because the code in question is never used in a context where two or more threads may use a stream at a time. This can be determined most of the time for application code; for library code which can be used in many contexts one should default to be conservative and use locking.”

“There are two basic mechanisms to avoid locking. The first is to use the **_unlocked** variants of the stream operations. The POSIX standard defines quite a few of those and the GNU C Library adds a few more. These variants of the functions behave just like the functions with the name without the suffix except that they do not lock the stream. *Using these functions is very desirable since they are potentially much faster.*”

Unlocked I/O Functions (contd.)

“This is not only because the locking operation itself is avoided. More importantly, functions like `putc` and `getc` are very simple and traditionally (before the introduction of threads) were implemented as macros which are very fast if the buffer is not empty. With the addition of locking requirements these functions are no longer implemented as macros since they would expand to too much code. But these macros are still available with the same functionality under the new names `putc_unlocked` and `getc_unlocked`. This possibly huge difference of speed also suggests the use of the `_unlocked` functions even if locking is required...locking then has to be performed in the program....”

Testing with a program that counts lines in a file (like “`wc -l`”) found that using `getc_unlocked()` instead of `getc()/fgetc()` reduced the runtime by *more than 50%* (i.e., ran more than twice as fast).

Unlocked I/O Functions (contd.)

POSIX standard `_unlocked` functions:

- `getc_unlocked` – read a `char` from a stream
- `getchar_unlocked` – read a `char` from `stdin`
- `putc_unlocked` – write a `char` to a stream
- `putchar_unlocked` – write a `char` to `stdout`

Glibc (the GNU C library, used in Linux) includes several additional *non-standard* `_unlocked` functions, including:

- `fgets_unlocked`
- `fputs_unlocked`
- `fread_unlocked`
- `fwrite_unlocked`

EOF

An important *stdio constant symbol* is `EOF`.

`EOF` is a value that is returned by some `stdio` functions to indicate that **end-of-file** was encountered *or* that there was an *error*.

The two conditions can be distinguished with `feof()/ferror()`:

```
int feof(FILE *stream)
int ferror(FILE *stream)
```

It is critical to understand that `EOF` is a value returned by some `stdio` functions—it is *not* a character/`char` (`'EOF'` is illegal).

Files *do not* have `EOF` or any other character as a **sentinel**.

Note that `EOF` is an `int`, typically defined as `-1` (but test for `EOF`).

EOF (contd.)

The fact that EOF is an `int` and typically negative can lead to unexpected problems with **portable code**.

Whether `char` is signed or unsigned is *left to the implementation*, and can vary from one architecture to another.

This means that if we assign EOF (-1) to a `char`, it might be interpreted as -1 or it might be interpreted as 255.

To avoid this problem, always use `int` vars to capture the “character” return value of functions like `fgetc()` that can return EOF.

Formatted I/O

Among the most used library I/O functions are the “formatted output” functions, including:

```
int printf(const char *format, ...)
int fprintf(FILE *stream, const char *format, ...)
```

The `format` parameter is a **format string**: a string constant that contains **format directives**.

Format directives:

- ordinary characters – copied directly to output, e.g., “value:”
- escape sequences – for special characters, e.g., ‘\n’ (newline)
- **conversion specifications** – start with %, consume an argument, specify how to format argument, e.g., %d

Formatted I/O (contd.)

The most important part of a *conversion specification* is the **conversion specifier**: a single character that determines how the argument is to be formatted.

The most used conversion specifiers are:

- `d, i` – `int` argument formatted in decimal
- `f, F` – `double` argument formatted in decimal notation style [-]ddd.ddd
- `e, E` – `double` argument formatted in decimal notation style [-]d.ddde±dd
- `s` – `char*` string `char`’s written up to ‘\0’
- `c` – `int` converted to `unsigned char`, output ASCII character

Formatted I/O (contd.)

A *conversion specification* can also contain multiple *optional components* between the % and the *conversion specifier* character.

Optional components include (in the following order):

- zero or more **flags** (`#`, `0`, `-`, *space*, `+`, `'`)
- minimum **field width**
- **precision**
- **length/type modifiers** (`h`, `hh`, `l`, `ll`, etc.)

Examples:

- `%8.2f` – print floating point in 8-wide field, 2 digits to right of decimal point
- `%05d` – print integer in (minimum) 5-wide field, padding with 0’s on left if less than 5 digits

Formatted I/O (contd.)

Formatted *input* functions include:

```
int scanf(const char *format, ...)
int fscanf(FILE *stream, const char *format, ...)
```

While the format conversion specifications are simpler, the basic structure is similar.

For more detailed information see “man fprintf” or “man fscanf”.

Formatting Strings

The I/O library provides related functions for formatting strings:

```
int sprintf(char *str, const char *format, ...)
int snprintf(char *str, size_t size, const char *format, ...)
```

These are generally a better choice for constructing complex strings than using string library functions such as `strncat`.

Example of constructing a file path string:

```
char filepath[PATH_MAX];
sprintf(filepath, "%s/out%d.data", getcwd(NULL,0), count);
```

Just remember that these functions *do not automatically allocate space* for the resulting string; you must do that ahead of time as with an array or `malloc()`.

Reading Lines

It not be too suprising to find out that C does *not* have a function that can read an *arbitrary length line* in as a string.

The closest function it does provide is `fgets`:

```
char *fgets(char *s, int size, FILE *stream)
```

`fgets()` will read in a line as a string, but you must provide the string memory of the *required size*:

“`fgets()` reads in at most one less than `size` characters from `stream` and stores them into the buffer pointed to by `s`. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte (`'\0'`) is stored after the last character in the buffer.”

Reading Lines (contd.)

If you know the maximum length of a line, `fgets()` works well:

```
char line[102]; //100-char line + newline + null-char
fgets(line,102,stdin);
```

If you do not, a complex loop and **dynamic memory** are required:

```
int incr = 10;
int total = incr + 1;
char *line = malloc(total);
int start = 0;
while (fgets(line+start,incr+1,stdin)!=NULL &&
      line[strlen(line)-1] != '\n') {
    start = total-1;
    total += incr;
    realloc(line,total);
}
```

getline() and getdelim()

While C does not provide functions that support reading lines of arbitrary length, two functions were added to POSIX (2008) that do (so are available on Linux/UNIX systems).

They are: `getline()` and `getdelim()`:

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream)
ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *stream)
```

- `getline()` reads an entire line from `stream`, storing the address of the buffer containing the text into `*lineptr`.
- `getdelim()` works similarly except that `delim` argument can specify a delimiter (other than newline).
- The buffer ends up null-char terminated and includes the newline/`delim` character (if one was found before file-end).

getline() and getdelim() (contd.)

They can allocate dynamic memory themselves, or accept pointers to already allocated dynamic memory (possibly expanding it):

- If `*lineptr` is NULL and `*n` is 0 when called, a new buffer will be allocated for storing the line
- Alternatively, `*lineptr` can contain a pointer to a `malloc()` allocated buffer of `*n` bytes. If the buffer is not large enough to hold the line, it will be resized (with `realloc()`). This may result in `*lineptr` and `*n` being changed to reflect updated address and size.
- If new buffer is to be allocated automatically, it must eventually be `free()`'d by the user program. Note that this is required even if `getline()/getdelim()` fails.

getline() and getdelim() (contd.)

Return values:

- Returns indicate the number of characters read, including the delimiter character, but not including the terminating null char.
- On failure to read a line (including file-end), -1 is returned; `errno` is set on error.

Though not technically part of the C standard I/O library, must `#include <stdio.h>` to use these functions.

You will probably also require a **feature test macro**, such as:

```
#define _POSIX_C_SOURCE 200809L
```

or

```
#define _GNU_SOURCE
```

getline() and getdelim() (contd.)

Example use of `getline()` from man page:

```
int main(void)
{
    FILE *stream;
    char *line = NULL;
    size_t len = 0;
    ssize_t read;

    stream = fopen("/etc/motd", "r");
    if (stream == NULL)
        exit(EXIT_FAILURE);

    while ((read = getline(&line, &len, stream)) != -1) {
        printf("Retrieved line of length %zu :\n", read);
        printf("%s", line);
    }

    free(line);
    fclose(stream);
    exit(EXIT_SUCCESS);
}
```

getline() and getdelim() (contd.)

The “`char **lineptr`” parameter declaration can be confusing.

It is a *pointer to a string* (`char*`) because this is how these functions pass back the buffer containing the line—i.e., it is a *string reference parameter* (in C++ terminology).

In the above example code, `line` is ultimately to contain the read line—i.e., to be a pointer to a `char` array containing the line.

This means the functions must make `line` point to the dynamic array they have newly allocated (holding the read line).

To do this, they must be given the address where `line`'s value (a pointer to `char` array) is stored, so they can set it to a new value: the pointer to the new dynamic array.

C Basics 8: Errors and Error Handling

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. **Errors and Error Handling**
 - **error returns**
 - **error checking code structure**
 - **stdio errors**
 - **errno**
 - **error messages**

C Basics 8: Errors and Error Handling

9. Functions and Parameter Passing
10. Multidimensional and String Arrays
11. Dynamic and Static Memory
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Error Returns

C does not have an **exception mechanism**, so errors are indicated by the **return value** of a function, typically one of:

- NULL – if the return is a pointer/string;
- -1 – if the return is an integer;
- EOF – for char/int return values in stdio;

Calls to functions will often need to be *wrapped* with an if that tests whether an error occurred and takes appropriate action:

```
if ((fptr = fopen(file,"r")) == NULL) {
    fprintf(stderr,"Error opening file %s\n",file);
    exit(EXIT_FAILURE);
}
```

Error Checking Code Structure

Because so many calls must be checked, we generally *test for the error return* and then print a message plus exit/return:

```
//Open file:
if ((fptr = fopen(file,"r")) == NULL) {
    fprintf(stderr,"Error opening file %s\n",file);
    exit(EXIT_FAILURE); }

//Read char from file:
if ((next = fgetc(fptr)) == EOF && ferror(fptr)) {
    fprintf(stderr,"Error reading from file %s\n",file);
    exit(EXIT_FAILURE); }

//Print char to stdout:
if (putchar(next) == EOF) {
    fprintf(stderr,"Error writing to stdout\n");
    exit(EXIT_FAILURE); }

//Done, successfully:
exit(EXIT_SUCCESS);
```

Error Checking Code Structure (contd.)

Notice how *testing for the non-error return* instead results in nested if-then-else's that are very hard to read:

```
if ((fptr = fopen(file,"r")) != NULL)
    if ((next = fgetc(fptr)) != EOF)
        if (putchar(next) != EOF)
            exit(EXIT_SUCCESS);
        else {
            fprintf(stderr,"Error writing to stdout\n");
            exit(EXIT_FAILURE); }
    else {
        fprintf(stderr,"Error reading from file %s\n",file);
        exit(EXIT_FAILURE); }
else {
    fprintf(stderr,"Error opening file %s\n",file);
    exit(EXIT_FAILURE); }
```

stdio Functions and Errors

Functions in stdio may use the same return value for both an error and an *end-of-file* condition (not an error).

E.g., `int fgetc(FILE *stream)`

Return: "the character read as an unsigned char cast to an int or EOF on end of file or error."

Two stdio functions (predicates) can be used to distinguish between end-of-file and error:

- `feof(FILE *stream)` – has stream encountered end-of-file?
- `ferror(FILE *stream)` – has stream encountered an error?

errno

`errno` is **global variable** containing an **error code**:

- it is *zero* when no errors have occurred, or
- it is a *positive integer* identifying the most recent error

When an error occurs during a **system call** (and many library functions make system calls to do their work), the kernel sets `errno`.

The header file `errno.h` defines `errno` and symbolic constants for the set of possible error codes:

e.g., `EACCESS` is the "permission denied" error.

See the man page for `errno.h` for a list of possible errors and their symbolic names.

errno (contd.)

Man pages for system calls and many library functions have an "ERRORS" section that lists the errors that could occur with the call.

In code involving system calls, `errno` can be tested to check for an error.

The OS never zeros `errno` out after an error, so if execution is to continue after an error, it is often necessary to reset `errno` to zero (through assignment).

Error Messages

When printing **error messages**, it is best to provide as much information as possible.

This helps a user understand what occurred and how to fix it.

For example, suppose a call to `fopen()` fails.

Consider a message like:

“Error opening file”

This hardly helps the user; was the wrong file being opened, were there permissions problems, did the file not exist, etc.?

A much better message would be:

“Error opening file test.txt: permission denied”

This makes it clear what file could not be opened, and why.

Error Messages (contd.)

C provides two library functions that will print the **system error message strings** that describe an error code:

```
void perror(const char *s)
char *strerror(int errnum)
```

`perror()` takes a string that becomes the prefix to the message:

```
if ((fptr = fopen(file,"r")) == NULL) {
    perror("Error opening file");
    exit(EXIT_FAILURE); }
```

Message would be like:

“Error opening file: permission denied”

This is generally considered the most basic sort of reasonable error message.

Error Messages (contd.)

`strerror()` provides the system error message string so that it can be used to construct more informative error messages:

```
if ((fptr = fopen(file,"r")) == NULL) {
    fprintf(stderr,"%s: Error opening file %s: %s\n",
            argv[0],file,strerror(errno));
    exit(EXIT_FAILURE); }
```

Message would be like:

“prog: Error opening file test.txt: permission denied”

This is a better message because it identifies the program having the issue, and also identifies the name of the file that could not be opened.

Error Messages (contd.)

In addition to these C standard error reporting functions, GCC/Glib provide two additional, non-standard, error reporting functions:

```
void error(int status, int errnum, const char *format, ...)
void error_at_line(int status, int errnum, const char *filename,
                  unsigned int linenum, const char *format, ...)
```

`error()` makes it a bit easier to accomplish what we showed with the `strerror()` example:

```
if ((fptr = fopen(file,"r")) == NULL)
    error(EXIT_FAILURE,errno,"Error opening file %s",file);
```

`error()` automatically prepends the program name (followed by colon, space) and appends a colon, space, and system error message.

In addition, if its `status` argument is non-zero, it then calls `exit(status)` to terminate the program.

Error Messages (contd.)

`error_at_line()` adds the parameters `filename` and `linenum`.

Its output differs in that after the program name, a colon, the value of `filename`, a colon, and the value of `linenum` get inserted in the output.

The preprocessor values `__FILE__` and `__LINE__` are typically used as the `filename` and `linenum` arguments.

C Basics 9: Functions and Parameter Passing

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling

C Basics 9: Functions and Parameter Passing

9. Functions and Parameter Passing

- **defining functions**
 - **functions vs. procedures**
 - **parameter passing: call-by-value**
 - **array parameters**
 - **pointer parameters**
 - **pointers as reference parameters**
 - **const type qualifier**
10. Dynamic and Static Memory
 11. Multidimensional and String Arrays
 12. Structs (Structures)
 13. Higher-Order Functions
 14. Variadic Functions

15. Multi-File Programs
16. Miscellaneous

Defining Functions

Functions are the the main code modules in C.

In programming language terminology, they are **subroutines**.

C *functions* are similar to **methods** in Java, except that functions are *not* members of classes.

The basic syntax for a function definition is:

return_type function_name (parameter_list) {function_body}

- *return_type* must be a *non-array* object type or else `void`
- *function_name* must adhere to naming rules for *identifiers*
- *parameter_list* is either `void` or a *comma-separated list of parameter_declaration's*
- a *parameter_declaration* specifies the parameter type and its name, similar to variable declarations

Defining Functions (contd.)

While all C functions must have a return type, that type may be `void`, in which case the function does not return an object value.

Such functions are frequently referred to as **procedures** in C literature, however, this term is not used in recent C standards.

Note that **main** is considered a function as well, though it is special in that it does not require a declaration and is the **entry point** for the overall program.

Functions and Arrays/Pointers

C functions do *not* allow (copies of) *arrays* to be passed:

- array *parameters* are automatically converted to equivalent pointer types
- array *return values* are *not* allowed (but pointer types are)

Because of the connection between arrays and pointers, these restrictions have little practical effect.

They do however mean that *array-type parameters* can leave the array size *unspecified*:

- `f(int x[10])` (size specified as 10, but not bounds checked)
- `f(int x[])` (size unspecified)
- `f(int x[*])` (size unspecified, *variable-length array*)

Functions and Arrays/Pointers

All of the above parameter declarations are effectively equivalent to:

- `f(int *x)`

Note that with *multidimensional array* parameters, only the *leftmost dimension* can be left unspecified:

- `f(int x[][20][30])`

The keywords `restrict` and `static` can appear with the `[]`'s of an array parameter:

- `f(int (* restrict x)[10])`
- `f(int x[restrict][10])`

- `f(int x[restrict 10][20])`
- `f(int x[restrict static 10][20])`

`[static]` implies the array has *at least as many elements* as specified by the size expression.

Parameter Passing

C uses a **call-by-value** scheme for passing function parameters: parameters are bound to *copies of the values* of the arguments.

This is the same scheme used by Java and most other modern programming languages.

The key effect of this scheme is that argument variables will not have their values changed by a function, only the local (copy) values change.

Since an array parameter is converted to a pointer, however, an array parameter is bound to a copy of the argument array *address*, so array elements *can be modified* inside of a function.

This also makes it *efficient* to use arrays as function parameters, because the array itself is not copied.

Parameter Passing (contd.)

Consider the following version of `swap()`:

```
void swap(int x, int y)
{
    int temp = x;
    x=y;
    y=temp;
}
```

This does not work:

```
int i=1, j=2;
swap(i,j);
printf("i:%d j:%d\n",i,j); //Oops: still i==1, j==2
```

Parameter Passing (contd.)

In C, the effect of **call-by-reference** can be achieved by using *pointer parameters*:

```
void swap(int *x, int *y)
{
    int temp = *x;
    *x=*y;
    *y=temp;
}
```

Now `swap()` works:

```
int i=1, j=2;
swap(&i,&j);
printf("i:%d j:%d\n",i,j); //Now i==2, j==1
```

Parameter Passing (contd.)

Using *pointer types as function parameters* is very common, but leads to confusion about where memory is being allocated.

For example, the `wait()` system call takes an `int*` parameter because it will change the value of its argument, but an `int` must have been pre-allocated:

```
int status;
wait(&status);
```

On the other hand, a function like `char *getline(FILE *fptr)` could create and return a string, so we can simply do:

```
char *line = getline(fpntr);
```

const Type Qualifier

const is a **keyword** that serves as a **type qualifier**.

This means that it can be added to a type declaration:

```
const int x = 10;
```

Adding the `const` qualifier tells the compiler that a variable (or parameter) is *not to have its value modified* (after it is initialized in the declaration).

If the code attempts to modify the value of a `const` variable, the compiler will generate an *error message*:

```
x = 20; //This line will cause the compiler to complain!
```

const Type Qualifier (contd.)

The only confusing thing about `const` is its use with *pointers*:

- `const int *ip = &i;`
 - `ip` is of type *pointer to const-qualified int*
 - `ip` can be modified (made to point to another `int`), but what it points to cannot be changed (using `ip`)
- `int * const ip = &i;`
 - `ip` is of type *const-qualified pointer to int*
 - `ip` cannot be modified (will always point to `i`), but what it points to (`i`) can be changed
- `const int * const ip = &i;`
 - `ip` is of type *const-qualified pointer to const-qualified int*
 - neither `ip` nor what it points to can be modified

C Basics 10: Static and Dynamic Memory

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing

C Basics 10: Static and Dynamic Memory

10. **Static and Dynamic Memory**
 - **identifiers and scope**
 - **storage durations and storage-class specifiers**
 - **static storage and static variables**
 - **dynamic/heap memory**
 - **memory management functions**
 - **dynamic arrays**
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Identifiers and Objects

Key elements of most programming languages are **identifiers**: names for variables, functions, structs/classes, etc.

The C99 standard says:

“An **identifier** can denote an **object**; a **function**; a **tag** or a **member** of a structure, union, or enumeration; a **typedef** name; a **label** name; a **macro** name; or a macro parameter.”

Object: “region of data storage in the execution environment, the contents of which can represent values. (When referenced, an object may be interpreted as having a particular type.)”

I.e., an *object* is an *instance* of a data type.

Identifiers and Bindings

Associating an entity (object/function/etc) with an identifier is called (**name**) **binding**.

The same *identifier* (name) can denote different *entities* in different contexts or at different points in a program.

Program context disambiguates the entity *type* for each identifier reference.

However, an identifier can have multiple *bindings* of the same entity type within a program.

Again, program context disambiguates the *entity/binding* for each identifier reference.

Identifiers and Namespaces

The same identifier can be used to name different *types* of entities simultaneously if the types are in different **namespaces**.

C has the following namespaces:

- **labels** (for `goto`'s)
- **tags** of structures, unions, and enumerations
- **members** of structures or unions;
(each structure/union has a separate member namespace)
- **ordinary identifiers**:
variables, functions, enumeration constants

Scope

Scope refers to the portion of a program where a particular *binding* to an identifier is *visible* (and so accessible).

E.g., suppose one defines a variable `ivar` as: `int ivar = 10;`
The *scope* of this binding to `ivar` is the portion of the program where you can refer to `ivar` and retrieve the value 10.

The C99 standard says:

“For each different entity that an identifier designates, the identifier is visible (i.e., can be used) only within a region of program text called its **scope**.... There are *four kinds of scopes*: **function**, **file**, **block**, and **function prototype**.”

Scope (contd.)

Scope is determined by the context of an identifier's declaration.

Labels (for `goto`'s) have *function scope*: visible only within the function body.

A **function prototype**'s parameter declarations have **function prototype scope**: visible only within the prototype declaration.

For other identifiers, scope depends on which of two contexts the declaration appears in:

1. outside of any block or function parameter list (“global”)
2. inside a block or within a function's parameter declarations

Scope (contd.)

For these two different cases, scope is as follows:

1. **file scope**: visible to the end of the **translation unit** (file)
2. **block scope**: visible to the “}” that closes the block

It is possible for an identifier (in the same namespace) to have two bindings with *overlapping* scopes.

In this case, one binding's scope, the **inner scope**, will be a strict subset other the other binding's scope, the **outer scope**.

Within the inner scope, the binding in the outer scope is **hidden** by the inner scope binding (i.e., **shadowing/masking**).

Scope (contd.)

Scope Examples:

```
int x = 1; //global definition
int y = 2;

int func(int x) //shadows any x
{
    int z = 3;
    return x + y + z; //func x argument + 2 + 3;
}

int main()
{
    int y = 4; //hides/shadows global y
    int z = 5; //scope is main block
    int w;     //declaration only, scope main block
    {
        int z = 6; //hides z of 5, scope inner block
        w = x + y + func(z); //1 + 4 + func(6)
    }
    w = x + y + func(z); //1 + 4 + func(5) (z back to 5 binding)
    ...
}
```

Linkage

In C, there is another concept called **linkage**, which interacts with scope, particularly with respect to *multi-file programs*.

Together, C scope and linkage determine whether/where an identifier is visible and which binding is referenced, at any particular point in an entire C program.

C has three types of linkage: external, internal, and none.

See lecture #15 for more info: **Multi-File C Programs**.

Storage Durations

In C, **storage duration** refers to the **lifetime/extent** of an object: the portion of program execution during which storage is guaranteed to be reserved for the object.

Each data object in C has one of three *storage durations*:

- **automatic** – allocation/deallocation is *automatic* upon entry/return from the containing function or block
- **static** – allocated permanently in executable, accessible *throughout program execution*
- **allocated** – allocation/deallocation is done *manually* (via calls to *memory management functions*)

An object's storage duration is determined by the *declaration* of its *identifier* (containing *variable*) or by how the object is created.

Storage Durations (contd.)

Automatic duration is the *default* for most variables such as function parameters and local variables.

Static duration is the default for **global variables** (variables declared outside of any function body).

Allocated duration objects must be created with specific functions (e.g., `malloc()`).

Default durations can be overridden with the `auto`, `static`, and `extern` **storage-class specifiers** (more on this later).

Storage Durations (contd.)

The storage duration of an object determines *what part of the executable's address space* it is stored in:

- **automatic** – storage is within the **stack**, *deallocation is automatic* upon return from the containing **stack frame**
- **static** – storage is in **initialized/uninitialized data segments** of the executable (set up by compiler)
- **allocated** – storage is within the **heap**, based on explicit calls to *memory management functions*

Dynamic memory or **dynamic storage** are the terms more commonly used to refer to *allocated storage*.

Storage-Class Specifiers

The C standards use the term **storage-class specifier** to refer to keywords that affect the *storage duration* (and possibly *linkage*) of identifiers/variables.

It is fairly common to see material on C talk about the “**storage class**” of identifiers and objects, typically relating “storage class” to where objects are stored in memory.

The C standards, however, refer minimally and inconsistently to the “*storage class*” of identifiers (and even objects).

Storage-Class Specifiers (contd.)

The storage-class specifiers are:

- **auto** – object will have *automatic storage duration* (rarely used, since default for objects declared within functions)
- **static** – object will have *static storage duration*
- **register** – try to make access to object as fast possible (e.g., store in CPU register if possible)
- **extern** – mainly for *linkage*, results in *static storage duration*

Two also affect **linkage**:

- **static** – sets *internal linkage* for functions and global variables
- **extern** – sets *external linkage* for identifiers

Static Storage/Variables

Objects with **static storage duration** are maintained *throughout program execution*.

A variable has static storage duration if it is:

1. declared outside of any function definition (a “*global*”), or
2. local but declared with the *static storage-class specifier*:

```
static int i = 1;
```

Static storage is allocated by the compiler, in the **data segment**.

If the variable declaration defines an initial value, that value is setup in the executable (**initialized data segment**).

Otherwise, the variable is setup with a default zero/NULL value (**uninitialized data segment**).

Static Storage/Variables (contd.)

Local variables are *automatic duration* by default, which raises two issues in their use:

1. pointers to their values *cannot be returned* from functions (because the storage will be automatically deallocated)
2. their values are not *retained between function calls*

Issue #1 can be addressed in two ways:

1. by returning *static storage* objects
2. by returning *allocated storage* objects (i.e., dynamic memory—discussed shortly)

Issue #2 can be solved only through the use of *static storage*: Declaring local variables to be *static*, will cause their values to be *maintained throughout program execution, and thus between function calls*.

Example: Static Local Variable in Function

Consider a function that needs to keep track of how many times it has been called:

```
void check_count()
{
    //Initialize num_calls:
    //(done once at program initialization)
    static int num_calls = 0;

    //check_count called again, so increment maintained count:
    num_calls++;

    if (num_calls > 10) //E.g., do something if check_count
        ...           //has been called more than 10 times

    return;
}
```

static Keyword

The keyword *static* is a bit confusing, as it has two distinct uses:

- to change the **storage duration** of a variable
- to *limit* the **scope** of functions and global variables

See lecture #15 for more info: **Multi-File C Programs**.

Dynamic Memory

Dynamic memory is the most commonly used term for storage space allocated on the program's **heap**.

In C, dynamic memory means *allocated storage duration* objects.

This storage is called *dynamic* because it can be allocated or deallocated at any point during program execution.

It has **indefinite extent**: it remains until explicitly deallocated (it is not automatically deallocated when a function/block ends).

Allocated storage is created by calling **memory management functions**.

Memory Management Functions

Dynamic memory management involves a set of four functions:

- **malloc** – allocates a number of bytes and returns a pointer to the first byte of the block
- **calloc** – like `malloc`, but allocates space for a number of objects of specified bytes each, plus *zeros memory*
- **realloc** – changes the size of a memory block previously allocated by `malloc`, etc.
- **free** – reclaims a previously allocated memory block

Memory Management Functions (contd.)

Syntax of memory management functions:

```
void *malloc(size_t size)
```

```
void *calloc(size_t num, size_t objsize)
```

```
void *realloc(void *ptr, size_t size)
```

```
void free(void *ptr)
```

- `size` is the number of *bytes* to be allocated
- `num` is the number of objects
- `objsize` is the size in bytes of each object
- `ptr` must be a pointer (address) previously returned by one of these calls
- `void*` return is either a pointer to an appropriate size memory block, else `NULL` on error

`malloc(num * objsize)` is equivalent to `calloc(num, objsize)` except that `calloc` also zeros out the memory block.

Example: Dynamic Memory

Using dynamic memory to return a new string that is a concatenation of two argument strings:

```
char *concat_strings(const char *str1, const char *str2)
{
    //Allocate space for final string:
    char *concat = malloc(strlen(str1) + strlen(str2) + 1);

    strcpy(concat, str1); //Copy str1 into concat
    strcat(concat, str2); //Concat str2 onto end of str1

    return concat;
}
```

(Linux/UNIX provide functions, `strdup()`/`strndup()`, that will internally call `malloc()` to copy a string, but not concat strings.)

Dynamic Arrays

Dynamic memory is often used to provide “**dynamic arrays.**”

Recall that an array is a contiguous block of memory, effectively represented by a pointer to the first byte of the array.

This is exactly what `malloc()` provides, but using heap memory.

Thus, we can use *array notation* with dynamic memory:

```
int *iarr = malloc(sizeof(int)*num_ints);
for (int i = 0; i < num_ints; i++)
    iarr[i] = 100 + i;
```

Not only are the sizes of “dynamic arrays” determined at runtime, but they can be *resized* (using `realloc()`).

Example: Resizing Dynamic Array

Resizing an int array:

```
int *iarr = malloc(sizeof(int)*num_ints);
for(int i = 0; i < num_ints; i++)
    iarr[i] = 100 + i;
...
numints += 10;
iarr = realloc(iarr,sizeof(int)*num_ints);
for(int i = numints - 10; i < num_ints; i++)
    iarr[i] = 100 + i;
```

Whether the array is resized *in-place* or the data has to be copied to a new block is transparent to the user as long as `iarr` is updated with the return value from `realloc()`.

Example: Resizing String

Reading in an arbitrary length line with `fgets()`:

```
char *readinline(FILE *fptr)
{
    int size = 51, start = 0;
    char *line = malloc(size), *result;

    while ((result = fgets(line+start,51,fptr)) != NULL) {
        if (line[strlen(line)-1] == '\n' || feof(fptr))
            break; //End of line/file found so quit reading.
        else {
            start = size - 1;
            size += 50;
            line = realloc(line,size); }
    }

    //If read error, return NULL:
    if (result == NULL)
        return NULL;

    return line;
}
```

Correct Usage of `realloc()`

Since it is possible for the dynamic memory management calls to fail, they must be error checked as usual.

Doing this properly for `realloc()` can be somewhat subtle.

Consider a simple example:

```
char *line = malloc(50); //No error checking here for simplicity
...
if ((line = realloc(line,100)) == NULL)
    free(line); //Runtime error since line will be NULL!
...
```

The problem with the above `realloc()` call is that if it should fail, `line` will get set to `NULL`, and you will lose the pointer to memory block allocated with `malloc()`.

Correct Usage of `realloc()` (contd.)

If your program is going to terminate on such an error then this is fine.

However, if it is going to continue running, as with a server, then you have the makings for a **memory leak** (discussed later).

The correct approach uses a temporary variable:

```
char *line = malloc(50); //No error checking here for simplicity
...
char *temp;
if ((temp = realloc(line,100)) != NULL)
    line = temp;
else
    deal appropriately with realloc call having failed
...
```

Reasons to Use Dynamic Memory

There are three reasons for using dynamic memory:

1. to allow a data object to exist beyond the **scope** in which it is created (e.g., outside a function)
2. to create *arrays* whose *size is determined at runtime*
3. to provide the ability to *resize an array* (including strings)

Because dynamic memory remains accessible until it is explicitly deallocated (*free'd*), it can be used to provide storage that exists beyond the span of a single function/block.

A key example of when this is required is when a *function returns a pointer type*, such as a string (`char*`).

The only other pointers (memory addresses) that are valid to return from functions are those for *static storage duration* objects.

Reasons to Use Dynamic Memory (contd.)

Originally, the size of a C array had to be declared with a *constant expression* only, meaning the array size had to be *known at compile time*.

C99 added **variable-length arrays** (VLAs):

the array size expression could involve variables/parameters, and so could be *determined at runtime*.

This meant that some uses of dynamic arrays could be replaced with variable-length arrays.

However, variable-length arrays *do not have indefinite extent*: storage for variable-length arrays is allocated *at the point of declaration* and *deallocated when the containing block is exited*.

Reasons to Use Dynamic Memory (contd.)

Furthermore, while the size of a variable-length array can be determined at runtime, once the storage has been allocated, the *array size cannot be changed*.

Dynamic arrays, on the other hand, can be *resized* as needed, using `realloc()`.

It is important to be aware, however, that resizing a dynamic array may result in the array contents having to be *copied to a new memory block* (by `realloc()`).

Since doing this frequently could waste significant CPU cycles, it may be more efficient to use a standard array that is large enough for all possibilities.

Costs of Dynamic Memory

When deciding whether to use dynamic memory it is important to understand that the memory management functions all add *runtime overhead* to a program.

Heap memory is organized so as to avoid **fragmentation** when repeatedly allocating and deallocating different sized blocks.

Thus a call to `malloc()` will require finding the best fit block size and updating the list of used/free blocks.

A call to `free()` will require verifying the memory pointer is valid and updating the appropriate list of used/free blocks.

Because there is *runtime overhead* to using dynamic memory, one should *use an array instead of dynamic memory if practical*.

Manual Memory Management

In C, *management of dynamic memory* is completely **manual**.

That is, allocating and reclaiming dynamic memory must be done via *explicit function calls*.

In Java, allocation is done manually with the `new` operator, but memory is reclaimed automatically via **garbage collection**.

Having automatic reclamation of dynamic memory avoids serious *memory management errors!*

Memory Management Bugs

Memory management errors can result in two key bugs:

- **memory leaks**
- **dangling pointers**

A **memory leak** occurs when `malloc()` (or related functions) is repeatedly called without calling `free()`.

This causes heap memory and so the overall program address space to grow over time, potentially *exhausting virtual memory*.

A **dangling pointer** occurs when a memory block gets `free'd` while a pointer to the block continues to be used.

These bugs are common in complex C programs because it can be very difficult to understand when it is correct to free dynamic memory if pointers are passed among functions.

C Basics 11: Multidimensional and String Arrays

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory

C Basics 11: Multidimensional and String Arrays

11. Multidimensional and String Arrays
 - multidimensional array definitions and access
 - arrays of arrays
 - linearization: row major order
 - arrays and pointers
 - subarrays and function parameters
 - string arrays: multidimensional vs. `char*` arrays
 - jagged arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Multidimensional Arrays

Arrays can be **multidimensional**:

```
int array_2d[10][20];
```

`array_2d` is a 10×20 array (10 “rows” with 20 “columns” each).

An multidimensional array element can be accessed as:

```
array_2d[1][19] = 20;
```

In C, “multidimensional arrays” are technically *arrays of arrays*.

`array_2d` is a *10-element array*, each element of which is a *20-element array* of `int`.

Multidimensional Arrays (contd.)

Multidimensional arrays are not limited to two dimensions:

```
int array_3d[10][20][30];
```

We would normally call this a $10 \times 20 \times 30$ array.

In C though, it is technically an 10-element array, each element of which is a 20-element array, each element of which is a 30-element array of `int`.

Linearization: Row Major Order

As with one-dimensional arrays, a multidimensional array is stored in a *block of sequential bytes*.

Array elements are *linearized* using **row major order**:

```
int a2d[3][5] = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
```

defines an array with *three rows* each with *five columns*.

The fifteen array elements are stored linearly in memory as:

```
a2d[0][0], a2d[0][1], a2d[0][2], a2d[0][3], a2d[0][4],  
a2d[1][0], a2d[1][1], ..., a2d[1][4], ..., a2d[2][4]
```

(Note: rightmost index/subscript changes fastest.)

Arrays and Pointers

As noted previously, there is a close connection between arrays and pointers in C.

An array identifier is effectively converted to a pointer to the first byte of the array memory block.

The C99 standard says:

- “E1[E2] is identical to *((E1)+(E2)).”
- “Successive subscript operators designate an element of a multidimensional array object.”
- “If E is an n -dimensional array...with dimensions $i \times j \times \dots \times k$, then E...is converted to a pointer to an $(n - 1)$ -dimensional array with dimensions $j \times \dots \times k$.”

Arrays and Pointers (contd.)

Because of the array-pointer connection, there are numerous ways to get the value of element i - j in `a2d[3][5]`:

```
a2d[i][j]  
*(a2d[i] + j)  
*(&a2d[0][0] + 5*i + j)  
*((a + i))[j]  
*((*(a + i))+j)
```

Arrays and Pointers (contd.)

The connection between multidimensional arrays and pointers can become fairly complicated to understand.

For example, the C99 standard has the following:

“Consider the array object defined by the declaration

```
int x[3][5];
```

Here `x` is a 3x5 array of ints; more precisely, `x` is an array of three element objects, each of which is an array of five ints. In the expression `x[i]`, which is equivalent to `*((x)+(i))`, `x` is first converted to a pointer to the initial array of five ints. Then `i` is adjusted according to the type of `x`, which conceptually entails multiplying `i` by the size of the object to which the pointer points, namely an array of five int objects. The results are added and indirection is applied to yield an array of five ints. When used in the expression `x[i][j]`, that array is in turn converted to a pointer to the first of the ints, so `x[i][j]` yields an int.”

Subarrays

Because multidimensional arrays are truly arrays of arrays, it is possible to refer to the *subarrays* using partial subscripts.

Consider again the array `int a2d[3][5]`.

The *rows* in `a2d` can be denoted as:

```
a2d[0], a2d[1], a2d[2]
```

Each row is of type: 5-element array of `int`.

(Note that if one wants to be able to access subarrays, dimension *order* is critical when defining a multidimensional array.)

Arrays as Function Parameters

When using multidimensional arrays as *function parameters*, the *leftmost dimension* need *not* be specified in the definition:

```
void process_2d(int iarr_2d[][20])
void process_3d(int iarr_3d[][20][30])
```

Even without knowing that the size of the leftmost dimension, the appropriate offset can be computed for any array element:

```
iarr_2d[i][j]: *(&iarr_2d[0][0] + 20*i + j)
iarr_3d[i][j][k]: *(&iarr_3d[0][0][0] + 20*30*i + 30*j + k)
```

(Of course without knowing all dimensions it is not possible to tell whether an element is “out of bounds,” but C does not check this anyway!)

Arrays of Strings

It is common to want to manipulate a set of related strings, and a typical way to store these strings is as an *array of strings*.

Since a string is an array of `char` in C, an array of strings is really an array, each of whose elements is an array of `char`:

```
char strarray[][]
```

An array of 10 strings of 20 characters each can be declared:

```
char strarr[10][21];
```

Such an array can be initialized using special notation:

```
char strarr[10][21] = {"first string", "second string",...};
```

It is possible to use a *single array index* to access each (sub)string:

```
strcpy(strarr[1], "test string");
```

Arrays of Strings (contd.)

Because a character array is effectively a pointer, another way to declare an array of strings is:

```
char *strarr[10];
```

An important difference between this declaration and the one above is that this one *does not allocate any space to actually store the strings*.

That would have to be done via something like:

```
for (int i=0; i<10; i++)
    strarray[i] = malloc(21);
```

This would be slower than using arrays.

On the other hand, it would allow the array to hold strings with very different lengths without having to allocate space for the longest possible string length for every string element.

Arrays of Strings (contd.)

We have already seen an array of strings with `main`, where `argv` is an array of the arguments as strings (array of `char*`):

```
main(int argc, char *argv[])
```

Because of the array-pointer connection, we can also do:

```
main(int argc, char **argv)
```

(`char**` is read *right to left*: `argv` is a pointer to `char*`, or array of `char*`, and `char*` is a pointer to `char`, or array of `char`.)

Using pointer arithmetic to print command line arguments:

```
int main(int argc, char **argv)
{
    char **args = argv;
    while(++args != NULL)
        printf("%s\n",*args);

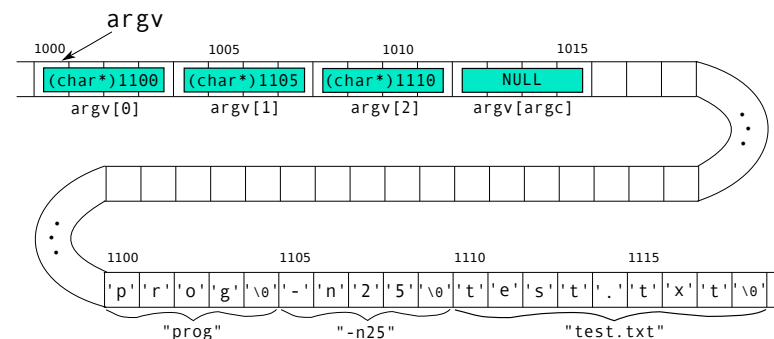
    return EXIT_SUCCESS;
}
```

Arrays of Strings (contd.)

Earlier, we showed `argv` for the command “`prog -n25 test.txt`” as an array of strings:

"prog"	"-n25"	"test.txt"	NULL
argv[0]	argv[1]	argv[2]	argv[argc]

But `argv` is actually an array of arrays of `char`:



Initializing Arrays of Strings

An array of strings can be initialized as:

```
const char *strarr[] = {
    "First entry",
    "Second entry",
    "Third entry",
};
```

Here `strarr[0]` is a pointer to the *string constant* "First entry".

Note that while e.g. `strarr[0]` can be changed to point to a different `char*`, modifying the elements (`char`'s) of the string constants can cause problems, hence the `const` declaration (so compiler will catch).

Jagged Arrays

A multidimensional array defined as `a2d[3][5]` is a **rectangular array**: each subarray is of exactly the same size.

By contrast, the array of strings above, `strarr`, is a **jagged array**: subarrays differ in size: `strarr[0]` is 12 `char`'s, `strarr[1]` is 13.

Jagged arrays of any type can be created, such as a jagged 2D `int` array of three rows (and varying columns):

```
int row0[] = {0,1,2};
int row1[] = {1,2,3,4,5};
int row2[] = {4,5};
int *jagged2d[] = { row0, row1, row2 };
```

C Basics 12: Structs (Structures)

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory

C Basics 12: Structs (Structures)

11. Multidimensional and String Arrays
12. **Structs (Structures)**
 - **structs vs. classes**
 - **defining structs**
 - **accessing members/fields**
 - **struct initialization**
 - **typedef**
 - **unions**
 - **structs, unions, and arrays**
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Structs/Structures vs. Classes

The main **heterogeneous composite data type** in C is the **struct** (short for **structure**).

Structs are similar to OOP **classes**:

- define a new **type**
- composed of multiple **members** (also often called **fields**)
- the component *members* can be of different types

Structs differ from classes:

- no **inheritance** (subclass/superclass structure)
- no **encapsulation** of **methods**
- no **information hiding** capabilities (e.g., `private`)

Defining a Struct

The syntax for defining a struct is:

```
struct [struct_name] {
    member_definition1;
    member_definition2;
    ...
    member_definitionn;
} [variable(s)];
```

The elements inside []'s are *optional*.

Each `member_definition` is similar to a standard C variable definition, with a *type* and a *name (identifier)*, which define the type and name of the next struct member:

```
int count;
char name[20];
```

Defining a Struct (contd.)

For example, a definition of a **named struct** to hold addresses:

```
struct address {
    char street[50];
    char city[25];
    char state[2];
    int zip;
};
```

An address struct variable `user_addr` can then be declared as:

```
struct address user_addr;
```

Note that the name of the resulting type is actually “struct address” rather than just “address.”

Defining a Struct (contd.)

Struct variable(s) can be declared as part of a struct definition:

```
struct address {
    char street[50];
    char city[25];
    char state[2];
    int zip;
} user_addr1, user_addr2;
```

These variables can still be “initialized” with **initializer notation**, but a *cast* is required (technically we are copying a **compound literal** rather than initializing a struct):

```
user_addr1 = (struct address){"Lincoln Ave",...};
```

Instead, we could set each field separately:

```
user_addr1.street = "Lincoln Ave";
...
```

Unnamed Structs

As shown earlier, the `struct_name` in a definition is *optional*.

A struct without a name is called an **unnamed struct**.

An *unnamed struct* along with variable definitions can be used if the struct type is not required again in the program:

```
struct {
    char street[50];
    char city[25];
    char state[2];
    int zip;
} user_addr1, user_addr2;
```

Nested Structs

The type of a struct member can be any type, including another struct.

Structs with member(s) that are structs, are called **nested structs**:

```
struct address {
    ...
};

struct person_record {
    char name[100];
    struct address mailing_address;
    int age;
};
```

It is important to understand the syntax for *initialization* and *member access* with nested structs (see below).

Nested Structs (contd.)

One limitation on members' types is that a struct *cannot* contain a member of the *struct's own type* (i.e., it cannot be "recursive").

It can, however, contain a member that is of type *pointer* to the struct's own type (so struct objects can be linked together):

```
struct address {
    char street[50];
    char city[25];
    char state[2];
    int zip;
    struct address *next_addr;
};
```

Nested Structs (contd.)

Unnamed structs can be used in nested structs but add little:

```
struct person_record {
    char name[100];
    struct {
        char street[50];
        char city[25];
        char state[2];
        int zip;
    };
    int age;
};
```

The members of the nested struct (street, etc.) are treated just like non-nested members (e.g., name) for access.

Dot/Member Operator

Structure members/fields are accessed using the **dot operator** (**member operator**): `user_addr.zip`

This is similar to Java/C++ notation for accessing a *class member*.

Example:

```
...
struct address bob_addr;
bob_addr.street = "101 Main St.";
bob_addr.city = "Carbondale";
bob_addr.state = "IL";
bob_addr.zip = 62901;
...
printf("City: %2s", bob_addr.city);
```

Dot/Member Operator (contd.)

Precedence/associativity allows simple *chaining* of dot operators with *nested structs*:

```
...
struct person_record {
    char name[100];
    struct address mailing_address;
    int age;
};

struct person_record bob_rec;
...
bob_rec.name = "Bob Smith";
bob_rec.mailing_address.zip = 62901;
...
int bob_zip = bob_rec.mailing_address.zip;
```

Struct Pointers

If a *function parameter* is a struct type, C's **pass-by-value** approach will result in actually *copying the struct* argument.

This could cause function calls to become very costly with large struct arguments.

For this reason, when struct parameters are required, it is *most common to use parameters that are struct pointers*.

For example:

```
int store_addr(const char *path, struct address *addrptr);
```

All that gets copied now is the *pointer* (memory address) of the struct rather than the entire struct.

Arrow Operator

It is common to use struct pointers, but because of precedence, accessing a member of a struct pointer requires using parens:

```
(*addrptr).zip
```

The **arrow operator** simplifies accessing fields of struct pointers:

```
addrptr->zip
```

(note that the "arrow operator" is two characters: - + >)

Example of function using arrow operator:

```
void print_address(struct address *addrptr)
{
    printf("%s\n%s,%s %5d\n", addrptr->street, addrptr->city,
          addrptr->state, addrptr->zip);
}
```

Arrow Operator (contd.)

The arrow operator can also be used on the LHS of *assignments*:

```
void change_zip(struct address *addrptr, int newzip)
{
    addrptr->zip = newzip;
}
```

Note that because a struct *pointer* is being passed, the function can modify the values of the struct argument's members!

Struct Initialization

Structs can be initialized using *brace notation* similar to that used for array initialization:

```
struct address bob_addr =
    {"101 Main St.", "Carbondale", "IL", 62901};
```

Members are filled in sequentially, so a nested struct could be initialized in two ways:

```
struct person_record bob =
    {"Bob Smith", {"101 Main St.", "Carbondale", "IL", 62901}, 52};
```

Or simply:

```
struct person_record bob =
    {"Bob Smith", "101 Main St.", "Carbondale", "IL", 62901, 52};
```

Struct Initialization (contd.)

If struct variables are *not initialized*, their member values are indeterminate (i.e., likely to be random values).

However, if an initializer is used, but there are fewer elements in the *brace-enclosed list* than members (or submembers), the remaining members will be *initialized implicitly*.

Implicitly initialized members end up with the same values as objects that have **static storage duration**.

Struct Initialization: Copying

A struct can also be initialized as a *copy* of another struct:

```
struct address fred_addr = bob_addr;
```

Note: this does what is called a **shallow copy** only!

A shallow copy means that the “top-level” members get copied; if any members are *pointers*—to separate objects—those objects do *not* get copied (recursively).

This means that when a struct contains pointers, copying it via such an assignment will result in two structs containing copies of the *same* pointer/address—i.e, sharing (sub)object(s).

So be very careful when copying structs via assignments!

Struct Initialization: Designators

With C99 and on, partial or out-of-order initialization can be done using **designators** of the form **.membername=**:

```
struct address bob_addr = {.zip = 62901, .city = "Carbondale"}
```

When using *designators*, unspecified members are automatically initialized with the same values as objects that have **static storage duration** (zero/NULL).

Initializers can mix designators with plain brace lists of members:

- each brace-enclosed initializer list has an associated member
- *w/o designator*, members of the *current* object are initialized, in struct declaration order
- *w/designator*, members of the *designated* member are initialized, in struct declaration order
- initialization then continues forward in struct declaration order, beginning with the next member

Struct Initialization (contd.)

With the addition of *designators*, there can be many alternative *brace notations* for initializing a struct.

Consider the nested structs used with POSIX timers:

```
struct itimerspec {
    struct timespec it_interval; /* Timer interval */
    struct timespec it_value;   /* Initial expiration */
};

struct timespec {
    time_t tv_sec;             /* Seconds */
    long tv_nsec;             /* Nanoseconds */
};
```

Struct Initialization (contd.)

An `itimerspec` struct could be initialized as:

```
struct itimerspec tspec = {...};
```

where many variations are possible for the brace list, e.g.:

- `{5,0,10,0};`
- `{{5,0},{10,0}};`
- `{.it_interval={5,0},.it_value={10,0}};`
- `{.it_value={10,0},.it_interval={5,0}};`
- `{.it_interval={5},.it_value={10}};`
- `{.it_interval=5,.it_value=10};`
- `{.it_interval={.tv_sec=5,.tv_nsec=0},.it_value={.tv_sec=10,.tv_nsec=0}};`
- `{.it_value={.tv_nsec=0,.tv_sec=10},.it_interval={.tv_nsec=0,.tv_sec=5}};`
- `{.it_interval={.tv_sec=5},.it_value={.tv_sec=10}};`
- `{.it_value.tv_sec=10,.it_value.tv_nsec=0,.it_interval.tv_sec=0,.it_interval.tv_nsec=0};`

Structs vs. Arrays

Because C effectively represents arrays as pointers but does not do this for structures, there are some confusing differences in using the two different composite data types.

One difference is that you cannot use the *equality operator* (`==`) to compare structures, while you can use it with arrays.

Of course with arrays, equality means *same memory block*; there is no builtin operator to compare the contents of two arrays to see if they are equal.

If you want to be able to compare structure variables to see if their members have the same values, you must write your own equality function.

Structs vs. Arrays (contd.)

A second key issue is that using a struct in an *assignment statement* will cause the same copying as we discussed above in relation to function parameters:

```
...
struct address bob_addr, tom_addr;
...
tom_addr = bob_addr;
```

This causes a *copy* to be made of the `bob_addr` struct and stored in `tom_addr`.

Sometimes this is what you want: `tom_addr` is separate from `bob_addr` but has the same member values (initially).

Structs vs. Arrays (contd.)

When using a struct pointer, **dynamic memory** may be used to allocate the struct's storage for the struct pointer variable:

```
...
struct address *bob_addrp;
bob_addrp = malloc(sizeof(struct address));
bob_addrp->street = "101 Main St.";
```

We can create struct pointer **alias** as follows:

```
...
struct address *tom_addrp;
tom_addrp = bob_addrp; //Struct is not copied, only pointer
```

Now `bob_addrp` and `tom_addrp` point to the same single struct object (allocated from the heap).

typedef

C provides limited methods for users to *define new data types*.

typedef allows users to create new **named types** from other C types.

The syntax is basically:

```
typedef C_type_spec new_type_identifier;
```

It has two primary uses:

- to provide a type name that reflects intended usage
- to allow simpler specification of composite types

typedef is how POSIX types like `size_t` are defined:

```
typedef unsigned long int size_t;  
(defines type size_t to be equivalent to unsigned long int)
```

typedef (contd.)

Example using typedef to simplifying using array as parameter:

```
typedef int 2d_int_array[10][10];  
  
void 2dfun(2d_int_array 2d_arr)  
{...}
```

`2d_int_array` is defined as a type equivalent to a 10 x 10 int array.

Without the typedef, the function definition would have to be:

```
void 2dfun(int 2d_arr[10][10])  
{...}
```

typedef (contd.)

typedef's are frequently used to simplify struct usage:

```
typedef struct point{  
    float x;  
    float y;  
} line[2];
```

This defines `line` as a type that is equivalent to an array of two struct `point` objects.

We can now define `line` parameters:

```
int linelen(line l)  
{...}
```

instead of the less clear:

```
int linelen(struct point l[2])  
{...}
```

typedef (contd.)

Example program showing use of struct and typedef:

```
#include <stdlib.h>  
#include <stdio.h>  
#include <math.h>  
  
typedef struct point{  
    double x;  
    double y;  
} line[2];  
  
double linelen(line l)  
{  
    return sqrt(pow(l[0].x - l[1].x, 2.0) + pow(l[0].y - l[1].y, 2.0));  
}  
  
int main(int argc, char *argv[])  
{  
    line l1 = {{0.0, 0.0}, {1.0, 1.0}};  
    struct point p1 = {0.5, 1.5};  
    struct point p2 = {3.25, 4.0};  
    line l2 = {p1, p2};  
  
    printf("Line #1 length: %f\n", linelen(l1));  
    printf("Line #2 length: %f\n", linelen(l2));  
    exit(EXIT_SUCCESS);  
}
```

More on typedefs with Structs

Another common use of typedef's with structs is eliminating the need to include the keyword `struct` with struct types.

For example, we can define `struct address` with a typedef as follows:

```
typedef struct address {
    char street[50];
    ...
} address_t;
```

This defines type `address_t` as equivalent to type `struct address`.

This allows us to now do:

```
address_t user_addr;
instead of:
struct address user_addr;
```

More on typedefs (contd.)

In fact, we can now use either "`address_t`" or "`struct address`".

By using an **unnamed struct** with a typedef, you can end up with a single type, e.g., `address_t`:

```
typedef struct {
    char street[50];
    ...
} address_t;
```

Using typedef's with structs is common, but it is by no means universally accepted as good style.

Some people believe you should always include the "struct" part of a struct type declaration, to make the struct aspect clear.

Unions

A **union** is a C data type mechanism that allows a *single memory address* to be accessed as different types, using different names.

For example, a type that can store a signed or unsigned int:

```
union soruint {
    signed int sval;
    unsigned int uval;
} var;
```

We can now access `soruint_var` as a signed int by doing:

```
var.sval
```

or as an unsigned int by doing:

```
var.uval
```

Note type of this union is: `union soruint`

Unions (contd.)

Unions can be *initialized* using *brace notation* similar to that for structs (and arrays).

Designators can also be used with unions (C99 and on).

However, there are some key differences between initializers for unions vs. for structs:

- a union's brace initializer expression must contain just a *single* expression
- without a designator, only the *first element* of the union can be initialized

Thus, brace list initializers are not terribly useful with unions (can simply do assignment to desired member).

Unions vs. Structs

Unions appear similar to structs:

- same syntax for defining a union as for a struct
- members/fields accessed with *dot and arrow operators*
- define new type: `union union_name`

Unions and structs are significantly different, however:

- *all struct members* will be valid for any struct object vs. *only one union member* will be valid for any union object at each point in time (same memory for all)
- the *size* of a struct object is the *sum* of the sizes of all its members vs. size of union object is the size of its largest member

Unions and Structs

Years ago, unions were often used to save address space in processes, by overloading the same memory for different uses in different parts of a program.

These days, the most common use for unions is in conjunction with structs: union(s) within a struct, or structs within a union.

Using the two together can provide *data type flexibility* that C otherwise lacks due to its lack of **type inheritance** (i.e., subclasses and superclasses).

For example, suppose a student record type (struct) must be able to accomodate one of any of several different ID types.

This could be done using multiple ID members of different types, but would be wasteful since only one will be in use at any time.

Unions and Structs (contd.)

Using a union nested within the struct avoids this waste:

```
struct student_record {
    struct name fullname;
    short id_type;
    union {
        drivlic_t dlic;
        socsec_t ss;
        visa_t visa; }
    ...
};
```

We can use this struct as so:

```
struct student_record newrecord;
...
newrecord.id_type = 1;
newrecord.dlic = "C61...";
```

Note the ability to use the names of the union's members just as if they were struct members.

Unions and Structs (contd.)

An example from the C standard, using structs inside a union:

```
union {
    struct {
        int alltypes; } n;
    struct {
        int type;
        int intnode; } ni;
    struct {
        int type;
        double doublenode; } nf;
} u;

u.nf.type = 1;
u.nf.doublenode = 3.14;
...
if (u.n.alltypes == 1)
    if (sin(u.nf.doublenode) == 0.0)
        ...
```

Note ability to access *int "common initial sequence"* in every struct, using any of union's member names.

Structs, Unions, and Arrays

Structs often contain members that are *structs*, *unions*, and/or *arrays*.

Since these types can all be initialized using *brace notation*, nested brace-enclosed initializer lists are quite common.

Consider this struct with embedded arrays, struct, and union:

```
struct test {
    int a;
    int b[2];
    union {
        int c;
        char d;
    };
    struct {
        int e;
        char f[5];
    } g;
    char h[3];
};
```

Structs, Unions, and Arrays (contd.)

Here are the results of various initializations:

- `struct test var = {'A'};`
a:65 b[0]:0 b[1]:0 c:0 d: g.e:0 g.f: h:
- `struct test var = {'A','B','C','D','E','F','G'};`
a:65 b[0]:66 b[1]:67 c:68 d:D g.e:69 g.f:FG h:
- `struct test var = {'A',{'B'},'C','D','E','F','G'};`
a:65 b[0]:66 b[1]:0 c:67 d:C g.e:68 g.f:EF h:
- `struct test var = {'A','B',.d='C','D','E','F','G'};`
a:65 b[0]:66 b[1]:0 c:67 d:C g.e:68 g.f:EF h:
- `struct test var = {'A',{'B'},'C','D',{'E','F'},'G'};`
a:65 b[0]:66 b[1]:0 c:67 d:C g.e:68 g.f:EF h:G
- `struct test var = {'A',{'[1]='B'},'C','D',{'E','F'},'G'};`
a:65 b[0]:0 b[1]:66 c:67 d:C g.e:68 g.f:EF h:G
- `struct test var = {'A',{'B'},{'C'},{'D','E','F'},{'G','H','I'}};`
a:65 b[0]:66 b[1]:0 c:67 d:C g.e:68 g.f:EF h:GHI
- `struct test var = {'A',{'B'},{'C'},{'D'},{'E','F'}},{'G','H','I'}};`
a:65 b[0]:66 b[1]:0 c:67 d:C g.e:68 g.f:EF h:GHI

Note: 'A' has decimal value 65, etc.

Details from the Standards

Key points from the C99/C11 standards:

- “a *structure* is a type consisting of a *sequence of members*, whose storage is allocated in an *ordered sequence*”
- “a *union* is a type consisting of a sequence of members whose storage *overlap*...size of a union is sufficient to contain the largest of its members”
- “Each non-bit-field member of a structure or union object is *aligned* in an *implementation-defined manner* appropriate to its type.”
- “Within a structure object, the non-bit-field members and [bit-field objects] have *addresses that increase in the order in which they are declared*.”
- “There may be *unnamed padding* within a structure object, but *not at its beginning*, [and]...at the *end* of a structure or union.”
- “A *pointer to a structure object*...points to its initial member (or if that member is a bit-field, then to [its] unit...)”
- “A *pointer to a union object*...points to each of its members (or if a member is a bit-field, then to [its] unit...)”
- “...a structure *shall not contain an instance of itself*, but may contain a *pointer to an instance of itself*...”

C Basics 13: Higher-Order Functions

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Multidimensional Arrays

C Basics 13: Higher-Order Functions

11. Dynamic and Static Memory
12. Structs (Structures)
13. **Higher-Order Functions**
 - **higher-order functions**
 - **function pointer parameters**
 - **library examples**
 - **statement expressions**
14. Variadic Functions
15. Multi-File Programs
16. Miscellaneous

Higher-Order Functions

Higher-order functions is the name given to functions that *take functions as arguments*.

Higher-order functions are common in mathematics:

e.g., $sum(m, n, f) \equiv \sum_{i=m}^n f(i)$

They are a key type of **abstraction mechanism** for algorithms.

We don't have to write: $sum_i(m, n)$, $sum_{i^2}(m, n)$, $sum_{i^3}(m, n)$, etc., because $sum(m, n, f)$ covers them all, by abstracting out what is common to all.

Higher-order functions are heavily used in “functional languages” like Lisp and Scheme.

Function parameters that are themselves functions are termed **procedural parameters** (though this term is not used in C).

Function Pointer Parameters

C allows functions to have *parameters that are functions*.

Technically, what gets passed to the function is a **function pointer**: a pointer to the function definition.

Thus, C *procedural parameters* are often termed **function pointer parameters**.

When the function pointer parameter name is used in a function call, the function pointer is *implicitly dereferenced* to invoke the passed function.

Function pointer parameters provide C with a capability that Java lacked until recently.

Note that *function arguments* are also often referred to as **callbacks**.

Higher-Order Functions in C

Higher-order functions can be defined in C by using **function pointer parameters**.

Their power is more limited than in a language like Lisp, however:

- C functions are not **first-class objects**
- C does not support **closures**
- C's type checking makes it difficult to use higher-order functions as **generic functions** (functions that can take different argument types—i.e., a form of **polymorphism**)
- the parameters of function pointer parameters will often have to be of type `void*`, requiring explicit *dereferencing* and explicit computation of *array offsets*
- function parameter arguments must be specified with constants; one cannot evaluate a variable to retrieve the function name and then get its address.

Defining Function Parameters

A *function pointer parameter* is specified with notation like:
`rettype (*funcname)(funcparam1, funcparam2,...)`

- `rettype` is the function parameter's *return type*
- `funcname` is the functiona parameter's name, used to call the argument function
- `funcparam1` is the function parameter's first parameter spec

Example: `int (*strtoint)(char *str)`

Specifies a function pointer parameter that has:

- `int` return type for function
- single `char*` parameter for function
- **formal parameter** name `strtoint`

Example Higher-Order Function: summation

A classic higher-order function most people are familiar with is *summation*, which is often written as: $sum(m, n, f) \equiv \sum_{i=m}^n f(i)$

We can implement summation in C as:

```
int summation(int start, int end, int (*func)(int))
{
    int sum = 0;
    for (int i=start; i<=end; i++)
        sum += func(i); //Note parameter func used in function call

    return sum;
}
```

Example: summation (contd.)

`summation()` would be used as follows:

```
int intident(int x)
{
    return x;
}

int intsquare(int x)
{
    return x * x;
}

int main()
{
    int sumfirst10 = sum(1,10,intident);

    int sumfirst5sq = sum(1,5,intsquare);

    ...
}
```

Standard Higher-Order Functions

Several library and system call functions are higher-order functions, including:

- `int atexit(void (*function)(void))`
- `int on_exit(void (*function)(int, void *), void *arg)`
- `void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *))`
- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`
- `void (*signal(int signum, void (*handler)(int)))(int)`

`signal()`'s definition is particularly confusing because it *returns a function pointer*, so it is typically defined using typedef:

```
typedef void (*sighandler_t)(int)
sighandler_t signal(int signum, sighandler_t handler)
```

Example: pthread_create()

E.g., `pthread_create()` takes a `void*` argument, which will be passed to its `start_routine` function (that has a `void*` parameter).

`void*` parameters are often used with function parameters because they allow any type of data to be passed to the function argument.

This allows different `start_routine()`'s to have very different input requirements and yet still match a fixed prototype.

To pass data to `start_routine()`, an appropriate struct is defined and instantiated, and the instantiation's address/pointer passed to `pthread_create()` (*casting as needed*).

Example: qsort()

`qsort()` is a *higher-order function* that is also a **generic function**:

```
void qsort(void *base, size_t nmemb, size_t size,
int(*compar)(const void *, const void *))
```

Generic sorting functions are another classic application for higher-order functions.

`qsort` allows an array of *any type* of data to be sorted, as long as a **binary comparison function** (basically \leq) exists for the type.

The important insight is that the comparison function will be passed *void pointers* to the two objects to be compared, so the argument pointers must be:

1. *cast* to the correct pointer type
2. *dereferenced* to obtain the objects to compare

Example: qsort() (contd.)

The man page for `qsort()` shows how to define a compatible comparison function for strings, using `strcmp()` to compare the two strings:

```
cmpstringp(const void *p1, const void *p2)
{
    /* The actual arguments to this function are "pointers to
    pointers to char", but strcmp(3) arguments are "pointers
    to char", hence the following cast plus dereference */

    return strcmp(*(char * const *) p1, *(char * const *) p2);
}
```

Example: qsort (contd.)

A comparison function for `int`'s could be written as:

```
cmpintp(const void *p1, const void *p2)
{
    int x = *(const int *)p1;
    int y = *(const int *)p2;

    if (x == y)
        return 0;
    else if (x < y)
        return -1;
    else
        return 1;
}
```

Example: qsort() (contd.)

It would be used as follows:

```
int main()
{
    int iarr[] = {5,2,4,5,8,1};

    ...

    qsort(iarr,6,sizeof(int),cmpintp);

    ...
}
```

Statement Expressions

As an extension, GCC allows **compound statement** to be used as an **expression** if *enclosed in parentheses*.

This allows you to use loops, switches, local variables, etc. within an expression—i.e., to implement a **functional programming** style.

(A *compound statement* is a sequence of statements surrounded by *braces*, while an *expression* has a value (rvalue).)

The *final statement* in the compound statement expression must be an *expression* (followed by a semicolon).

The value of the final expression becomes the value of the entire compound statement expression.

Statement Expressions (contd.)

Example:

```
static char *strings[NUMSTRINGS];
...
if ( ({int retn=0;
      for(int i=0;i<NUMSTRINGS;i++)
          if(strings[i]!=NULL){retn=1; break;}
      retn;}) )
    printf("strings contains some strings\n");
else
    printf("strings is empty!\n");
```

This is basically the code to check if an array of strings contains any strings or not, written right inside of the `if` clause.

(The expression is written on multiple lines only to make it readable for the slides.)

Statement Expressions (contd.)

This avoid having to do:

```
static char *strings[NUMSTRINGS];
...
int notempty=0;
for(int i=0;i<NUMSTRINGS;i++)
    if(strings[i]!=NULL) {
        notempty=1;
        break; }
if (notempty)
    printf("strings contains some strings\n");
else
    printf("strings is empty!\n");
```

Statement Expressions (contd.)

Or having to define a function:

```
int contains(char *strarr[], int arrsize)
{
    for(int i=0;i<arrsize;i++)
        if(strarr[i]!=NULL)
            return 1;
    return 0;
}
```

And then call it just once:

```
if (contains(strings,NUMSTRINGS))
    printf("strings contains some strings\n");
else
    printf("strings is empty!\n");
```

C Basics 14: Variadic Functions

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Multidimensional Arrays

C Basics 14: Variadic Functions

11. Dynamic and Static Memory
12. Structs (Structures)
13. Higher-Order Functions
14. **Variadic Functions**
 - **variadic functions**
 - **stdarg.h and its macros**
 - **examples**
 - **variadic preprocessor macros**
15. Multi-File Programs
16. Miscellaneous

Variadic Functions

Variadic functions are functions that can accept a *variable number of arguments*.

They are also simply called functions with *variable argument lists*.

Another way of saying this is that they are functions of *indefinite arity*.

A key example in C is `printf()`, which takes a single format string, but then an *arbitrary* number of value arguments:

```
int printf(const char *format, ...);
```

Lisp has a wide range of variadic functions, such as the arithmetic operators: `(+ x y z w)`.

C allows users to define variadic functions and macros.

stdarg.h

To implement variadic functions in C, the header file `stdarg.h` must be included.

`stdarg.h` declares the type `va_list` and four macros:

- `void va_start(va_list ap, last)`
- `type va_arg(va_list ap, type)`
- `void va_end(va_list ap)`
- `void va_copy(va_list dest, va_list src)` (C99 addition)

See “`man stdarg`” for further info.

A key limitation is that C variadic functions with *no fixed arguments* are *not possible*.

stdarg.h (contd.)

void va_start(va_list ap, last):

- initializes ap for subsequent use by va_arg() and va_end()
- must be *called first*
- last is the name of the last argument before the variable argument list
- last argument the calling function knows the type of

type va_arg(va_list ap, type):

- expands to an expression that has the type and value of the next argument in the call
- when first called after va_start(), it returns the argument immediately after last
- each subsequent call returns the next argument

stdarg.h (contd.)

void va_end(va_list ap):

- each va_start() invocation must be matched by a corresponding va_end() invocation in the same function
- multiple argument list traversals are allowed, each must be bracketed by va_start() and va_end()

void va_copy(va_list dest, va_list src):

- copies the (previously initialized) variable argument list src to dest
- each va_copy() invocation must be matched by a corresponding va_end() invocation in the same function

Example Variadic Function

Example C variadic function from Wikipedia:

```
double average(int count, ...)
{
    va_list ap;
    int j;
    double sum = 0;

    va_start(ap, count);
    for (j = 0; j < count; j++) {
        sum += va_arg(ap, int);
    }
    va_end(ap);

    return sum / count;
}

int main(int argc, char const *argv[])
{
    printf("%f\n", average(3, 1, 2, 3) );
    return EXIT_SUCCESS;
}
```

Example Variadic Function

Example C variadic function from stdarg man page:

```
void simple_printf(char *fmt, ...)
{
    va_list ap;
    int d;
    char c, *s;

    va_start(ap, fmt);
    while (*fmt)
        switch (*fmt++) {
            case 's': /* string */
                s = va_arg(ap, char *);
                printf("string %s\n", s);
                break;
            case 'd': /* int */
                d = va_arg(ap, int);
                printf("int %d\n", d);
                break;
            case 'c': /* char */
                /* cast needed since va_arg takes only fully promoted types */
                c = (char) va_arg(ap, int);
                printf("char %c\n", c);
                break;
        }
    va_end(ap);
}
```

Variadic Preprocessor Macros

C99 added the ability to define **variadic preprocessor macros**.

Example (from GCC manual):

```
#define debug(format, ...) fprintf(stderr, format, __VA_ARGS__)
```

GCC allows a name to be given to the variable arguments:

```
#define debug(format, args...) fprintf(stderr, format, args)
```

The `debug` macro could be called like:

```
debug("The value of x is: %d\n",x);
```

Variadic Preprocessor Macros (contd.)

The C standard does not allow the variable argument to be left out entirely (you are allowed to pass an empty argument).

E.g., the following invocation is invalid in ISO C, because there is no comma after the string:

```
debug("A message")
```

It will lead GCC will complain since the expansion of the macro will contain an extra comma after the format string:

```
fprintf(stderr, "A message", )
```

GCC allows the `##` operator to be used to remove the comma before it if the variable arguments are omitted or empty:

```
#define debug(format, ...) fprintf(stderr, format, ## __VA_ARGS__)
```

C Basics 15: Multi-File Programs

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Static and Dynamic Memory

C Basics 15: Multi-File Programs

11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. **Multi-File Programs**
 - **translation units**
 - **declarations and definitions**
 - **linkage and scope**
 - **linkage and storage-class specifiers**
16. Miscellaneous

Translation Units

A large C program can be broken up into a set of **source files**, **header files**, and **libraries**.

Recall that “translation” of a C program into machine code involves both **preprocessing** followed by true compilation.

Preprocessing Translation Unit:

A *source file* along with *header files*, etc. to be **#include**'d.

Translation Unit:

A *source file* that has undergone **preprocessing**.

A *complete* C program can include *multiple translation units*.

Translation units can be translated (compiled) *separately*, and then later **linked** to produce a single **executable**.

Translation Units (contd.)

Code in separate translation units interacts via functions and objects that have **external linkage**:

- calling functions defined in other units
- accessing **global variables** defined in other units
- functions/globals not declared **static**

Declaration vs Definitions

The terms **declaration** and **definition** have distinct meanings in C, which can be critical to understand, especially for multi-file programs.

A **declaration** of an *identifier* specifies the “interpretation and attributes” of the identifier: data type, storage duration, linkage, and same for parameters of a function.

A **definition** of an identifier is a declaration that:

- for an object, causes storage to be reserved/allocated
- for a function, includes the function body (code)
- declares an enumeration constant or typedef name

Declaration vs Definitions (contd.)

Multiple declarations are allowed within and across translation units of a C program (for the same *identifier* in the same *scope*).

However, only a *single definition* is allowed within and across translation units of a C program (for the same *identifier* in the same *scope*).

With multi-file programs in particular, care must be paid to ensure that only a single definition is seen, while also ensuring that a declaration is seen prior to each identifier being referenced.

(When there are multiple declarations, they must all be “consistent.”)

Declaration vs Definitions (contd.)

Declarations of *global variables (file scope)* can be confusing.

Whether the declaration includes an **initializer** (initial value) or not is important:

- with an initializer, the declaration is an **external definition**
- without an initializer (and *not* declared **extern**), the declaration is a **tentative definition**

Tentative definitions are relevant when a translation unit does not include an external definition:

If a translation unit contains only tentative definition(s) for an identifier, it will be equivalent to an external definition with initializer of “0” (zero).

Declaration vs Definitions (contd.)

Examples:

```
//Global (file scope) variables:
int i = 1;           //external definition (initializer)
int j;              //tentative definition (no initializer)
extern int k;       //declaration (extern keyword, so defined externally)
static int l;       //tentative definition (no initializer, internal linkage)
static int m = 0;   //external definition (initializer, internal linkage)

double add(double, double); //declaration (prototype)

double add(double x, double y) //definition
{
    double z1;           //definition (no initializer, but not file scope)
    double z2 = 2.5;     //definition (with initializer)
    extern double z3;    //declaration (references external linkage var z3)

    ...
}
```

Scope

Scope refers to the portion of a program where a particular **binding** of a value to an **identifier** is **visible** (accessible).

Scope was covered in lecture #10: **Static and Dynamic Memory**.

Linkage

In addition to scope, C includes the concept of **linkage**.

The C99 standard says:

“An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called **linkage**. There are *three kinds of linkage*: **external**, **internal**, and **none**.”

Together, an identifier’s scope and linkage determine whether the identifier is visible and which binding is referenced, at any particular point in a C program.

Identifier linkage must be taken into account when mapping standard programming language concepts of “scope” to C.

Linkage (contd.)

The rules for linkage are as follows for identifiers:

- **external linkage** means that each declaration of the identifier denotes the same binding (object/function) across the entire set of translation units and libraries
- **internal linkage** means that each declaration of the identifier denotes the same binding (object/function) within each single translation unit
- **no linkage** means each declaration of an identifier denotes a unique binding

Identifiers have a *default linkage* depending on their object type and their placement in code.

Linkage may be able to be modified using the **storage-class specifiers**: **extern** and **static** in a declaration.

Linkage (contd.)

Default linkage (no use of `extern` or `static`):

- **no linkage**:
 - block scope identifier for an object
 - identifier for anything other than an object or a function
 - identifier for a function parameter
- **external linkage**:
 - file scope identifier for an object (i.e., a global variable)
 - identifier for a function
(unless previous declaration included `static`)

Linkage (contd.)

Including **static** in a declaration of an identifier for an *object* or *function* results in the identifier having **internal linkage**.

Including **extern** in a declaration of an identifier means:

- if declaration is within the scope of prior declaration:
if the prior declaration specifies internal or external linkage, that linkage remains for the new declaration
- if not in scope of prior declaration or no linkage specified in prior declaration:
then the identifier has **external linkage**

(Objects with *external linkage* have **static storage duration**.)

Global Scope

By default, *C functions* and *global variables* have **file scope** and **external linkage**.

This results in what is more commonly termed **global scope**:

- functions can be called from any translation unit (file)
- global variables are shared across all translation units (files)

Note however, that C compilers must still see declarations of global scope identifiers before they encounter any references.

Providing declarations can differ slightly for global variables versus functions.

In both cases, only a *single definition* must be provided!

Global Scope (contd.)

With a global variable, there must be a *single definition* of the variable within all the translation units (i.e., be in just one unit):

```
int count = 0;
```

The **extern** keyword is used to provide *declarations that are not definitions* in other translation units:

```
extern int count;    Or    int extern count;
```

The **extern** declaration can be:

- written explicitly into each source code file that references the global, or
- placed in a *header file* `#include'd` by all the source code files of the program

Note: the definition *cannot* be in the header file!

Global Scope (contd.)

Handling of functions differs slight due to the existence of **function prototypes**, which are merely declarations.

The *definition* (i.e., code) for each global scope function will be placed into a single source file.

Prototypes for these functions must be made available to all source files that call the functions.

This is most commonly done by placing the prototypes into one or more *header files* that are `#include'd` by source files that call the functions.

Prototypes can include the **extern** or not, it makes no difference.

Internal Linkage

Adding the `static` modifier to the definition of a function or global variable changes its **linkage** to **internal**:

```
static int check_count(int number) {...}
static double counter = 0.0;
```

This effectively reduces the *scope* of the function/variable to *the remainder of the file containing the definition*.

Internal linkage is thus a type of “**information hiding**” mechanism.

It can prevent inherently global scope functions and “globals” from being improperly accessed from other program modules.

It also helps limit *name conflicts*, since C has a *single namespace* for most identifiers.

Internal Linkage (contd.)

In a single C source file, we may have both `static` and non-`static` function definitions.

Those functions defined with `static` are intended to be use *locally*, i.e., only with other functions defined within the file.

Note that if a function prototype is declared `static`, and that prototype is seen before the definition, the function will have *internal linkage* whether or not `static` occurs in the definition.

C Basics 16: Miscellaneous

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Multidimensional Arrays

C Basics 16: Miscellaneous

11. Dynamic and Static Memory
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Multi-File Programs
16. **Miscellaneous**
 - **array allocation and performance**
 - **restricted pointers**
 - **unions and type punning**
 - **strict aliasing**
 - **alignment**
 - **padding and packing in structs**
 - **struct bit-fields**

Array Allocation and Performance

There are several ways to allocate space for arrays, and these approaches can affect performance.

The four basic approaches are:

1. standard main/function variable with size specified via a *constant*: *automatic storage duration*, allocated on *stack*
2. "global" or static variable (size must be specified via a constant): *static storage duration*, allocated in *data/BSS segment*
3. standard main/function variable with size specified via a *variable*: *variable-length array*, allocated on *stack*
4. pointer variable allocated memory via `malloc()`, etc.: *allocated storage duration*, allocated on *heap*

(Also, an array definition may *initialize* the array or not for 1–3.)

Array Allocation and Performance (contd.)

Performance with arrays can be affected by the runtime to:

- allocate array storage
- access array elements
- deallocate array storage

Allocation performance (likely):

- (2) fastest since space is allocated by the compiler
- (1) next fastest, since stack operations set by compiler
- (3) slower because size determined at runtime
- (4) slowest due to dynamic memory operations at runtime

Array Allocation and Performance (contd.)

Deallocation performance is similar:

- (2) fastest since space is not deallocated during run
- (1) and (3) comparable: return from stack frame
- (4) slowest due to dynamic memory operations at runtime

Element access performance is complicated, being affected by:

- runtime cost to compute element offset
- speed of retrieval of element from memory
- need to reload data from RAM

Array Allocation and Performance (contd.)

The storage/access scheme for array elements is discussed in C Basics lectures #5 (**Pointers and Arrays**) and #11 (**Multidimensional and String Arrays**).

Element offset computation speed depend on how array indices are specified:

- indices specified with *constants* (`arr[3][5]`), allow offset to be computed at compile time
- indices specified with *variables* (`arr[x][y]`), will require offsets be computed at runtime

Note that **jagged arrays** will make element access slower, since each *subarray* (represented as a *pointer*) will require its own successive offset computation.

Array Allocation and Performance (contd.)

How fast a particular array element can be retrieved from memory once its address has been computed, can depend significantly on where it is stored in a process' address space.

If the element is in the CPU **cache**, access will be fastest.

If it is in a **swapped out** memory page, access will be slow.

In general, arrays stored on the *stack* will be most likely to be in cache and so accessed most rapidly.

Speed of access to static and heap arrays will depend on the access pattern of each program.

Modern computers with sufficient RAM do little/no swapping.

Array Allocation and Performance (contd.)

Another issue that can affect array access time is whether fixed elements need to be *reloaded* each time they are referenced.

With arrays defined in the *local scope* (e.g., in a function), the compiler can be sure no other *thread* could have modified the array between operations.

This can allow it to avoid reloading (from RAM) for each access.

Global arrays, however, could be modified by other threads, so elements have to be reloaded for each access.

```
for (int i=0; i<n; i++)
    arr2[i] += arr1[0]; //kept in register or must be reloaded?
```

Array Allocation and Performance (contd.)

One last array allocation consideration is the *size of the array*.

The OS imposes limits on the size of certain **process segments**, and the CPU architecture imposes total **address space** limits.

With 64-bit CPUs, address space is rarely a consideration, but it can be with 32-bit CPUs (or smaller “embedded processors”).

Linux process limits can be seen with “*help ulimit*”:

- -d – the maximum size of a process’s data segment
- -s – the maximum stack size

On a 64-bit Linux, we see:

- `ulimit -d` – unlimited
- `ulimit -s` – $8192 \times 1024 = 8 \text{ MB}$

Array Allocation and Performance (contd.)

This means that large arrays generally need to be allocated in static storage or on the heap (allocated storage), not in stack/automatic storage (though the stack limit can be increased).

While GCC will compile a program with a large automatic array, the program will immediately **segfault** when it is run!

So if you are creating a large array, store it as a “global” variable, a **static** variable, or one dynamically allocated with `malloc()`.

Of course, the stack limit can also be increased using `ulimit`.

Restricted Pointers

A properly written C program is usually about as fast as possible.

However, the flexibility of C pointers can cause C programs to be less efficient than Fortran or assembler programs when dealing with pointers and arrays.

The problem is that C allows pointers to be **aliases**:

- point to exact same memory block
- point to *overlapping* blocks of memory

This can lead to slower execution than may be possible:

- values may have to be repeatedly *reloaded* from RAM
- the compiler may be prevented from *reordering* operations
- the CPU may be prevented from *parallelizing* operations (via **vector instructions** or **instruction pipelining**)

Restricted Pointers (contd.)

Wikipedia has this C example:

```
void updatePtrs(size_t *ptrA, size_t *ptrB, size_t *val)
{
    *ptrA += *val;
    *ptrB += *val;
}
```

In C, the three pointer parameters could all refer to the *same location*, so `*val` must be loaded from RAM twice.

Faster code can be produced if the pointers are known to be unique addresses.

Fortran code with arrays is faster than C code, because aliases must be explicitly created with **equivalence**, so the compiler knows when arrays/pointers might be aliases.

Restricted Pointers (contd.)

To allow C compilers to produce comparable code, C99 added the **restrict** *type qualifier* (for *pointer declarations*):

“An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association ... requires that all accesses to that object use, directly or indirectly, the value of that particular pointer. The intended use of the restrict qualifier ... is to promote optimization...”

The compiler can be informed of that the parameters in the above example will *not* be aliases, by using **restrict**:

```
void updatePtrs(size_t *restrict ptrA, size_t *restrict ptrB, size_t *restrict val)
```

These are called **restricted pointers**.

They can be used to qualify *blocks* of memory, including standard and dynamic arrays

Restricted Pointers (contd.)

There are four types of restricted pointers:

- **file scope** – should point into a single array object for the duration of the program, array may not be referenced both through the restricted pointer and through its declared name (if exists) or another restricted pointer; useful in providing access to dynamically allocated global arrays
- **function parameters** – useful for pointer parameters of a function to indicate there is no aliasing of parameters (parameter provides exclusive access to its memory/array)
- **block scope** – makes an aliasing assertion that is limited to its block, e.g., loop body
- **structure member** – restricted pointer member of a structure makes an aliasing assertion, where scope is the scope of the identifier used to access the structure

Restricted Pointers (contd.)

The **restrict** keyword can also be used inside of an array parameter's []'s, to make the pointer that an array argument is converted to, be restricted.

The following declarations are all compatible: (C11 standard)

```
void f(double (* restrict a)[5]);
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
void f(double a[restrict static 3][5]);
```

Unions

A **union** (type) allows the same memory to be accessed as *different types*.

A union declaration looks similar to a *struct* declaration:

```
union id_data {
    int ival;
    double dval;
};
```

Variable definition and **member** access also looks similar:

```
union id_data uvar;
uvar.ival = 10;
```

Unlike a struct, however, only one member at a time should be accessed, since all the members all share the same memory.

Type Punning

Type punning refers to programming techniques that subvert a language's type checking to allow objects to be treated as more than one data type.

In C, **unions** and **pointer casts** can be used for type punning.

There are some legitimate uses of type punning in C, due to its lack of **subtype polymorphism**.

E.g., casting a specific socket address type to the generic socket address type in the `bind()` syscall:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

However, type punning can also result from programming errors!

Strict Aliasing

C compilers may try to detect type punning that occurs via **pointer aliases**.

Strict aliasing (or the **strict aliasing rule**), refers to a C compiler making the following assumption:

"An object of one type is assumed never to reside at the same address as an object of a different type, unless the types are almost the same. For example, an unsigned int can alias an int, but not a void or a double. A character type may alias any other type."* (GCC manual)

GCC has the options `-fstrict-aliasing` and `-fno-strict-aliasing` (as well as warning options) for controlling whether the compiler can make optimizations by assuming pointers are not aliases based on types.

Strict Aliasing (contd.)

Strict aliasing arises from this element of the C standard:

"An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

- a type compatible with the effective type of the object,*
- a qualified version of a type compatible with the effective type of the object,*
- a type that is the signed or unsigned type corresponding to the effective type of the object,*
- a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,*
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or*
- a character type."* (C99 standard)

Strict Aliasing (contd.)

The C standard allows type punning only with *unions* or with *char pointer aliases*.

The following code is fine since memory is accessed via a union:

```
union aunion {
    int i;
    double d;
};
...
union aunion u;
u.d = 3.0;
int j = u.i;
```

Alignment

A memory address is **n-byte aligned** when the address is a *multiple of n bytes* (where n is a power of 2).

(**Virtual and physical memory addresses** on modern CPUs/OSs are in terms of *bytes*, i.e., bytes are the smallest addressible memory units.)

An *n-byte data object* is thus said to be **aligned** if its memory block starts at an address that is *n-byte aligned*.

Otherwise, the object is said to be **misaligned**.

E.g., one-byte objects (e.g., `char`'s) are always aligned, two-byte objects are aligned only if their first byte is in an even-numbered address (0, 2, 4,...), four-byte objects are aligned if their first byte is in an address divisible by 4 (0, 4, 8,...), etc.

Alignment (contd.)

A **word** is the basic number of bits/bytes of memory that a CPU can access and operate on as a unit.

Modern CPUs generally have word sizes of 8, 16, 32, or 64 bits (1, 2, 3, or 4 *bytes*), with 32 and 64 bits now most common (for all but embedded systems).

CPUs access memory a word at a time, but for many CPUs, the words must be *aligned*.

E.g., 32-bit (four-byte) words must start at (byte) addresses that are multiples of 4.

If data objects are *aligned* (and no larger than word size), the CPU can access/operate on the units in a *single CPU cycle*.

Alignment (contd.)

When data objects are *misaligned*, however, the CPU may either not be able to access them or may require multiple cycles to access them (e.g., fetching two words each of which contain part of the object),

The ability of CPU architectures to access/operate on misaligned data, and the speed penalty for operating on such data when possible, varies across CPU architectures.

Because of this, C compilers such as GCC by default generate code that results in all data objects (variables, struct members, etc.) being properly aligned.

(Still, when writing C code that must be portable across 32 and 64-bit x86 as well as ARM, etc., alignment can be an issue.)

Padding

How does a C compiler ensure that data objects are aligned?

In one of two ways:

- adding **padding** bytes between objects
- *reordering* objects (where allowed by C standard)

The easiest way to see padding in action is with *structs* (C standard does not allow struct members to be reordered).

First, be aware that C compilers generally:

- struct instance will have the alignment of its *widest scalar member* (to ensure that all the members are self-aligned)
- there is no *leading padding*
- *trailing padding* may be added to ensure following object is aligned as struct (**stride address**)

Padding and Structs

Sizes of some C types on 64-bit Linux on x86_64:

- char: 1 byte
- short: 2 bytes
- int: 4 bytes
- void*: 8 bytes

An n -byte aligned address will have a minimum of $\log_2(n)$ least-significant zeros when expressed in binary.

Addresses are generally given in hex, so only the *least-significant hex digit* is relevant to alignment.

E.g., for 4-byte alignment it would have to be 0, 4, 8, or C (12).

Padding and Structs

Consider the following code:

```
struct test1{
    int x1;
    short x2;
    int x3;
};

int v0 = 0;
struct test1 v1 = {1,2,3};
int v2 = 0;

printf("v0:%p v1:%p x1:%p x2:%p x3:%p size:%ld v2:%p\n",
      &v0,&v1,&v1.x1,&v1.x2,&v1.x3,sizeof(v1),&v2);
```

Output:

```
v0:0x7ffcdf05008c //v0 4-byte aligned, immediately after struct x3 member
v1:0x7ffcdf050080 //struct start, 4-byte aligned
x1:0x7ffcdf050080 //x1 4-byte aligned, same as start of struct
x2:0x7ffcdf050084 //x2 automatically 4-byte aligned, but only 2-byte required
x3:0x7ffcdf050088 //x3 4-byte aligned, 2 bytes of padding before required
size:12 //struct 12 bytes instead of required 10, due to padding
v2:0x7ffcdf05007c //v2 4-byte aligned, immediately before struct
```

Struct Packing

The C standard does not allow a compiler to reorder the members of a struct (since the order may be required for some reason as in an OS device driver).

However, a program's developer is obviously free to redesign structs so as to minimize padding.

This is known as **struct(ure) packing**.

There are two equally valid approaches for minimizing padding in structs:

- order members from smallest to largest
- order members from largest to smallest

Struct Packing (contd.)

While struct packing is not critical in most applications on modern systems, programs that create *many thousands* of struct instances can sometimes significantly improve their memory performance.

On systems with limited RAM, such as many *embedded systems*, struct packing could have a significant impact on memory use (and so performance) or even could be the only way to allow a program to run.

Bit-Fields

Bit-fields (also *bitfields* or *bit fields*) are another approach for packing more struct members into a given number of bytes.

Consider a struct that contains a number of (binary) **flags**:

```
struct flags {
    int flag1;
    int flag2;
    ...
    int flag8;
};
```

This struct requires $8 \times 4 = 32$ bytes to store only *8 bits* (one byte) of information.

Thus, it wastes 31 bytes per struct instance.

Bit-Fields (contd.)

C bit-fields allow the *number of bits* that should be used for `_Bool`, `signed int`, or `unsigned int` struct members to be specified.

A bit-field is specified by adding a colon (:) and an integer constant after a **member_definition** (see **Structs** lecture):

```
struct flags {
    unsigned int flag1: 1;
    unsigned int flag2: 1;
    ...
    unsigned int flag8: 1;
};
```

This struct theoretically requires just one byte!

(Types have been changed from `int` to `unsigned int`, because a single bit cannot have a sign bit.)

Bit-Fields (contd.)

In reality, more space will generally be required due to alignment and related CPU instruction issues.

Bit-fields are implemented using machine instructions that operate on the bits in units such as *words*, so the minimum width of storage will be at least one word typically.

For example, the above bit-field `flags` struct takes 4 bytes on an `x86_64` system using `GCC`.

Of course the earlier `int` members version of `flags` takes 32 bytes, so there is still a substantial savings.

Bit-Fields (contd.)

Many elements of bit-fields are left to the implementation, so may vary between compilers and between CPU architectures.

In particular, the order bit-field elements are stored is implementation dependent, and can be affected by specifics of the fields, CPU **endianess**, etc.

This means that programmers must be very careful when using bit-fields, especially when code must be portable!

And of course it is up to the programmer to ensure values stored in bit-fields are appropriate for the number of bits and type.

E.g., consider a *4-bit* `int` bit-field member; here are how certain assigned values may end up being stored/interpreted:

$10 \Rightarrow -6$, $15 \Rightarrow -1$, $17 \Rightarrow 1$, $20 \Rightarrow 4$.

Bit-Fields (contd.)

Since bit-fields may not be on byte-boundaries, you are not allowed to get the *address* of a bit-field member of a struct.

Alignment affected by **unnamed, zero-width** bit-field:

- declaration omits identifier, specifies width of 0 (zero)
- indicates that no further bit-field is to be packed into the unit the previous bit-field was stored in
- next bit-field is to use a new storage unit, so be *aligned*

Example:

```
struct flags {
    unsigned int flag1:1; //aligned, start of bit-field object
    unsigned int flag2:1;
    int :0;
    unsigned int flag3:1; //aligned
    unsigned int flag4:1;
};
```

Bit-Fields (contd.)

Key points from the C99 standard:

- “An implementation may allocate any *addressable storage unit* large enough to hold a bit-field.”
- “The alignment of the addressable storage unit is unspecified.”
- “If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined.”
- “The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined.”
- “Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared.”
- “A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa.”