6.087 Lecture 1 – January 11, 2010

- Introduction to C
- Writing C Programs
- Our First C Program

What is C?

- Dennis Ritchie – AT&T Bell Laboratories – 1972
  - 16-bit DEC PDP-11 computer (right)
- Widely used today
  - extends to newer system architectures
  - efficiency/performance
  - low-level access

Features of C

C features:

- Few keywords
- Structures, unions – compound data types
- Pointers – memory, arrays
- External standard library – I/O, other facilities
- Compiles to native code
- Macro preprocessor

Versions of C

Evolved over the years:

- 1972 – C invented
- 1978 – *The C Programming Language* published; first specification of language
- 1989 – C89 standard (known as ANSI C or Standard C)
- 1990 – ANSI C adopted by ISO, known as C90
- 1999 – C99 standard
    - mostly backward-compatible
    - not completely implemented in many compilers
- 2007 – work on new C standard C1X announced

In this course: ANSI/ISO C (C89/C90)

What is C used for?

Systems programming:

- OSes, like Linux
- microcontrollers: automobiles and airplanes
- embedded processors: phones, portable electronics, etc.
- DSP processors: digital audio and TV systems
- ...

C vs. related languages

- More recent derivatives: C++, Objective C, C#
- Influenced: Java, Perl, Python (quite different)
- C lacks:
  - exceptions
  - range-checking
  - garbage collection
  - object-oriented programming
  - polymorphism
  - ...
- Low-level language $\Rightarrow$ faster code (usually)

Warning: low-level language!

Inherently unsafe:

- No range checking
- Limited type safety at compile time
- No type checking at runtime

Handle with care.

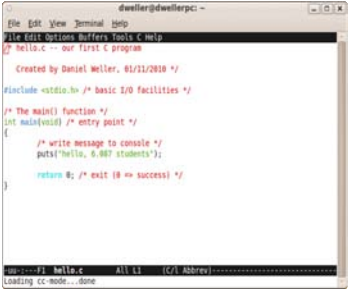
- Always run in a debugger like `gdb` (more later...)
- Never run as `root`
- Never test code on the Athena[1] servers

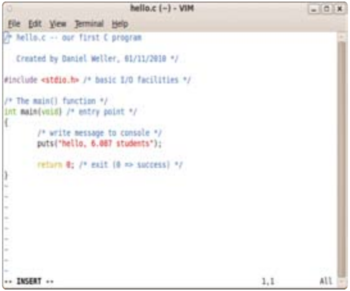
[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

- Introduction to C
- Writing C Programs
- Our First C Program

Editing C code

- `.c` extension
- Editable directly





- More later...

Compiling a program

- `gcc` (included with most Linux distributions): compiler
- `.o` extension
  - omitted for common programs like `gcc`

More about gcc

- Run `gcc`:

  athena%[1] gcc -Wall *infilename*.c -o *outfilename*.o

- `-Wall` enables most compiler warnings
- More complicated forms exist
  - multiple source files
  - auxiliary directories
  - optimization, linking
- Embed debugging info and disable optimization:

  athena% gcc -g -O0 -Wall *infilename*.c -o *outfilename*.o

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Debugging

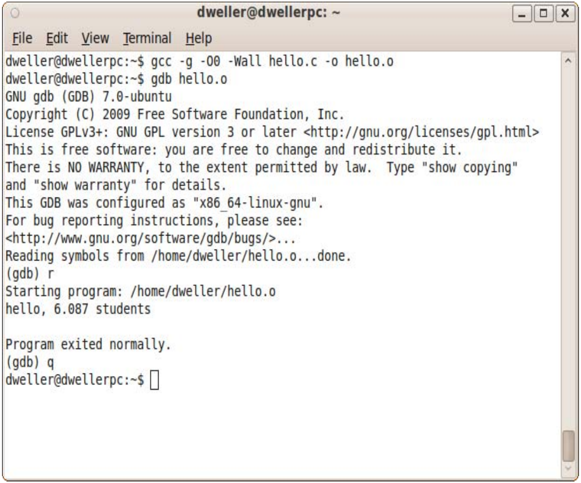


Figure: `gdb`: command-line debugger

Using gdb

Some useful commands:

- `break` *`linenumber`* – create breakpoint at specified line
- `break` *`file:linenumber`* – create breakpoint at line in file
- `run` – run program
- `c` – continue execution
- `next` – execute next line
- `step` – execute next line or step into function
- `quit` – quit `gdb`
- `print` *`expression`* – print current value of the specified expression
- `help` *`command`* – in-program help

Memory debugging

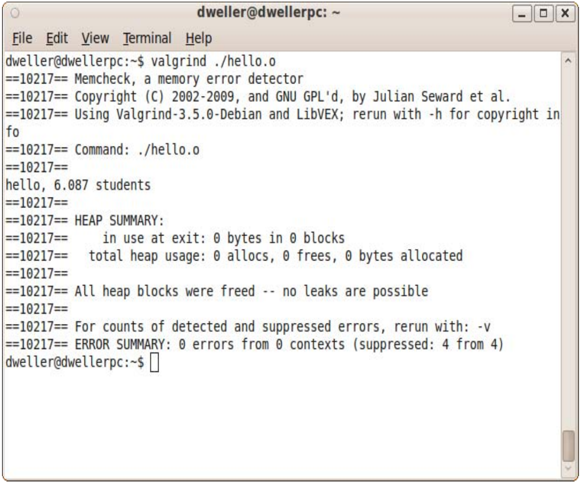
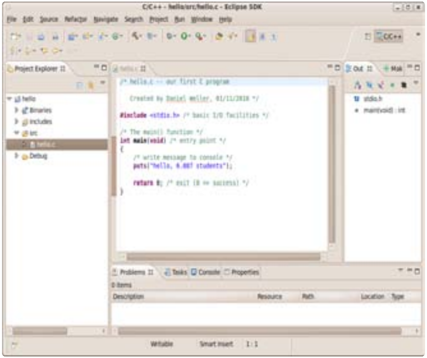


Figure: `valgrind`: diagnose memory-related problems

The IDE – all-in-one solution

- Popular IDEs: Eclipse (CDT), Microsoft Visual C++ (Express Edition), KDevelop, Xcode, . . .
- Integrated editor with compiler, debugger
- Very convenient for larger programs



Courtesy of The Eclipse Foundation. Used with permission.

Using Eclipse

- Need Eclipse CDT for C programs (see `http://www.eclipse.org/cdt/`)
- Use New > C Project
  - choose “Hello World ANSI C Project” for simple project
  - “Linux GCC toolchain” sets up `gcc` and `gdb` (must be installed separately)
- Recommended for final project

- Introduction to C
- Writing C Programs
- Our First C Program

Hello, 6.087 students

- In style of “Hello, world!”
- `.c` file structure
- Syntax: comments, macros, basic declarations
- The `main()` function and function structure
- Expressions, order-of-operations
- Basic console I/O (`puts()`, etc.)

Structure of a `.c` file

```
/* Begin with comments about file contents */

Insert #include statements and preprocessor
definitions

Function prototypes and variable declarations

Define main() function
{
  Function body
}

Define other function
{
  Function body
}
⋮
```

Comments

- Comments: `/* this is a simple comment */`
- Can span multiple lines

```
/* This comment
   spans
   multiple lines */
```

- Completely ignored by compiler
- Can appear almost anywhere

```
/* hello.c -- our first C program

   Created by Daniel Weller, 01/11/2010 */
```

The `#include` macro

- Header files: constants, functions, other declarations
- **#include** <stdio.h> – read the contents of the *header file* `stdio.h`
- `stdio.h`: standard I/O functions for console, files

```
/* hello.c -- our first C program

   Created by Daniel Weller, 01/11/2010 */

#include <stdio.h> /* basic I/O facilities */
```

More about header files

- `stdio.h` – part of the C Standard Library
  - other important header files: `ctype.h`, `math.h`, `stdlib.h`, `string.h`, `time.h`
  - For the ugly details: visit `http://www.unix.org/single_unix_specification/` (registration required)
- Included files must be on *include path*
  - `-I`*`directory`* with `gcc`: specify additional include directories
  - standard include directories assumed by default
- **#include** `"stdio.h"` – searches `./` for `stdio.h` first

Declaring variables

- Must declare variables before use
- Variable declaration:

```c
int n;
float phi;
```

- `int` - integer data type
- `float` - floating-point data type
- Many other types (more next lecture…)

Initializing variables

- Uninitialized, variable assumes a default value
- Variables initialized via assignment operator:

```
n = 3;
```

- Can also initialize at declaration:

```
float phi = 1.6180339887;
```

- Can declare/initialize multiple variables at once:

```
int a, b, c = 0, d = 4;
```

Arithmetic expressions

Suppose `x` and `y` are variables

- `x+y`, `x-y`, `x*y`, `x/y`, `x%y`: binary arithmetic
- A simple statement:
  y = x+3*x/(y−4);
- Numeric literals like 3 or 4 valid in expressions
- Semicolon ends statement (not newline)
- `x += y`, `x -= y`, `x *= y`, `x /= y`, `x %= y`: arithmetic and assignment

Order of operations

- Order of operations:

Operator	Evaluation direction
`+`, `-` (sign)	right-to-left
`*`, `/`, `%`	left-to-right
`+`, `-`	left-to-right
`=`, `+=`, `-=`, `*=`, `/=`, `%=`	right-to-left

- Use parentheses to override order of evaluation

Order of operations

Assume $x = 2.0$ and $y = 6.0$. Evaluate the statement

```
float z = x+3*x/(y-4);
```

1. Evaluate expression in parentheses

   `float z = x+3*x/(y-4);` $\rightarrow$ `float z = x+3*x/2.0;`

Order of operations

Assume $x = 2.0$ and $y = 6.0$. Evaluate the statement

float z = x+3∗x/(y−4);

1. Evaluate expression in parentheses

   float z = x+3∗x/(y−4); $\rightarrow$ **float** z = x+3∗x/2.0;

2. Evaluate multiplies and divides, from left-to-right

   float z = x+3∗x/2.0; $\rightarrow$ **float** z = x+6.0/2.0; $\rightarrow$ **float** z = x+3.0;

Order of operations

Assume $x = 2.0$ and $y = 6.0$. Evaluate the statement

```
float z = x+3*x/(y-4);
```

1. Evaluate expression in parentheses

   float z = x+3∗x/(y−4); $\rightarrow$ **float** z = x+3∗x/2.0;

2. Evaluate multiplies and divides, from left-to-right

   float z = x+3∗x/2.0; $\rightarrow$ **float** z = x+6.0/2.0; $\rightarrow$ **float** z = x+3.0;

3. Evaluate addition

   float z = x+3.0; $\rightarrow$ **float** z = 5.0;

Order of operations

Assume $\mathrm{x} = 2.0$ and $\mathrm{y} = 6.0$. Evaluate the statement

float z = x+3$*$x/(y$-$4);

1. Evaluate expression in parentheses

   float z = x+3$*$x/(y$-$4); $\rightarrow$ **float** z = x+3$*$x/2.0;

2. Evaluate multiplies and divides, from left-to-right

   float z = x+3$*$x/2.0; $\rightarrow$ **float** z = x+6.0/2.0; $\rightarrow$ **float** z = x+3.0;

3. Evaluate addition

   float z = x+3.0; $\rightarrow$ **float** z = 5.0;

4. Perform initialization with assignment

   Now, $\mathrm{z} = 5.0$.

Order of operations

Assume $x = 2.0$ and $y = 6.0$. Evaluate the statement

`float z = x+3*x/(y-4);`

1. Evaluate expression in parentheses

   `float z = x+3*x/(y-4);` $\rightarrow$ `float z = x+3*x/2.0;`

2. Evaluate multiplies and divides, from left-to-right

   `float z = x+3*x/2.0;` $\rightarrow$ `float z = x+6.0/2.0;` $\rightarrow$ `float z = x+3.0;`

3. Evaluate addition

   `float z = x+3.0;` $\rightarrow$ `float z = 5.0;`

4. Perform initialization with assignment

   Now, $z = 5.0$.

How do I insert parentheses to get $z = 4.0$?

Order of operations

Assume $\mathrm{x} = 2.0$ and $\mathrm{y} = 6.0$. Evaluate the statement

`float z = x+3*x/(y-4);`

1. Evaluate expression in parentheses

   `float z = x+3*x/(y-4);` $\rightarrow$ `float z = x+3*x/2.0;`

2. Evaluate multiplies and divides, from left-to-right

   `float z = x+3*x/2.0;` $\rightarrow$ `float z = x+6.0/2.0;` $\rightarrow$ `float z = x+3.0;`

3. Evaluate addition

   `float z = x+3.0;` $\rightarrow$ `float z = 5.0;`

4. Perform initialization with assignment

   Now, $\mathrm{z} = 5.0$.

How do I insert parentheses to get $\mathrm{z} = 4.0$?

`float z = (x+3*x)/(y-4);`

Function prototypes

- Functions also must be declared before use
- Declaration called *function prototype*
- Function prototypes:

  int factorial (**int**); or **int** factorial (**int** n);
- Prototypes for many common functions in header files for C Standard Library

Function prototypes

- General form:
  return_type function_name(arg1,arg2,...);
- Arguments: local variables, values passed from caller
- Return value: single value returned to caller when function exits
- `void` – signifies no return value/arguments

```
int rand(void);
```

The `main()` function

- `main()`: entry point for C program
- Simplest version: no inputs, outputs 0 when successful, and nonzero to signal some error

```c
int main(void);
```

- Two-argument form of `main()`: access command-line arguments

```c
int main(int argc, char **argv);
```

- More on the `char **argv` notation later this week...

Function definitions

```
Function declaration
{
  declare variables;
  program statements;
}
```

- Must match prototype (if there is one)
  - variable names don't have to match
  - no semicolon at end
- Curly braces define a *block* – region of code
  - Variables declared in a block exist only in that block
- Variable declarations before any other statements

Our `main()` function

```c
/* The main() function */
int main(void) /* entry point */
{
  /* write message to console */
  puts("hello, 6.087 students");

  return 0; /* exit (0 => success) */
}
```

- `puts()`: output text to console window (stdout) and end the line
- String literal: written surrounded by double quotes
- **return** 0;
  exits the function, returning value 0 to caller

Alternative `main()` function

- Alternatively, store the string in a variable first:

```c
int main(void) /* entry point */
{
  const char msg[] = "hello, 6.087 students";

  /* write message to console */
  puts(msg);
```

- `const` keyword: qualifies variable as constant
- `char`: data type representing a single character; written in quotes: `'a'`, `'3'`, `'n'`
- `const char msg[]`: a constant array of characters

More about strings

- Strings stored as character array
- Null-terminated (last character in array is `'\0'` null)
    - Not written explicitly in string literals
- Special characters specified using `\` (escape character):
    - `\\` – backslash, `\'` – apostrophe, `\"` – quotation mark
    - `\b`, `\t`, `\r`, `\n` – backspace, tab, carriage return, linefeed
    - `\ooo`, `\xhh` – octal and hexadecimal ASCII character codes, *e.g.* `\x41` – `'A'`, `\060` – `'0'`

Console I/O

- stdout, stdin: console output and input streams
- `puts(`*`string`*`)`: print string to stdout
- `putchar(`*`char`*`)`: print character to stdout
- *`char`* `= getchar()`: return character from stdin
- *`string`* `= gets(`*`string`*`)`: read line from stdin into string
- Many others - later this week

Preprocessor macros

- Preprocessor macros begin with `#` character
  #include <stdio.h>
- **#define** msg `"hello, 6.087 students"`
  defines *msg* as "`hello, 6.087 students`" throughout source file
- many constants specified this way

Defining expression macros

- **#define** can take arguments and be treated like a function

  ```
  #define add3(x,y,z) ((x)+(y)+(z))
  ```

- parentheses ensure order of operations
- compiler performs inline replacement; not suitable for recursion

Conditional preprocessor macros

- **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, **#endif** conditional preprocessor macros, can control which lines are compiled
  - evaluated before code itself is compiled, so conditions must be preprocessor defines or literals
  - the `gcc` option `-D`*`name`*`=`*`value`* sets a preprocessor define that can be used
  - Used in header files to ensure declarations happen only once

Conditional preprocessor macros

- **#pragma**
  preprocessor directive
- **#error**, **#warning**
  trigger a custom compiler error/warning
- **#undef** msg
  remove the definition of `msg` at compile time

Compiling our code

After we save our code, we run `gcc`:

```
athena%[1] gcc -g -O0 -Wall hello.c -o
hello.o
```

Assuming that we have made no errors, our compiling is complete.

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Running our code

Or, in gdb,

```
athena%[1] gdb hello.o
⋮
Reading symbols from hello.o...done.
(gdb) run
Starting program:  hello.o
hello, 6.087 students

Program exited normally.
(gdb) quit
athena%
```

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Summary

Topics covered:

- How to edit, compile, and debug C programs
- C programming fundamentals:
  - comments
  - preprocessor macros, including `#include`
  - the `main()` function
  - declaring and initializing variables, scope
  - using `puts()` – calling a function and passing an argument
  - returning from a function

6.087 Practical Programming in C
IAP 2010

6.087 Lecture 2 – January 12, 2010

- Review
- Variables and data types
- Operators
- Epilogue

Review: C Programming language

- C is a fast, small,general-purpose,platform independent programming language.
- C is used for systems programming (*e.g.,* compilers and interpreters, operating systems, database systems, microcontrollers *etc.*)
- C is static (compiled), typed, structured and imperative.
- "C is quirky, flawed, and an enormous success."–Ritchie

Review: Basics

- Variable declarations: `int i; float f;`
- Intialization: `char c='A'; int x=y=10;`
- Operators: +,−,∗,/,%
- Expressions: `int x,y,z; x=y*2+z*3;`
- Function: `int factorial (int n); /*function takes int, returns int */`

6.087 Lecture 2 – January 12, 2010

- Review
- Variables and data types
- Operators
- Epilogue

Definitions

Datatypes:

- The **datatype** of an object in memory determines the set of values it can have and what operations that can be performed on it.
- C is a *weakly* typed language. It allows implicit conversions as well as forced (potentially dangerous) casting.

Operators:

- **Operators** specify how an object can be manipulated (*e.g.*,, numeric vs. string operations).
- operators can be unary(*e.g.,* -,++),binary (*e.g.,* +,-,*,/),ternary (?:)

Definitions (contd.)

Expressions:

- An expression in a programming language is a combination of values, variables, operators, and functions

Variables:

- A variable is as named link/reference to a value stored in the system's memory or an expression that can be evaluated.

Consider: `int x=0,y=0; y=x+2;`.

- x, y are variables
- $y = x + 2$ is an expression
- $+$ is an operator.

Variable names

Naming rules:

- Variable names can contain letters,digits and _
- Variable names should start with letters.
- Keywords (*e.g.,* for,while *etc.*) cannot be used as variable names
- Variable names are case sensitive. `int x; int X` declares two different variables.

Pop quiz (correct/incorrect):

- `int money$owed;` (incorrect: cannot contain $)
- `int total_count` (correct)
- `int score2` (correct)
- `int 2ndscore` (incorrect: must start with a letter)
- `int long` (incorrect: cannot use keyword)

Data types and sizes

C has a small family of datatypes.

- Numeric (int,float,double)
- Character (char)
- User defined (struct,union)

Numeric data types

Depending on the precision and range required, you can use one of the following datatypes.

	signed	unsigned
short	**short int** x;**short** y;	**unsigned short** x;**unsigned short int** y;
default	**int** x;	**unsigned int** x;
long	**long** x;	**unsigned long** x;
float	**float** x;	N/A
double	**double** x;	N/A
char	**char** x; **signed char** x;	**unsigned char** x;

- The unsigned version has roughly double the range of its signed counterparts.
- Signed and unsigned characters differ only when used in arithmetic expressions.
- Titbit: Flickr changed from unsigned long $(2^{32} - 1)$ to string two years ago.

Big endian vs. little endian

The individual sizes are machine/compiler dependent. However, the following is guaranteed:

`sizeof(char)<sizeof(short)<=sizeof(int)<=sizeof(long)` and

`sizeof(char)<sizeof(short)<=sizeof(float)<=sizeof(double)`

"NUXI" problem: For numeric data types that span multiple bytes, the order of arrangement of the individual bytes is important. Depending on the device architecture, we have "big endian" and "little endian" formats.

Big endian vs. little endian (cont.)

- Big endian: the **most** significant bits (MSBs) occupy the lower address. This representation is used in the powerpc processor. Networks generally use big-endian order, and thus it is called **network order**.
- Little endian : the **least** signficant bits (LSBs) occupy the lower address. This representation is used on all x86 compatible processors.

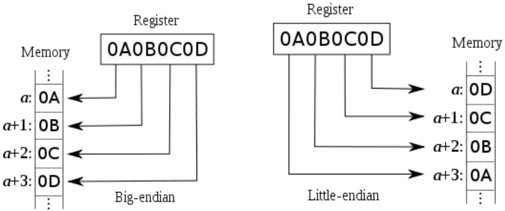


Figure: (from http://en.wikipedia.org/wiki/Little_endian)

Constants

Constants are literal/fixed values assigned to variables or used directly in expressions.

Datatype	example	meaning
	int i=3;	integer
	long l=3;	long integer
integer	**unsigned long** ul= 3UL;	unsigned long
	int i=0xA;	hexadecimal
	int i=012;	octal number
	float pi=3.14159	float
floating point	**float** pi=3.141F	float
	double pi=3.1415926535897932384L	double

Constants (contd.)

Datatype	example	meaning

character	`'A'`
	`'\x41'`
	`'\0101'`
		character
		specified in hex
		specified in octal

string	`"hello world"` `"hello""world"`	string literal same as "hello world"
enumeration	**enum** BOOL {NO,YES} **enum** COLOR {R=1,G,B,Y=10}	NO=0,YES=1 G=2,B=3

Declarations

The general format for a declaration is
type variable-name [=*value*] .
Examples:

- **char** x; /* uninitialized */
- **char** x='A'; /* intialized to 'A'*/
- **char** x='A',y='B'; /*multiple variables initialized */
- **char** x=y='Z';/*multiple initializations */

Pop quiz II

- **int** x=017;**int** y=12; /*is x>y?*/
- **short int** s=0xFFFF12; /*correct?*/
- **char** c=−1;**unsigned char** uc=−1; /*correct?*/
- puts(`"hel"+"lo"`);puts(`"hel""lo"`);/*which is correct?*/
- **enum** sz{S=0,L=3,XL}; /*what is the value of XL?*/
- **enum** sz{S=0,L=−3,XL}; /*what is the value of XL?*/

- Review
- Variables and data types
- **Operators**
- Epilogue

Arithmetic operators

operator	meaning	examples
+	addition	x=3+2; /*constants*/ y+z; /*variables*/ x+y+2; /*both*/
-	subtraction	3−2; /*constants*/ int x=y−z; /*variables*/ y−2−z; /*both*/
*	multiplication	int x=3*2; /*constants*/ int x=y*z; /*variables*/ x*y*2; /*both*/

Arithmetic operators (contd.)

operator	meaning	examples
/	division	**float** x=3/2; /*produces x=1 (int /) */ **float** x=3.0/2 /*produces x=1.5 (float /) */ **int** x=3.0/2; /*produces x=1 (int conversion)*/
%	modulus (remainder)	**int** x=3%2; /*produces x=1*/ **int** y=7;**int** x=y%4; /*produces 3*/ **int** y=7;**int** x=y%10; /*produces 7*/

Relational Operators

Relational operators compare two operands to produce a 'boolean' result. In C any non-zero value (1 by convention) is considered to be 'true' and 0 is considered to be false.

operator	meaning	examples
>	greater than	3>2; /*evaluates to 1 */ 2.99>3 /*evaluates to 0 */
>=	greater than or equal to	3>=3; /*evaluates to 1 */ 2.99>=3 /*evaluates to 0 */
<	lesser than	3<3; /*evaluates to 0 */ 'A'<'B' /*evaluates to 1*/
<=	lesser than or equal to	3<=3; /*evaluates to 1 */ 3.99<3 /*evaluates to 0 */

Relational Operators

Testing equality is one of the most commonly used relational operator.

operator	meaning	examples
==	equal to	3==3; /*evaluates to 1 */ 'A'=='a' /*evaluates to 0 */
!=	not equal to	3!=3; /*evaluates to 0 */ 2.99!=3 /*evaluates to 1 */

Gotchas:

- Note that the "==" equality operator is different from the "=", assignment operator.
- Note that the "==" operator on float variables is tricky because of finite precision.

Logical operators

operator	meaning	examples
&&	AND	((9/3)==3) && (2*3==6); /*evaluates to 1 */ ('A'=='a') && (3==3) /*evaluates to 0 */
||	OR	2==3 || 'A'=='A'; /*evaluates to 1 */ 2.99>=3 || 0 /*evaluates to 0 */
!	NOT	!(3==3); /*evaluates to 0 */ !(2.99>=3) /*evaluates to 1 */

Short circuit: The evaluation of an expression is discontinued if the value of a conditional expression can be determined early. Be careful of any side effects in the code.
Examples:

- (3==3) || ((c=getchar())=='y'). The second expression is not evaluated.
- (0) && ((x=x+1)>0) . The second expression is not evaluated.

Increment and decrement operators

Increment and decrement are common arithmetic operation. C provides two short cuts for the same.

Postfix

- x++ is a short cut for x=x+1
- x−− is a short cut for x=x−1
- y=x++ is a short cut for y=x;x=x+1. x is evaluated **before** it is incremented.
- y=x−− is a short cut for y=x;x=x−1. x is evaluated **before** it is decremented.

Increment and decrement operators

Prefix:

- ++x is a short cut for x=x+1
- $--$x is a short cut for x=x$-$1
- y=++x is a short cut for x=x+1;y=x;. x is evaluate **after** it is incremented.
- y=$--$x is a short cut for x=x$-$1;y=x;. x is evaluate **after** it is decremented.

Bitwise Operators

<table>
<tr><th>operator</th><th>meaning</th><th>examples</th></tr>
<tr><td>&</td><td>AND</td><td>0x77 & 0x3; /*evaluates to 0x3 */
0x77 & 0x0; /*evaluates to 0 */</td></tr>
<tr><td>|</td><td>OR</td><td>0x700 | 0x33; /*evaluates to 0x733 */
0x070 | 0 /*evaluates to 0x070 */</td></tr>
<tr><td>^</td><td>XOR</td><td>0x770 ^ 0x773; /*evaluates to 0x3 */
0x33 ^ 0x33; /*evaluates to 0 */</td></tr>
<tr><td>«</td><td>left shift</td><td>0x01<<4; /*evaluates to 0x10 */
1<<2; /*evaluates to 4 */</td></tr>
<tr><td>»</td><td>right shift</td><td>0x010>>4; /*evaluates to 0x01 */
4>>1 /*evaluates to 2 */</td></tr>
</table>

Notes:

- AND is true only if **both** operands are true.
- OR is true if **any** operand is true.
- XOR is true if **only one** of the operand is true.

Assignment Operators

Another common expression type found while programming in C is of the type var = var (op) expr

- x=x+1
- x=x*10
- x=x/2

C provides compact assignment operators that can be used instead.

- x+=1 /*is the same as x=x+1*/
- x−=1 /*is the same as x=x−1*/
- x*=10 /*is the same as x=x*10 */
- x/=2 /* is the same as x=x/2
- x%=2 /*is the same as x=x%2

Conditional Expression

A common pattern in C (and in most programming) languages is the following:

```
if (cond)
  x=<expra>;
else
  x=<exprb>;
```

C provides *syntactic sugar* to express the same using the ternary operator '?:'

`sign=x>0?1:-1;`
`if (x>0)`
`  sign=1`
`else`
`  sign=-1`
	`isodd=x%2==1?1:0;`
	`if (x%2==1)`
	`  isodd=1`
	`else`
	`  isodd=0`

Notice how the ternary operator makes the code shorter and easier to understand (syntactic sugar).

- Review
- Variables and data types
- Operators
- Epilogue

Type Conversions

When variables are promoted to higher precision, data is preserved. This is automatically done by the compiler for mixed data type expressions.

```
int i;
float f;
f=i+3.14159; /*i is promoted to float, f=(float)i+3.14159*/
```

Another conversion done automatically by the compiler is 'char' → 'int'. This allows comparisons as well as manupilations of character variables.

```
isupper=(c>='A' && c<='Z')?1:0; /*c and literal constants
                                  are converted to int*/
if(!isupper)
  c=c-'a'+'A'; /*subtraction is possible
                 because of integer conversion*/
```

As a rule (with exceptions), the compiler promotes each term in an binary expression to the highest precision operand.

Precedence and Order of Evaluation

- ++,--,(cast),sizeof have the highest priority
- *,/,% have higher priority than +,-
- ==,!= have higher priority than &&,||
- assignment operators have very low priority

Use () generously to avoid ambiguities or side effects associated with precendence of operators.

- y=x*3+2 /*same as y=(x*3)+2*/
- x!=0 && y==0 /*same as (x!=0) && (y==0)*/
- d= c>='0'&& c<='9'/*same as d=(c>='0') && (c<='9')*/

6.087 Practical Programming in C
IAP 2010

6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Review: Definitions

- Variable - name/reference to a stored value (usually in memory)
- Data type - determines the size of a variable in memory, what values it can take on, what operations are allowed
- Operator - an operation performed using 1-3 variables
- Expression - combination of literal values/variables and operators/functions

Review: Data types

- Various sizes (**char**, **short**, **long**, **float**, **double**)
- Numeric types - **signed**/**unsigned**
- Implementation - little or big endian
- Careful mixing and converting (casting) types

Review: Operators

- Unary, binary, ternary (1-3 arguments)
- Arithmetic operators, relational operators, binary (bitwise and logical) operators, assignment operators, etc.
- Conditional expressions
- Order of evaluation (precedence, direction)

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Blocks and compound statements

- A simple statement ends in a semicolon:

  `z = foo(x+y);`

- Consider the multiple statements:

```
temp = x+y;
z = foo(temp);
```

- Curly braces – combine into compound statement/*block*

Blocks

- Block can substitute for simple statement
- Compiled as a single unit
- Variables can be declared inside

```
{
  int temp = x+y;
  z = foo(temp);
}
```

- Block can be empty `{ }`
- No semicolon at end

Nested blocks

- Blocks nested inside each other

```
{
  int temp = x+y;
  z = foo(temp);
  {
    float temp2 = x*y;
    z += bar(temp2);
  }
}
```

6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Control conditions

- Unlike C++ or Java, no *boolean* type (in C89/C90)
  - in C99, bool type available (use `stdbool.h`)
- Condition is an expression (or series of expressions)
  e.g. `n < 3` or `x < y || z < y`
- Expression is non-zero $\Rightarrow$ condition true
- Expression must be numeric (or a pointer)

```c
const char str[] = "some text";
if (str) /* string is not null */
  return 0;
```

Conditional statements

- The `if` statement
- The `switch` statement

The `if` statement

```
if (x % 2)
  y += x/2;
```

- Evaluate condition
  `if (x % 2 == 0)`
- If true, evaluate inner statement
  `y += x/2;`
- Otherwise, do nothing

The `else` keyword

```
if (x % 2 == 0)
  y += x/2;
else
  y += (x+1)/2;
```

- Optional
- Execute statement if condition is false

  y += (x+1)/2;
- Either inner statement may be block

The `else if` keyword

```
if (x % 2 == 0)
  y += x/2;
else if (x % 4 == 1)
  y += 2*((x+3)/4);
else
  y += (x+1)/2;
```

- Additional alternative control paths
- Conditions evaluated in order until one is met; inner statement then executed
- If multiple conditions true, only first executed
- Equivalent to nested `if` statements

Nesting `if` statements

```
if (x % 4 == 0)
  if (x % 2 == 0)
    y = 2;
else
  y = 1;
```

To which `if` statement does the `else` keyword belong?

Nesting `if` statements

To associate `else` with outer `if` statement: use braces

```
if (x % 4 == 0) {
  if (x % 2 == 0)
    y = 2;
} else
  y = 1;
```

The `switch` statement

- Alternative conditional statement
- Integer (or character) variable as input
- Considers cases for value of variable

```c
switch (ch) {
  case 'Y': /* ch == 'Y' */
    /* do something */
    break;
  case 'N': /* ch == 'N' */
    /* do something else */
    break;
  default: /* otherwise */
    /* do a third thing */
    break;
}
```

Multiple cases

- Compares variable to each case in order
- When match found, starts executing inner code until `break;` reached
- Execution “falls through” if `break;` not included

```c
switch (ch) {
  case 'Y':
  case 'y':
    /* do something if
       ch == 'Y' or
       ch == 'y' */
    break;
}
```

```c
switch (ch) {
  case 'Y':
    /* do something if
       ch == 'Y' */
  case 'N':
    /* do something if
       ch == 'Y' or
       ch == 'N' */
    break;
}
```

The `switch` statement

- Contents of `switch` statement a block
- Case labels: different entry points into block
- Similar to labels used with `goto` keyword (next lecture...)

Loop statements

- The `while` loop
- The `for` loop
- The `do-while` loop
- The `break` and `continue` keywords

The `while` loop

```
while (/* condition */)
  /* loop body */
```

- Simplest loop structure – evaluate body as long as condition is true
- Condition evaluated first, so body may never be executed

The `for` loop

```
int factorial(int n) {
  int i, j = 1;
  for (i = 1; i <= n; i++)
    j *= i;
  return j;
}
```

- The “counting” loop
- Inside parentheses, three expressions, separated by semicolons:
  - Initialization: `i = 1`
  - Condition: `i <= n`
  - Increment: `i++`
- Expressions can be empty (condition assumed to be “true”)

The `for` loop

Equivalent to `while` loop:

```c
int factorial(int n) {
  int j = 1;
  int i = 1; /* initialization */
  while (i <= n /* condition */) {
    j *= i;
    i++; /* increment */
  }
  return j;
}
```

The `for` loop

- Compound expressions separated by commas

```
int factorial(int n) {
  int i, j;
  for (i = 1, j = 1; i <= n; j *= i, i++)
    ;
  return j;
}
```

- Comma: operator with lowest precedence, evaluated left-to-right; not same as between function arguments

The `do-while` loop

```c
char c;
do {
  /* loop body */
  puts("Keep going? (y/n) ");
  c = getchar();
  /* other processing */
} while (c == 'y' && /* other conditions */);
```

- Differs from `while` loop – condition evaluated after each iteration
- Body executed at least once
- Note semicolon at end

The `break` keyword

- Sometimes want to terminate a loop early
- **break**; exits innermost loop or `switch` statement to exit early
- Consider the modification of the `do-while` example:

```c
char c;
do {
  /* loop body */
  puts("Keep going? (y/n) ");
  c = getchar();
  if (c != 'y')
    break;
  /* other processing */
} while (/* other conditions */);
```

The `continue` keyword

- Use to skip an iteration
- **continue**; skips rest of innermost loop body, jumping to loop condition
- Example:

```c
#define min(a,b) ((a) < (b) ? (a) : (b))

int gcd(int a, int b) {
  int i, ret = 1, minval = min(a,b);
  for (i = 2; i <= minval; i++) {
    if (a % i) /* i not divisor of a */
      continue;
    if (b % i == 0) /* i is divisor of both a and b */
      ret = i;
  }
  return ret;
}
```

6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Functions

- Already seen some functions, including `main()`:

```
int main(void) {
  /* do stuff */
  return 0; /* success */
}
```

- Basic syntax of functions explained in Lecture 1
- How to write a program using functions?

Divide and conquer

- Conceptualize how a program can be broken into smaller parts
- Let's design a program to solve linear Diophantine equation ($ax + by = c$,x, y: integers):

```
get a, b, c from command line
compute g = gcd(a,b)
if (c is not a multiple of the gcd)
  no solutions exist; exit
run Extended Euclidean algorithm on a, b
rescale x and y output by (c/g)
print solution
```

- Extended Euclidean algorithm: finds integers x, y s.t.

$$ax + by = \text{gcd}(a, b).$$

Computing the gcd

- Compute the gcd using the Euclidean algorithm:

```
int gcd(int a, int b) {
  while (b) { /* if a < b, performs swap */
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

- Algorithm relies on $\gcd(a, b) = \gcd(b, a \bmod b)$, for natural numbers $a > b$.

[Knuth, D. E. The Art of Computer Programming, Volume 1: Fundamental Algorithms. 3rd ed. Addison-Wesley, 1997.]

Extended Euclidean algorithm

Pseudocode for Extended Euclidean algorithm:

```
Initialize state variables (x,y)
if (a < b)
  swap(a,b)
while (b > 0) {
  compute quotient, remainder
  update state variables (x,y)
}
return gcd and state variables (x,y)
```

[Menezes, A. J., et al. Handbook of Applied Cryptography. CRC Press, 1996.]

Returning multiple values

- Extended Euclidean algorithm returns gcd, and two other state variables, x and y
- Functions only return (up to) one value
- Solution: use *global* variables
- Declare variables for other outputs outside the function
    - variables declared outside of a function block are globals
    - persist throughout life of program
    - can be accessed/modified in any function

Divide and conquer

- Break down problem into simpler sub-problems
- Consider iteration and recursion
  - How can we implement gcd(a,b) recursively?
- Minimize transfer of state between functions
- Writing pseudocode first can help

6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Programming modules in C

- C programs do not need to be monolithic
- Module: interface and implementation
  - interface: header files
  - implementation: auxilliary source/object files
- Same concept carries over to external libraries (next week...)

The Euclid module

- Euclid's algorithms useful in many contexts
- Would like to include functionality in many programs
- Solution: make a module for Euclid's algorithms
- Need to write header file (`.h`) and source file (`.c`)

The source: euclid.c

Implement gcd() in euclid.c:

```
/* The gcd() function */
int gcd(int a, int b) {
  while (b) { /* if a < b, performs swap */
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

Extended Euclidean algorithm implemented as ext_euclid(), also in euclid.c

The `extern` keyword

- Need to inform other source files about functions/global variables in `euclid.c`
- For functions: put function prototypes in a header file
- For variables: re-declare the global variable using the `extern` keyword in header file
- `extern` informs compiler that variable defined somewhere else
- Enables access/modifying of global variable from other source files

The header: euclid.h

Header contains prototypes for gcd() and ext_euclid():

```c
/* ensure included only once */
#ifndef __EUCLID_H__
#define __EUCLID_H__

/* global variables (declared in euclid.c) */
extern int x, y;

/* compute gcd */
int gcd(int a, int b);

/* compute g = gcd(a,b) and solve ax+by=g */
int ext_euclid(int a, int b);

#endif
```

Using the Euclid module

- Want to be able to call `gcd()` or `ext_euclid()` from the main file `diophant.c`
- Need to include the header file `euclid.h`:
  #include `"euclid.h"` (file in ". ", not search path)
- Then, can call as any other function:

```
/* compute g = gcd(a,b) */
g = gcd(a,b);

/* compute x and y using Extended Euclidean alg. */
g = ext_euclid(a,b);
```

- Results in global variables `x` and `y`

```
/* rescale so ax+by = c */
grow = c/g;
x *= grow;
y *= grow;
```

Compiling with the Euclid module

- Just compiling `diophant.c` is insufficient
- The functions `gcd()` and `ext_euclid()` are defined in `euclid.c`; this source file needs to be compiled, too
- When compiling the source files, the outputs need to be linked together into a single output
- One call to `gcc` can accomplish all this:

```
athena%[1] gcc -g -O0 -Wall diophant.c
euclid.c -o diophant.o
```

- `diophant.o` can be run as usual

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Variable scope

- *scope* – the region in which a variable is valid
- Many cases, corresponds to block with variable's declaration
- Variables declared outside of a function have global scope
- Function definitions also have scope

An example

What is the scope of each variable in this example?

```c
int nmax = 20;

/* The main() function */
int main(int argc, char ** argv) /* entry point */
{
  int a = 0, b = 1, c, n;
  printf("%3d: %d\n",1,a);
  printf("%3d: %d\n",2,b);
  for (n = 3; n <= nmax; n++) {
    c = a + b; a = b; b = c;
    printf("%3d: %d\n",n,c);
  }
  return 0; /* success */
}
```

Scope and nested declarations

How many lines are printed now?

```c
int nmax = 20;

/* The main() function */
int main(int argc, char ** argv) /* entry point */
{
  int a = 0, b = 1, c, n, nmax = 25;
  printf("%3d: %d\n",1,a);
  printf("%3d: %d\n",2,b);
  for (n = 3; n <= nmax; n++) {
    c = a + b; a = b; b = c;
    printf("%3d: %d\n",n,c);
  }
  return 0; /* success */
}
```

Static variables

- `static` keyword has two meanings, depending on where the static variable is declared
- Outside a function, `static` variables/functions only visible within that file, not globally (cannot be `extern`'ed)
- Inside a function, `static` variables:
  - are still local to that function
  - are initialized only during program initialization
  - do not get reinitialized with each function call

```
static int somePersistentVar = 0;
```

Register variables

- During execution, data processed in *registers*
- Explicitly store commonly used data in registers – minimize load/store overhead
- Can explicitly declare certain variables as registers using `register` keyword
  - must be a simple type (implementation-dependent)
  - only local variables and function arguments eligible
  - excess/unallowed register declarations ignored, compiled as regular variables
- Registers do not reside in addressed memory; pointer of a register variable illegal

Example

Variable scope example, revisited, with `register` variables:

```c
/* The main() function */
int main(register int argc, register char ** argv)
{
  register int a = 0, b = 1, c, n, nmax = 20;
  printf("%3d: %d\n",1,a);
  printf("%3d: %d\n",2,b);
  for (n = 3; n <= nmax; n++) {
    c = a + b; a = b; b = c;
    printf("%3d: %d\n",n,c);
  }
  return 0; /* success */
}
```

Summary

Topics covered:

- Controlling program flow using conditional statements and loops
- Dividing a complex program into many simpler sub-programs using functions and modular programming techniques
- Variable scope rules and `extern`, `static`, and `register` variables

6.087 Practical Programming in C
January (IAP) 2010

6.087 Lecture 4 – January 14, 2010

- Review
- Control flow
- I/O
  - Standard I/O
  - String I/O
  - File I/O

Blocks

- Blocks combine multiple statements into a single unit.
- Can be used when a single statement is expected.
- Creates a local scope (variables declared inside are local to the block).
- Blocks can be nested.

```
{
  int x=0;
  {
    int y=0; /*both x and y visible*/
  }
  /*only x visible*/
}
```

Conditional blocks

if ... **else**..**else if** is used for conditional branching of execution

```
if (cond)
{
  /*code executed if cond is true*/
}
else
{
  /*code executed if cond is false*/
}
```

Conditional blocks

switch..case is used to test multiple conditions (more efficient than if else ladders).

```
switch(opt)
{
  case 'A':
    /*execute if opt=='A'*/
    break;
  case 'B':
  case 'C':
    /*execute if opt=='B' || opt=='C'*/
  default:
}
```

Iterative blocks

- **while** loop tests condition before execution of the block.
- **do**..**while** loop tests condition after execution of the block.
- **for** loop provides initialization, testing and iteration together.

6.087 Lecture 4 – January 14, 2010

- Review
- Control flow
- I/O
  - Standard I/O
  - String I/O
  - File I/O

goto

- **goto** allows you to jump **unconditionally** to arbitrary part of your code (within the same function).
- the location is identified using a label.
- a label is a named location in the code. It has the same form as a variable followed by a ':'

```
start:
{
  if (cond)
    goto outside;
  /*some code*/
  goto start;
}
outside:
/*outside block*/
```

Spaghetti code

Dijkstra. *Go To Statement Considered Harmful*.
Communications of the ACM 11(3),1968

- Excess use of **goto** creates *sphagetti code*.
- Using **goto** makes code harder to read and debug.
- Any code that uses **goto** can be written without using one.

error handling

Language like C++ and Java provide exception mechanism to recover from errors. In C, **goto** provides a convenient way to exit from nested blocks.

```
for (..)
{
  for (..)
  {
    if (error_cond)
      goto error;
      /*skips 2 blocks*/
  }
}
error:
```

```
cont_flag=1;
for (..)
{
  for (init;cont_flag;iter)
  {
    if (error_cond)
    {
      cont_flag=0;
      break;
    }
    /*inner loop*/
  }
  if (!cont_flag) break;
  /*outer loop*/
}
```

6.087 Lecture 4 – January 14, 2010

- Review
- Control flow
- I/O
  - Standard I/O
  - String I/O
  - File I/O

Preliminaries

- Input and output facilities are provided by the standard library `<stdio.h>` and not by the language itself.
- A text stream consists of a series of lines ending with `'\n'`. The standard library takes care of conversion from `'\r\n'–'\n'`
- A binary stream consists of a series of raw bytes.
- The streams provided by standard library are **buffered**.

Standard input and output

```
int putchar(int)
```

- putchar(c) puts the character c on the *standard output*.
- it returns the character printed or EOF on error.

```
int getchar()
```

- returns the next character from *standard input*.
- it returns EOF on error.

Standard input and output

What does the following code do?

```
int main()
{
  char c;
  while((c=getchar())!=EOF)
  {
    if(c>='A' && c<='Z')
      c=c-'A'+'a';
    putchar(c);
  }
  return 0;
}
```

To use a file instead of standard input, use '<' operator (*nix).

- Normal invocation: ./a.out
- Input redirection: a.out < file.txt. Treats file.txt as source of standard input.This is an OS feature, not a language feature.

Standard output:formatted

```
int printf(char format[],arg1,arg2,...)
```

- printf() can be used for formatted output.
- It takes in a **variable** number of arguments.
- It returns the number of characters printed.
- The format can contain literal strings as well as format specifiers (starts with %).

Examples:

```
printf("hello world\n");
printf("%d\n",10);/*format: %d (integer),argument:10*/
printf("Prices:%d and %d\n",10,20);
```

printf format specification

The format specification has the following components

%[flags][width][.precision][length]<type>

type:

type	meaning	example
d,i	integer	printf ("%d",10); /*prints 10*/
x,X	integer (hex)	printf ("%x",10); /*print 0xa*/
u	unsigned integer	printf ("%u",10); /*prints 10*/
c	character	printf ("%c",'A'); /*prints A*/
s	string	printf ("%s","hello"); /*prints hello*/
f	float	printf ("%f",2.3); /* prints 2.3*/
d	double	printf ("%d",2.3); /* prints 2.3*/
e,E	float(exp)	1e3,1.2E3,1E−3
%	literal %	printf ("%d %%",10); /*prints 10%*/

printf format specification (cont.)

%[flags][width][.precision][modifier]<type>

width:

format	output
printf ("%d",10)	"10"
printf ("%4d",10)	bb10 (b:space)
printf ("%s","hello")	hello
printf ("%7s","hello")	bbhello

printf format specification (cont.)

%[flags][width][.precision][modifier]<type>

flag:

format	output
printf (`"%d,%+d,%+d"`,10,−10)	10,+10,-10
printf (`"%04d"`,10)	0010
printf (`"%7s"`,`"hello"`)	bbhello
printf (`"%-7s"`,`"hello"`)	hellobb

printf format specification (cont.)

%[flags][width][.precision][modifier]<type>

precision:

format	output
printf ("%.2f,%.0f,1.141,1.141)	1.14,1
printf ("%.2e,%.0e,1.141,100.00)	1.14e+00,1e+02
printf ("%.4s","hello")	hell
printf ("%.1s","hello")	h

printf format specification (cont.)

%[flags][width][.precision][modifier]<type>

modifier:

modifier	meaning
h	interpreted as short. Use with i,d,o,u,x
l	interpreted as long. Use with i,d,o,u,x
L	interpreted as double. Use with e,f,g

Digression: character arrays

Since we will be reading and writing strings, here is a brief digression

- strings are represented as an array of characters
- C does not restrict the length of the string. The end of the string is specified using 0.

For instance, "hello" is represented using the array
`{'h','e','l','l','\0'}`.
Declaration examples:

- `char str[]="hello"; /*compiler takes care of size*/`
- `char str[10]="hello"; /*make sure the array is large enough*/`
- `char str[]={'h','e','l','l',0};`

Note: use \" if you want the string to contain ".

Digression: character arrays

Comparing strings: the header file <string.h> provides the function `int strcmp(char s[],char t[])` that compares two strings in dictionary order (lower case letters come **after** capital case).

- the function returns a value <0 if s comes before t
- the function return a value 0 if s is the same as t
- the function return a value >0 if s comes after t
- strcmp is case sensitive

Examples

- `strcmp("A","a") /*<0*/`
- `strcmp("IRONMAN","BATMAN") /*>0*/`
- `strcmp("aA","aA") /*==0*/`
- `strcmp("aA","a") /*>0*/`

Formatted input

`int scanf(char* format ,...)` is the input analog of printf.

- scanf reads characters from standard input, interpreting them according to format specification
- Similar to printf , scanf also takes variable number of arguments.
- The format specification is the same as that for printf
- When multiple items are to be read, each item is assumed to be separated by white space.
- It returns the number of **items** read or EOF.
- **Important:** scanf ignores white spaces.
- **Important:** Arguments have to be address of variables (pointers).

Formatted input

int scanf(char* format ,...) is the input analog of printf.
Examples:

printf ("%d",x)	scanf("%d",&x)
printf ("%10d",x)	scanf("%d",&x)
printf ("%f",f)	scanf("%f",&f)
printf ("%s",str)	scanf("%s",str) /*note no & required*/
printf ("%s",str)	scanf("%20s",str) /*note no & required*/
printf ("%s %s",fname,lname)	scanf("%20s %20s",fname,lname)

String input/output

Instead of writing to the standard output, the formatted data can be written to or read from character arrays.

```
int sprintf (char string [], char format[], arg1,arg2)
```

- The format specification is the same as printf.
- The output is written to string (does not check size).
- Returns the number of character written or negative value on error.

```
int sscanf(char str [], char format[], arg1,arg2)
```

- The format specification is the same as scanf;
- The input is read from str variable.
- Returns the number of items read or negative value on error.

File I/O

So far, we have read from the standard input and written to the standard output. C allows us to read data from text/binary files using fopen().

FILE∗ fopen(**char** name[],**char** mode[])

- mode can be "r" (read only),"w" (write only),"a" (append) among other options. "b" can be appended for binary files.
- fopen returns a **pointer** to the file stream if it exists or NULL otherwise.
- We don't need to know the details of the FILE data type.
- **Important:** The standard input and output are also FILE* datatypes (stdin,stdout).
- **Important:** stderr corresponds to standard error output(different from stdout).

File I/O(cont.)

```c
int fclose(FILE* fp)
```

- closes the stream (releases OS resources).
- fclose() is automatically called on all open files when program terminates.

File input

```
int getc(FILE* fp)
```

- reads a single character from the stream.
- returns the character read or EOF on error/end of file.

Note: getchar simply uses the standard input to read a character. We can implement it as follows:

```
#define getchar() getc(stdin)
```

```
char[] fgets(char line[], int maxlen,FILE* fp)
```

- reads a single line (upto maxlen characters) from the input stream (including linebreak).
- returns a pointer to the character array that stores the line (read-only)
- return NULL if end of stream.

File output

```c
int putc(int c,FILE* fp)
```

- writes a single character c to the output stream.
- returns the character written or EOF on error.

Note: putchar simply uses the standard output to write a character. We can implement it as follows:

```c
#define putchar(c) putc(c,stdout)
```

```c
int fputs(char line[], FILE* fp)
```

- writes a single line to the output stream.
- returns zero on success, EOF otherwise.

```c
int fscanf(FILE* fp,char format[],arg1,arg2)
```

- similar to scanf,sscanf
- reads items from input stream fp.

Command line input

- In addition to taking input from standard input and files, you can also pass input while invoking the program.
- *Command line parameters* are very common in *nix environment.
- So far, we have used `int main()` as to invoke the main function. However, main function can take arguments that are populated when the program is invoked.

Command line input (cont.)

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

- argc: count of arguments.
- argv[]: an array of pointers to each of the arguments
- note: the arguments include the name of the program as well.

Examples:

- ./cat a.txt b.txt (argc=3,argv[0]="cat" argv[1]="a.txt" argv[2]="b.txt"
- ./cat (argc=1,argv[0]="cat")

6.087 Practical Programming in C
January (IAP) 2010

6.087 Lecture 5 – January 15, 2010

- Review
- Pointers and Memory Addresses
  - Physical and Virtual Memory
  - Addressing and Indirection
  - Functions with Multiple Outputs
- Arrays and Pointer Arithmetic
- Strings
  - String Utility Functions
- Searching and Sorting Algorithms
  - Linear Search
  - A Simple Sort
  - Faster Sorting
  - Binary Search

Review: Unconditional jumps

- **goto** keyword: jump somewhere else in the same function
- Position identified using labels
- Example (**for** loop) using **goto**:

```
{
  int i = 0, n = 20; /* initialization */
  goto loop_cond;
loop_body:
  /* body of loop here */
  i++;
loop_cond:
  if (i < n) /* loop condition */
    goto loop_body;
}
```

- Excessive use of **goto** results in “spaghetti” code

Review: I/O Functions

- I/O provided by `stdio.h`, not language itself
- Character I/O: `putchar()`, `getchar()`, `getc()`, `putc()`, etc.
- String I/O: `puts()`, `gets()`, `fgets()`, `fputs()`, etc.
- Formatted I/O: `fprintf()`, `fscanf()`, etc.
- Open and close files: `fopen()`, `fclose()`
- File read/write position: `feof()`, `fseek()`, `ftell()`, etc.
- ...

Review: `printf()` and `scanf()`

- Formatted output:

  int printf (**char** format[], arg1, arg2, ...)

- Takes variable number of arguments
- Format specification:

  `%[flags][width][.precision][length]<type>`

  - types: d, i (int), u, o, x, X (unsigned int), e, E, f, F, g, G (double), c (char), s (string)
  - flags, width, precision, length - modify meaning and number of characters printed
- Formatted input: `scanf()` - similar form, takes pointers to arguments (except strings), ignores whitespace in input

Review: Strings and character arrays

- Strings represented in C as an array of characters (**char** [])
- String must be null-terminated (`'\0'` at end)
- Declaration:

  char str [] = `"I am a string."`; or

  char str[20] = `"I am a string."`;
- `strcpy()` - function for copying one string to another
- More about strings and string functions today...

6.087 Lecture 5 – January 15, 2010

- Review
- Pointers and Memory Addresses
  - Physical and Virtual Memory
  - Addressing and Indirection
  - Functions with Multiple Outputs
- Arrays and Pointer Arithmetic
- Strings
  - String Utility Functions
- Searching and Sorting Algorithms
  - Linear Search
  - A Simple Sort
  - Faster Sorting
  - Binary Search

Pointers and addresses

- Pointer: memory address of a variable
- Address can be used to access/modify a variable from anywhere
- Extremely useful, especially for data structures
- Well known for obfuscating code

Physical and virtual memory

- Physical memory: physical resources where data can be stored and accessed by your computer
  - cache
  - RAM
  - hard disk
  - removable storage
- Virtual memory: abstraction by OS, addressable space accessible by your code

Physical memory considerations

- Different sizes and access speeds
- Memory management – major function of OS
- Optimization – to ensure your code makes the best use of physical memory available
- OS moves around data in physical memory during execution
- Embedded processors – may be very limited

Virtual memory

- How much physical memory do I have?
  Answer: 2 MB (cache) + 2 GB (RAM) + 100 GB (hard drive) + . . .
- How much virtual memory do I have?
  Answer: <4 GB (32-bit OS), typically 2 GB for Windows, 3-4 GB for linux
- Virtual memory maps to different parts of physical memory
- Usable parts of virtual memory: *stack* and *heap*
  - stack: where declared variables go
  - heap: where dynamic memory goes

Addressing variables

- Every variable residing in memory has an address!
- What doesn't have an address?
  - register variables
  - constants/literals/preprocessor defines
  - expressions (unless result is a variable)
- How to find an address of a variable? The `&` operator

```
int n = 4;
double pi = 3.14159;
int *pn = &n; /* address of integer n */
double *ppi = &pi; /* address of double pi */
```

- Address of a variable of type *t* has type *t* *

Dereferencing pointers

- I have a pointer – now what?
- Accessing/modifying addressed variable: dereferencing/indirection operator *

```
/* prints "pi = 3.14159\n" */
printf("pi = %g\n",*ppi);

/* pi now equals 7.14159 */
*ppi = *ppi + *pn;
```

- Dereferenced pointer like any other variable
- null pointer, *i.e.* 0 (NULL): pointer that does not reference anything

Casting pointers

- Can explicitly cast any pointer type to any other pointer type

  ```
  ppi = (double *)pn; /* pn originally of type (int *) */
  ```

- Implicit cast to/from `void *` also possible (more next week...)
- Dereferenced pointer has new type, regardless of real type of data
- Possible to cause segmentation faults, other difficult-to-identify errors
  - What happens if we dereference `ppi` now?

Functions with multiple outputs

- Consider the Extended Euclidean algorithm `ext_euclid(a,b)` function from Wednesday's lecture
- Returns $\gcd(a,b)$, `x` and `y` s.t. $ax + by = \gcd(a,b)$
- Used global variables for `x` and `y`
- Can use pointers to pass back multiple outputs:

```c
int ext_euclid(int a, int b, int *x, int *y);
```

- Calling `ext_euclid()`, pass pointers to variables to receive `x` and `y`:

```c
int x, y, g;
/* assume a, b declared previously */
g = ext_euclid(a,b,&x,&y);
```

- Warning about `x` and `y` being used before initialized

Accessing caller's variables

- Want to write function to swap two integers
- Need to modify variables in caller to swap them
- Pointers to variables as arguments

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

- Calling `swap()` function:

```c
int a = 5, b = 7;
swap(&a, &b);
/* now, a = 7, b = 5 */
```

Variables passing out of scope

- What is wrong with this code?

```c
#include <stdio.h>

char * get_message() {
  char msg[] = "Aren't pointers fun?";
  return msg;
}

int main(void) {
  char * string = get_message();
  puts(string);
  return 0;
}
```

Variables passing out of scope

- What is wrong with this code?

```c
#include <stdio.h>

char * get_message() {
  char msg[] = "Aren't pointers fun?";
  return msg;
}

int main(void) {
  char * string = get_message();
  puts(string);
  return 0;
}
```

- Pointer invalid after variable passes out of scope

- Review
- Pointers and Memory Addresses
  - Physical and Virtual Memory
  - Addressing and Indirection
  - Functions with Multiple Outputs
- Arrays and Pointer Arithmetic
- Strings
  - String Utility Functions
- Searching and Sorting Algorithms
  - Linear Search
  - A Simple Sort
  - Faster Sorting
  - Binary Search

Arrays and pointers

- Primitive arrays implemented in C using pointer to block of contiguous memory
- Consider array of 8 ints:

  ```c
  int arr[8];
  ```

- Accessing `arr` using array entry operator:

  ```c
  int a = arr[0];
  ```

- `arr` is like a pointer to element 0 of the array:

  `int *pa = arr;` $\Leftrightarrow$ `int *pa = &arr[0];`

- Not modifiable/reassignable like a pointer

The `sizeof()` operator

- For primitive types/variables, size of type in bytes:

  ```
  int s = sizeof(char); /* == 1 */
  double f; /* sizeof(f) == 8 */
  ```
  (64-bit OS)

- For primitive arrays, size of array in bytes:

  ```
  int arr[8]; /* sizeof(arr) == 32 */
  long arr[5]; /* sizeof(arr) == 40 */
  ```
  (64-bit OS)

- Array length:

  ```
  /* needs to be on one line when implemented */
  #define array_length(arr) (sizeof(arr) == 0 ?
    0 : sizeof(arr)/sizeof((arr)[0]))
  ```

- More about `sizeof()` next week...

Pointer arithmetic

- Suppose **int** $*$pa = arr;
- Pointer not an `int`, but can add or subtract an `int` from a pointer:
  pa + i points to `arr[i]`
- Address value increments by i times size of data type
  Suppose `arr[0]` has address 100. Then `arr[3]` has address 112.
- Suppose **char** $*$ pc = (**char** $*$)pa; What value of i satisfies (**int** $*$)(pc+i) == pa + 3?

Pointer arithmetic

- Suppose **int** $*$pa = arr;
- Pointer not an `int`, but can add or subtract an `int` from a pointer:
  pa + i points to `arr[i]`
- Address value increments by i times size of data type
  Suppose `arr[0]` has address 100. Then `arr[3]` has address 112.
- Suppose **char** $*$ pc = (**char** $*$)pa; What value of i satisfies (**int** $*$)(pc+i) == pa + 3?
  - $i = 12$

6.087 Lecture 5 – January 15, 2010

- Review
- Pointers and Memory Addresses
  - Physical and Virtual Memory
  - Addressing and Indirection
  - Functions with Multiple Outputs
- Arrays and Pointer Arithmetic
- Strings
  - String Utility Functions
- Searching and Sorting Algorithms
  - Linear Search
  - A Simple Sort
  - Faster Sorting
  - Binary Search

Strings as arrays

- Strings stored as null-terminated character arrays (last character == `'\0'`)
- Suppose `char str[] = "This is a string.";` and
  `char * pc = str;`
- Manipulate string as you would an array

```
*(pc+10) = 'S';
puts(str); /* prints "This is a String." */
```

String utility functions

- String functions in standard header `string.h`
- Copy functions: `strcpy()`, `strncpy()`
  char ∗ strcpy(strto,strfrom); – copy *strfrom* to *strto*
  char ∗ strncpy(strto,strfrom,n); – copy n chars from *strfrom* to *strto*
- Comparison functions: `strcmp()`, `strncmp()`
  int strcmp(str1,str2); – compare *str1*, *str2*; return 0 if equal, positive if *str1*>*str2*, negative if *str1*<*str2*
  int strncmp(str1,str2,n); – compare first n chars of *str1* and *str2*
- String length: `strlen()`
  int strlen(str); – get length of *str*

More string utility functions

- Concatenation functions: `strcat()`, `strncat()`

  char * strcat(strto,strfrom); – add *strfrom* to end of *strto*

  char * strncat(strto,strfrom,n); – add *n* chars from *strfrom* to end of *strto*

- Search functions: `strchr()`, `strrchr()`

  char * strchr(str,c); – find char *c* in *str*, return pointer to first occurrence, or NULL if not found

  char * strrchr(str,c); – find char *c* in *str*, return pointer to last occurrence, or NULL if not found

- Many other utility functions exist…

6.087 Lecture 5 – January 15, 2010

- Review
- Pointers and Memory Addresses
  - Physical and Virtual Memory
  - Addressing and Indirection
  - Functions with Multiple Outputs
- Arrays and Pointer Arithmetic
- Strings
  - String Utility Functions
- Searching and Sorting Algorithms
  - Linear Search
  - A Simple Sort
  - Faster Sorting
  - Binary Search

Searching and sorting

- Basic algorithms
- Can make good use of pointers
- Just a few examples; not a course in algorithms
- Big-O notation

Searching an array

- Suppose we have an array of `int`'s

```
int arr[100]; /* array to search */
```

- Let's write a simple search function:

```
int * linear_search(int val) {
  int * parr, * parrend = arr + array_length(arr);
  for (parr = arr; parr < parrend; parr++) {
    if (*parr == val)
      return parr;
  }
  return NULL;
}
```

A simple sort

- A simple insertion sort: $O(n^2)$
  - iterate through array until an out-of-order element found
  - insert out-of-order element into correct location
  - repeat until end of array reached
- Split into two functions for ease-of-use

```
int arr[100]; /* array to sort */

void shift_element(unsigned int i) {
  /* do insertion of out-of-order element */
}

void insertion_sort() {
  /* main insertion sort loop */
  /* call shift_element() for
     each out-of-order element */
}
```

Shifting out-of-order elements

- Code for shifting the element

```
/* move previous elements down until
   insertion point reached */
void shift_element(unsigned int i) {
  int ivalue;
  /* guard against going outside array */
  for (ivalue = arr[i]; i && arr[i-1] > ivalue; i--)
    arr[i] = arr[i-1]; /* move element down */
  arr[i] = ivalue; /* insert element */
}
```

Insertion sort

- Main insertion sort loop

```
/* iterate until out-of-order element found;
   shift the element, and continue iterating */
void insertion_sort(void) {
  unsigned int i, len = array_length(arr);
  for (i = 1; i < len; i++)
    if (arr[i] < arr[i-1])
      shift_element(i);
}
```

- Can you rewrite using pointer arithmetic instead of indexing?

Quicksort

- Many faster sorts available (shellsort, mergesort, quicksort, ...)
- Quicksort: $O(n \log n)$ average; $O(n^2)$ worst case
  - choose a pivot element
  - move all elements less than pivot to one side, all elements greater than pivot to other
  - sort sides individually (recursive algorithm)
- Implemented in C standard library as `qsort()` in `stdlib.h`

Quicksort implementation

- Select the pivot; separate the sides:

```
void quick_sort(unsigned int left,
                unsigned int right) {
  unsigned int i, mid;
  int pivot;
  if (left >= right)
    return; /* nothing to sort */
  /* pivot is midpoint; move to left side */
  swap(arr+left, arr + (left+right)/2);
  pivot = arr[mid = left];
  /* separate into side < pivot (left+1 to mid)
     and side >= pivot (mid+1 to right) */
  for (i = left+1; i <= right; i++)
    if (arr[i] < pivot)
      swap(arr + ++mid, arr + i);
```

[Kernighan and Ritchie. The C Programming Language. 2nd ed. Prentice Hall, 1988.]

Quicksort implementation

- Restore the pivot; sort the sides separately:

```
/* restore pivot position */
swap(arr+left, arr+mid);
/* sort two sides */
if (mid > left)
  quick_sort(left, mid-1);
if (mid < right)
  quick_sort(mid+1,right);
}
```

- Starting the recursion:

```
quick_sort(0, array_length(arr) - 1);
```

[Kernighan and Ritchie. The C Programming Language. 2nd ed. Prentice Hall, 1988.]

Discussion of quicksort

- Not *stable* (equal-valued elements can get switched) in present form
- Can sort *in-place* – especially desirable for low-memory environments
- Choice of pivot influences performance; can use random pivot
- Divide and conquer algorithm; easily parallelizeable
- Recursive; in worst case, can cause stack overflow on large array

Searching a sorted array

- Searching an arbitrary list requires visiting half the elements on average
- Suppose list is sorted; can make use of sorting information:
  - if desired value greater than value and current index, only need to search after index
  - each comparison can split list into two pieces
  - solution: compare against middle of current piece; then new piece guaranteed to be half the size
  - divide and conquer!
- More searching next week…

Binary search

- Binary search: $O(\log n)$ average, worst case:

```c
int * binary_search(int val) {
  unsigned int L = 0, R = array_length(arr), M;
  while (L < R) {
    M = (L+R-1)/2;
    if (val == arr[M])
      return arr+M; /* found */
    else if (val < arr[M])
      R = M; /* in first half */
    else
      L = M+1; /* in second half */
  }
  return NULL; /* not found */
}
```

Binary search

- Worst case: logarithmic time
- Requires random access to array memory
  - on sequential data, like hard drive, can be slow
  - seeking back and forth in sequential memory is wasteful
  - better off doing linear search in some cases
- Implemented in C standard library as `bsearch()` in `stdlib.h`

Summary

Topics covered:

- Pointers: addresses to memory
  - physical and virtual memory
  - arrays and strings
  - pointer arithmetic
- Algorithms
  - searching: linear, binary
  - sorting: insertion, quick

6.087 Practical Programming in C
January (IAP) 2010

6.087 Lecture 6 – January 19, 2010

- Review
- User defined datatype
  - Structures
  - Unions
  - Bitfields
- Data structure
  - Memory allocation
  - Linked lists
  - Binary trees

Review: pointers

- Pointers: memory address of variables
- '&' (address of) operator.
- Declaring: **int** x=10; **int** * px= &x;
- Dereferencing: *px=20;
- Pointer arithmetic:
  - `sizeof()`
  - incrementing/decrementing
  - absolute value after operation depends on pointer datatype.

Review: string.h

- String copy: `strcpy()`,`strncpy()`
- Comparison: `strcmp()`,`strncmp()`
- Length: `strlen()`
- Concatenation: `strcat()`
- Search: `strchr()`,`strstr()`

Searching and sorting

Searching

- Linear search: $O(n)$
- Binary search: $O(logn)$. The array has to be sorted first.

Sorting

- Insertion sort: $O(n^2)$
- Quick sort: $O(n \log n)$

6.087 Lecture 6 – January 19, 2010

- Review
- User defined datatype
  - Structures
  - Unions
  - Bitfields
- Data structure
  - Memory allocation
  - Linked lists
  - Binary trees

Structure

Definition: A structure is a collection of related variables (of possibly different types) grouped together under a single name. This is a an example of **composition**–building complex structures out of simple ones.

Examples:

```c
struct point
{
 int x;
 int y;
};
/*notice the ; at the end*/
```

```c
struct employee
{
  char fname[100];
  char lname[100];
  int  age;
};
/*members of different
  type*/
```

Structure

- **struct** defines a new datatype.
- The name of the structure is optional.
  struct {...} x,y,z;
- The variables declared within a structure are called its *members*
- Variables can be declared like any other built in data-type.
  struct point ptA;
- Initialization is done by specifying values of every member.
  struct point ptA={10,20};
- Assignment operator copies every member of the structure (be careful with pointers).

Structure (cont.)

More examples:

```
struct triangle
{
  struct point ptA;
  struct point ptB;
  struct point ptC;
};
/*members can be structures*/
```

```
struct chain_element
{
 int data;
 struct chain_element* next
};
/*members can be
self referential*/
```

Structure (cont.)

- Individual members can be accessed using '.' operator.
  `struct point pt={10,20}; int x=pt.x; int y=pt.y;`
- If structure is nested, multiple '.' are required

```
struct rectangle
{
  struct point tl;/*top left*/
  struct point br;/*bot right*/
};
struct rectangle rect;
int tlx=rect.tl.x; /*nested*/
int tly=rect.tl.y;
```

Structure pointers

- Structures are copied element wise.
- For large structures it is more efficient to pass pointers.
  void foo(**struct** point * pp); **struct** point pt; foo(&pt)
- Members can be accesses from structure pointers using '->' operator.

```
struct point p={10,20};
struct point* pp=&p;
pp->x = 10; /*changes p.x*/
int y= pp->y; /*same as y=p.y*/
```

Other ways to access structure members?

```
struct point p={10,20};
struct point* pp=&p;
(*pp).x = 10; /*changes p.x*/
int y= (*pp).y; /*same as y=p.y*/
```

why is the `()` required?

Arrays of structures

- Declaring arrays of int: `int x[10];`
- Declaring arrays of structure: `struct point p[10];`
- Initializing arrays of int: `int x[4]={0,20,10,2};`
- Initializing arrays of structure:

```
struct point p[3]={0,1,10,20,30,12};
struct point p [3]={{0,1},{10,20},{30,12}};
```

Size of structures

- The size of a structure is greater than or equal to the sum of the sizes of its members.
- Alignment

```
struct {
char c;
/*padding*/
int i;
```

- Why is this an important issue? libraries, precompiled files, SIMD instructions.
- Members can be explicitly aligned using **compiler extensions.**

```
__attribute__((aligned(x))) /*gcc*/
__declspec((aligned(x))) /*MSVC*/
```

Union

A union is a variable that may hold objects of different types/sizes in the same memory location. Example:

```
union data
{
   int idata;
   float fdata;
   char* sdata;
} d1,d2,d3;
d1.idata=10;
d1.fdata=3.14F;
d1.sdata="hello world";
```

Unions (cont.)

- The size of the union variable is equal to the size of its largest element.
- **Important:** The compiler does not test if the data is being read in the correct format.

  ```
  union data d; d.idata=10; float f=d.fdata; /* will give junk*/
  ```
- A common solution is to maintain a separate variable.

```
enum dtype{INT,FLOAT,CHAR};
struct variant
{
  union data d;
  enum dtype t;
};
```

Bit fields

Definition: A bit-field is a set of adjacent bits within a single 'word'. Example:

```
struct flag{
unsigned int is_color:1;
unsigned int has_sound:1;
unsigned int is_ntsc:1;
};
```

- the number after the colons specifies the width in bits.
- each variables should be declared as **unsigned int**

Bit fields vs. masks

CLR=0x1,SND=0x2,NTSC=0x4;	**struct** flag f;
x|= CLR; x|=SND; x|=NTSC	f.has_sound=1;f.is_color=1;
x&= ~CLR; x&=~SND;	f.has_sound=0;f.is_color=0;
if (x & CLR || x& NTSC)	**if** (f.is_color || f.has_sound)

6.087 Lecture 6 – January 19, 2010

- Review
- User defined datatype
  - Structures
  - Unions
  - Bitfields
- Data structure
  - Memory allocation
  - Linked lists
  - Binary trees

Digression: dynamic memory allocation

void∗ malloc(size_t n)

- `malloc()` allocates blocks of memory
- returns a pointer to **unitialized** block of memory on success
- returns NULL on failure.
- the returned value should be cast to appropriate type using (). **int**∗ ip=(**int**∗)malloc(**sizeof**(**int**)∗100)

void∗ calloc(size_t n,size_t size)

- allocates an array of n elements each of which is 'size' bytes.
- initializes memory to 0

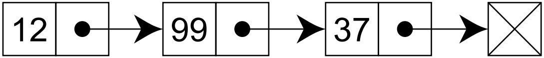
void free(**void**∗)

- Frees memory allocated my malloc()
- Common error: accessing memory after calling free

Linked list

Definition: A dynamic data structure that consists of a sequence of records where each element contains a **link** to the next record in the sequence.

- Linked lists can be *singly linked*, *doubly linked* or *circular*. For now, we will focus on *singly* linked list.
- Every node has a *payload* and a link to the next node in the list.
- The start (*head*) of the list is maintained in a separate variable.
- End of the list is indicated by NULL (*sentinel*).



Linked list

```c
struct node
{
  int data;/*payload*/
  struct node* next;
};
struct node* head;/*beginning*/
```

Linked list vs. arrays

	linked-list	array
size	dynamic	fixed
indexing	O(n)	O(1)
inserting	O(1)	O(n)
deleting	O(1)	O(n)

Linked list

Creating new element:

```
struct node* nalloc(int data)
{
  struct node* p=(struct node*)malloc(sizeof(node));
  if(p!=NULL)
  {
   p->data=data;
   p->next=NULL;
  }
  return p;
}
```

Linked list

Adding elements to front:

```
struct node* addfront(struct node* head, int data)
{
  struct node* p=nalloc(data);
  if(p==NULL) return head;
  p->next=head;
  return p;
```

Linked list

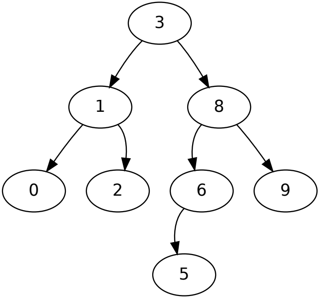
Iterating:

```
for(p=head;p!=NULL;p=p->next)
  /*do something*/

for(p=head;p->next!=NULL;p=p->next)
  /*do something*/
```

Binary trees

- A binary tree is a dynamic data structure where each node has at most two children. A binary **search** tree is a binary tree with ordering among its children.
- Usually, all elements in the left subtree are assumed to be ”less” than the root element and all elements in the right subtree are assumed to be "greater" than the root element.



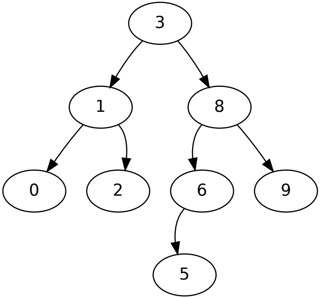
Binary tree (cont.)

```c
struct tnode
{
    int data; /*payload*/
    struct tnode* left;
    struct tnode* right;
};
```

The operation on trees can be framed as recursive operations.

Traversal (printing, searching):

- pre-order: root, left subtree, right subtree
- Inorder: left subtree, root, right subtree
- post-order: right subtree, right subtree, root



Binary tree (cont.)

Add node:

```c
struct tnode* addnode(struct tnode* root, int data)
{
  struct tnode* p=NULL;
  /*termination condition*/
  if(root==NULL)
  {
    /*allocate node*/
    /*return new root*/
  }
  /*recursive call*/
  else if(data< root->data)
    root->left=addnode(root->left,data)
  else
    root->right=addnode(root->right,data)
}
```

6.087 Practical Programming in C
January (IAP) 2010

6.087 Lecture 7 – January 20, 2010

- Review

- More about Pointers
  - Pointers to Pointers
  - Pointer Arrays
  - Multidimensional Arrays

- Data Structures
  - Stacks
  - Queues
  - Application: Calculator

Review: Compound data types

- **struct** - structure containing one or multiple fields, each with its own type (or compound type)
  - size is combined size of all the fields, padded for byte alignment
  - anonymous or named
- **union** - structure containing one of several fields, each with its own type (or compound type)
  - size is size of largest field
  - anonymous or named
- Bit fields - structure fields with width in bits
  - aligned and ordered in architecture-dependent manner
  - can result in inefficient code

Review: Compound data types

- Consider this compound data structure:

```
struct foo {
  short s;
  union {
    int i;
    char c;
  } u;
  unsigned int flag_s : 1;
  unsigned int flag_u : 2;
  unsigned int bar;
};
```

- Assuming a 32-bit x86 processor, evaluate
  `sizeof(struct foo)`

Review: Compound data types

- Consider this compound data structure:

```
struct foo {
  short s;                          ← 2 bytes
  union {                           ← 4 bytes,
    int i;                          4 byte-aligned
    char c;
  } u;
  unsigned int flag_s : 1;          ← bit fields
  unsigned int flag_u : 2;
  unsigned int bar;                 ← 4 bytes,
};                                  4 byte-aligned
```

- Assuming a 32-bit x86 processor, evaluate
  sizeof(**struct** foo)

Review: Compound data types

- How can we rearrange the fields to minimize the size of **struct** foo?

Review: Compound data types

- How can we rearrange the fields to minimize the size of **struct** foo?
- Answer: order from largest to smallest:

```
struct foo {
  union {
    int i;
    char c;
  } u;
  unsigned int bar;
  short s;
  unsigned int flag_s : 1;
  unsigned int flag_u : 2;
};
```

sizeof(**struct** foo) $= 12$

Review: Linked lists and trees

- Linked list and tree dynamically grow as data is added/removed
- Node in list or tree usually implemented as a **struct**
- Use `malloc()`, `free()`, etc. to allocate/free memory dynamically
- Unlike arrays, do not provide fast random access by index (need to iterate)

- Review
- More about Pointers
  - Pointers to Pointers
  - Pointer Arrays
  - Multidimensional Arrays
- Data Structures
  - Stacks
  - Queues
  - Application: Calculator

Pointer review

- Pointer represents address to variable in memory
- Examples:

  int ∗pn; – pointer to `int`

  struct div_t ∗ pdiv; – pointer to structure `div_t`
- Addressing and indirection:

```
double pi = 3.14159;
double *ppi = &pi;
printf("pi = %g\n", *ppi);
```

- Today: pointers to pointers, arrays of pointers, multidimensional arrays

Pointers to pointers

- Address stored by pointer also data in memory
- Can address location of address in memory – pointer to that pointer

```c
int n = 3;
int *pn = &n; /* pointer to n */
int **ppn = &pn; /* pointer to address of n */
```

- Many uses in C: pointer arrays, string arrays

Pointer pointers example

- What does this function do?

```
void swap(int **a, int **b) {
  int *temp = *a;
  *a = *b;
  *b = temp;
}
```

Pointer pointers example

- What does this function do?

```c
void swap(int **a, int **b) {
  int *temp = *a;
  *a = *b;
  *b = temp;
}
```

- How does it compare to the familiar version of swap?

```c
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

Pointer arrays

- Pointer array – array of pointers
  `int *arr[20];` – an array of pointers to `int`'s
  `char *arr[10];` – an array of pointers to `char`'s
- Pointers in array can point to arrays themselves
  `char *strs[10];` – an array of `char` arrays (or strings)

Pointer array example

- Have an array `int arr[100];` that contains some numbers
- Want to have a sorted version of the array, but not modify `arr`
- Can declare a pointer array `int * sorted_array[100];` containing pointers to elements of `arr` and sort the pointers instead of the numbers themselves
- Good approach for sorting arrays whose elements are very large (like strings)

Pointer array example

Insertion sort:

```
/* move previous elements down until
   insertion point reached */
void shift_element(unsigned int i) {
  int *pvalue;
  /* guard against going outside array */
  for (pvalue = sorted_array[i]; i &&
         *sorted_array[i-1] > *pvalue; i--) {
    /* move pointer down */
    sorted_array[i] = sorted_array[i-1];
  }
  sorted_array[i] = pvalue; /* insert pointer */
}
```

Pointer array example

Insertion sort (continued):

```
/* iterate until out-of-order element found;
   shift the element, and continue iterating */
void insertion_sort(void) {
  unsigned int i, len = array_length(arr);
  for (i = 1; i < len; i++)
    if (*sorted_array[i] < *sorted_array[i-1])
      shift_element(i);
}
```

String arrays

- An array of strings, each stored as a pointer to an array of chars
- Each string may be of different length

```
char str1[] = "hello"; /* length = 6 */
char str2[] = "goodbye"; /* length = 8 */
char str3[] = "ciao"; /* length = 5 */
char * strArray[] = {str1, str2, str3};
```

- Note that strArray contains only pointers, not the characters themselves!

Multidimensional arrays

- C also permits multidimensional arrays specified using `[ ]` brackets notation:

  `int world[20][30];` is a 20x30 2-D array of `int`'s
- Higher dimensions possible:

  `char bigcharmatrix [15][7][35][4];` – what are the dimensions of this?
- Multidimensional arrays are rectangular; pointer arrays can be arbitrary shaped

- Review
- More about Pointers
  - Pointers to Pointers
  - Pointer Arrays
  - Multidimensional Arrays
- Data Structures
  - Stacks
  - Queues
  - Application: Calculator

More data structures

- Last time: linked lists
- Today: stack, queue
- Can be implemented using linked list or array storage

The stack

- Special type of list - last element in (push) is first out (pop)
- Read and write from same end of list
- The stack (where local variables are stored) is implemented as a *gasp* stack

Stack as array

- Store as array buffer (static allocation or dynamic allocation):

```c
int stack_buffer[100];
```

- Elements added and removed from end of array; need to track end:

```c
int itop = 0; /* end at zero => initialized for empty stack */
```

Stack as array

- Add element using `void push(int);`

```
void push(int elem) {
  stack_buffer[itop++] = elem;
}
```

- Remove element using `int pop(void);`

```
int pop(void) {
  if (itop > 0)
    return stack_buffer[--itop];
  else
    return 0; /* or other special value */
}
```

- Some implementations provide `int top(void);` to read last (top) element without removing it

Stack as linked list

- Store as linked list (dynamic allocation):

```
struct s_listnode {
  int element;
  struct s_listnode * pnext;
};
```

  `struct s_listnode * stack_buffer = NULL;` – start empty

- “Top” is now at front of linked list (no need to track)

Stack as linked list

- Add element using `void push(int);`

```
void push(int elem) {
  struct s_listnode *new_node = /* allocate new node */
    (struct s_listnode *)malloc(sizeof(struct s_listnode));
  new_node->pnext = stack_buffer;
  new_node->element = elem;
  stack_buffer = new_node;
}
```

- Adding an element pushes back the rest of the stack

Stack as linked list

- Remove element using `int pop(void);`

```
int pop(void) {
  if (stack_buffer) {
    struct s_listnode *pelem = stack_buffer;
    int elem = stack_buffer->element;
    stack_buffer = pelem->pnext;
    free(pelem); /* remove node from memory */
    return elem;
  } else
    return 0; /* or other special value */
}
```

- Some implementations provide `int top(void);` to read last (top) element without removing it

The queue

- Opposite of stack - first in (enqueue), first out (dequeue)
- Read and write from opposite ends of list
- Important for UIs (event/message queues), networking (Tx, Rx packet queues)
- Imposes an ordering on elements

Queue as array

- Again, store as array buffer (static or dynamic allocation);

```
float queue_buffer[100];
```

- Elements added to end (rear), removed from beginning (front)
- Need to keep track of front and rear:

```
int ifront = 0, irear = 0;
```

- Alternatively, we can track the front and number of elements:

```
int ifront = 0, icount = 0;
```

- We'll use the second way (reason apparent later)

Queue as array

- Add element using **void** enqueue(**float**);

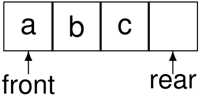
```
void enqueue(float elem) {
  if (icount < 100) {
    queue_buffer[ifront+icount] = elem;
    icount++;
  }
}
```

- Remove element using **float** dequeue(**void**);

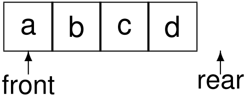
```
float dequeue(void) {
  if (icount > 0) {
    icount--;
    return queue_buffer[ifront++];
  } else
    return 0.; /* or other special value */
}
```

Queue as array

- This would make for a very poor queue! Observe a queue of capacity 4:



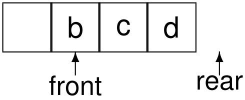
- Enqueue `'d'` to the rear of the queue:



The queue is now full.

Queue as array

- Dequeue `'a'`:



- Enqueue `'e'` to the rear: where should it go?
- Solution: use a circular (or “ring”) buffer
  - `'e'` would go in the beginning of the array

Queue as array

- Need to modify **void** enqueue(**float**); and **float** dequeue(**void**);
- New **void** enqueue(**float**);:

```
void enqueue(float elem) {
  if (icount < 100) {
    queue_buffer[(ifront+icount) % 100] = elem;
    icount++;
  }
}
```

Queue as array

- New float dequeue(void);:

```
float dequeue(void) {
  if (icount > 0) {
    float elem = queue_buffer[ifront];
    icount--;
    ifront++;
    if (ifront == 100)
      ifront = 0;
    return elem;
  } else
    return 0.; /* or other special value */
}
```

- Why would using “front” and “rear” counters instead make this harder?

Queue as linked list

- Store as linked list (dynamic allocation):

```
struct s_listnode {
  float element;
  struct s_listnode * pnext;
};
```

  `struct s_listnode *queue_buffer = NULL;` – start empty

- Let front be at beginning – no need to track front
- Rear is at end – we should track it:

  `struct s_listnode *prear = NULL;`

Queue as linked list

- Add element using `void enqueue(float);`

```
void enqueue(float elem) {
  struct s_listnode *new_node = /* allocate new node */
    (struct s_listnode *)malloc(sizeof(struct s_listnode));
  new_node->element = elem;
  new_node->pnext = NULL; /* at rear */
  if (prear)
    prear->pnext = new_node;
  else /* empty */
    queue_buffer = new_node;
  prear = new_node;
}
```

- Adding an element doesn't affect the front if the queue is not empty

Queue as linked list

- Remove element using **float** dequeue(**void**);

```c
float dequeue(void) {
  if (queue_buffer) {
    struct s_listnode *pelem = queue_buffer;
    float elem = queue_buffer->element;
    queue_buffer = pelem->pnext;
    if (pelem == prear) /* at end */
      prear = NULL;
    free(pelem); /* remove node from memory */
    return elem;
  } else
    return 0.; /* or other special value */
}
```

- Removing element doesn't affect rear unless resulting queue is empty

A simple calculator

- Stacks and queues allow us to design a simple expression evaluator
- Prefix, infix, postfix notation: operator before, between, and after operands, respectively

Infix	Prefix	Postfix
A + B	+ A B	A B +
A * B - C	- * A B C	A B * C -
(A + B) * (C - D)	* + A B - C D	A B + C D - *

- Infix more natural to write, postfix easier to evaluate

Infix to postfix

- "Shunting yard algorithm" - Dijkstra (1961): input and output in queues, separate stack for holding operators
- Simplest version (operands and binary operators only):
  1. dequeue token from input
  2. if operand (number), add to output queue
  3. if operator, then pop operators off stack and add to output queue as long as
     - top operator on stack has higher precedence, or
     - top operator on stack has same precedence and is left-associative

     and push new operator onto stack
  4. return to step 1 as long as tokens remain in input
  5. pop remaining operators from stack and add to output queue

Infix to postfix example

- Infix expression: A + B * C - D

Token	Output queue	Operator stack
A	A	
+	A	+
B	A B	+
*	A B	+ *
C	A B C	+ *
-	A B C * +	-
D	A B C * + D	-
(end)	A B C * + D -	

- Postfix expression: A B C * + D -
- What if expression includes parentheses?

Example with parentheses

- Infix expression: (A + B) * (C - D)

Token	Output queue	Operator stack
(		(
A	A	(
+	A	(+
B	A B	(+
)	A B +	
*	A B +	*
(	A B +	* (
C	A B + C	* (
-	A B + C	* (-
D	A B + C D	* (-
)	A B + C D -	*
(end)	A B + C D - *	

- Postfix expression: A B + C D - *

Evaluating postfix

- Postfix evaluation very easy with a stack:
  1. dequeue a token from the postfix queue
  2. if token is an operand, push onto stack
  3. if token is an operator, pop operands off stack (2 for binary operator); push result onto stack
  4. repeat until queue is empty
  5. item remaining in stack is final result

Postfix evaluation example

- Postfix expression: 3 4 + 5 1 - *

Token	Stack
3	3
4	3 4
+	7
5	7 5
1	7 5 1
-	7 4
*	28
(end)	answer $= 28$

- Extends to expressions with functions, unary operators
- Performs evaluation in one pass, unlike with prefix notation

Summary

Topics covered:

- Pointers to pointers
  - pointer and string arrays
  - multidimensional arrays
- Data structures
  - stack and queue
  - implemented as arrays and linked lists
  - writing a calculator

6.087 Practical Programming in C
January (IAP) 2010

6.087 Lecture 8 – January 21, 2010

- Review
- Pointers
  - Void pointers
  - Function pointers
- Hash table

Review:Pointers

- pointers: **int** x; **int*** p=&x;
- pointers to pointer: **int** x; **int*** p=&x;**int**** pp=&p;
- Array of pointers: **char*** names[]={"abba","u2"};
- Multidimensional arrays: **int** x[20][20];

Review: Stacks

- LIFO: last in first out data structure.
- items are inserted and removed from the same end.
- operations: `push()`,`pop()`,`top()`
- can be implemented using arrays, linked list

Review: Queues

- FIFO: first in first out
- items are inserted at the rear and removed from the front.
- operations: `queue()`,`dequeue()`
- can be implemented using arrays, linked list

Review: Expressions

- Infix: `(A+B)*(C-D)`
- prefix: *+AB-CD
- postfix: AB+CD-*

6.087 Lecture 8 – January 21, 2010

- Review
- Pointers
  - Void pointers
  - Function pointers
- Hash table

Void pointers

- C does not allow us to declare and use void **variables.**
- void can be used only as return type or parameter of a function.
- C allows void **pointers**
- Question: What are some scenarios where you want to pass void pointers?
- void pointers can be used to point to any data type
  - `int x; void* p=&x; /*points to int */`
  - `float f;void* p=&f; /*points to float */`
- void pointers cannot be dereferenced. The pointers should always be cast before dereferencing.

```
void* p; printf("%d",*p); /* invalid */
void* p; int *px=(int*)p; printf("%d",*px); /*valid */
```

Function pointers

- In some programming languages, functions are first class variables (can be passed to functions, returned from functions etc.).
- In C, function itself is not a variable. But it is possible to declare pointer to functions.
- Question: What are some scenarios where you want to pass pointers to functions?
- Declaration examples:
  - `int (*fp)(int) /*notice the () */`
  - `int (*fp)(void*,void*)`
- Function pointers can be assigned, pass to and from functions, placed in arrays etc.

Callbacks

Definition: Callback is a piece of executable code passed to functions. In C, callbacks are implemented by passing function pointers.
Example:

```c
void qsort(void* arr, int num,int size, int (*fp)(void* pa,void*pb))
```

- `qsort()` function from the standard library can be sort an array of any datatype.
- Question: How does it do that? callbacks.
- `qsort()` calls a function whenever a comparison needs to be done.
- The function takes two arguments and returns (<0,0,>0) depending on the relative order of the two items.

Callback (cont.)

```
int arr[]={10,9,8,1,2,3,5};
/*callback*/
int asc(void* pa,void* pb)
{
  return (* (int*)pa - *(int*)pb);
}
/*callback*/
int desc(void* pa,void* pb)
{
  return (* (int*)pb - *(int*)pa);
}
/*sort in ascending order*/
qsort(arr,sizeof(arr)/sizeof(int),sizeof(int),asc);
/*sort in descending order*/
qsort(arr,sizeof(arr)/sizeof(int),sizeof(int),desc);
```

Callback (cont.)

Consider a linked list with nodes defined as follows:

```
struct node{
  int data;
  struct node* next;
};
```

Also consider the function 'apply' defined as follows:

```
void apply(struct node* phead,
        void (*fp)(void*,void* ),
        void* arg) /*only fp has to be named*/
{
    struct node* p=phead;
    while(p!=NULL)
    {
      fp(p,arg); /*can also use (*fp)(p,arg)*/
      p=p->next;
    }
}
```

Callback (cont.)

Iterating:

```
struct node* phead;
/*populate somewhere*/
void print(void* p,void* arg)
{
    struct node* np=(struct node*)p;
    printf("%d ",np->data);
}
apply(phead,print,NULL);
```

Callback (cont.)

Counting nodes:

```
void dototal(void* p, void* arg)
{
  struct node* np=(struct node*)p;
  int* ptotal         =(int*)arg;
  *ptotal += np->data;
}
int total=0;
apply(phead,dototal,&total);
```

Array of function pointers

Example:Consider the case where different functions are called based on a value.

```
enum TYPE{SQUARE,RECT,CIRCILE,POLYGON};
struct shape{
  float params[MAX];
  enum TYPE type;
};
void draw(struct shape* ps)
{
  switch(ps->type)
  {
    case SQUARE:
      draw_square(ps);break;
    case RECT:
      draw_rect(ps);break;
    ...
  }
}
```

Array of function pointers

The same can be done using an array of function pointers instead.

```
void (*fp[4])(struct shape* ps)=
{&draw_square,&draw_rec,&draw_circle,&draw_poly};
typedef void (*fp)(struct shape* ps) drawfn;
drawfn fp[4]=
{&draw_square,&draw_rec,&draw_circle,&draw_poly};
void draw(struct shape* ps)
{
  (*fp[ps->type])(ps); /* call the correct function*/
}
```

6.087 Lecture 8 – January 21, 2010

- Review
- Pointers
  - Void pointers
  - Function pointers
- Hash table

Hash table

Hash tables (hashmaps) combine linked list and arrays to provide an *efficient* data structure for storing dynamic data. Hash tables are commonly implemented as an array of linked lists (hash tables with chaining).

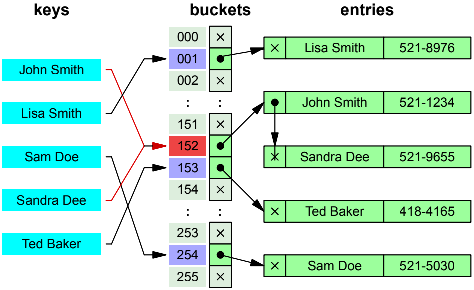


Figure: Example of a hash table with chaining (source: wikipedia)

Hash table

- Each data item is associated with a *key* that determines its location.
- *Hash functions* are used to generate an evenly distributed hash value.
- A *hash collision* is said to occur when two items have the same hash value.
- Items with the same hash keys are chained
- Retrieving an item is $O(1)$ operation.

Hash tables

Hash functions:

- A hash function maps its input into a finite range: hash value, hash code.
- The hash value should ideally have uniform distribution. why?
- Other uses of hash functions: cryptography, caches (computers/internet), bloom filters etc.
- Hash function types:
  - Division type
  - Multiplication type
- Other ways to avoid collision: linear probing, double hashing.

Hash table: example

```
#define MAX_BUCKETS 1000
#define MULTIPLIER 31
struct wordrec
{
  char* word;
  unsigned long count;
  struct wordrec* next;
};

/*hash bucket*/
struct wordrec* table[MAX_LEN];
```

Hash table: example

```
unsigned long hashstring(const char* str)
{
  unsigned long hash=0;
  while(*str)
    {
      hash= hash*MULTIPLIER+*str;
      str++;
    }
  return hash%MAX_BUCKETS;
}
```

Hash table: example

```
struct wordrec* lookup(const char* str, int create)
{
  struct wordrec* curr=NULL;
  unsigned long hash=hashstring(str);
  struct wordrec* wp=table[hash];
  for(curr=wp;curr!=NULL ;curr=curr->next)
    /*search*/;
notfound:
  if(create)
      /*add to front*/
  return curr;
}
```

6.087 Practical Programming in C
January (IAP) 2010

6.087 Lecture 9 – January 22, 2010

- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Review: Void pointers

- Void pointer – points to any data type:

```c
int x; void * px = &x; /* implicit cast to (void *) */
float f; void * pf = &f;
```

- Cannot be dereferenced directly; void pointers must be cast prior to dereferencing:

```c
printf("%d %f\n", *(int *)px, *(float *)pf);
```

Review: Function pointers

- Functions not variables, but also reside in memory (i.e. have an address) – we can take a pointer to a function
- Function pointer declaration:
  ```
  int (*cmp)(void *, void *);
  ```
- Can be treated like any other pointer
- No need to use & operator (but you can)
- Similarly, no need to use * operator (but you can)

Review: Function pointers

```
int strcmp_wrapper(void * pa, void * pb) {
  return strcmp((const char *)pa, (const char *)pb);
}
```

- Can assign to a function pointer:

  `int (*fp)(void *, void *) = strcmp_wrapper;` or

  `int (*fp)(void *, void *) = &strcmp_wrapper;`

- Can call from function pointer: (`str1` and `str2` are strings)

  `int ret = fp(str1, str2);` or

  `int ret = (*fp)(str1, str2);`

Review: Hash tables

- Hash table (or hash map): array of linked lists for storing and accessing data efficiently
- Each element associated with a key (can be an integer, string, or other type)
- Hash function computes hash value from key (and table size); hash value represents index into array
- Multiple elements can have same hash value – results in collision; elements are chained in linked list

6.087 Lecture 9 – January 22, 2010

- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Symbols and libraries

- External libraries provide a wealth of functionality – example: C standard library
- Programs access libraries' functions and variables via identifiers known as *symbols*
- Header file declarations/prototypes mapped to symbols at compile time
- Symbols linked to definitions in external libraries during *linking*
- Our own program produces symbols, too

Functions and variables as symbols

- Consider the simple hello world program written below:

```c
#include <stdio.h>

const char msg[] = "Hello, world.";

int main(void) {
  puts(msg);
  return 0;
}
```

- What variables and functions are declared globally?

Functions and variables as symbols

- Consider the simple hello world program written below:

```
#include <stdio.h>

const char msg[] = "Hello, world.";

int main(void) {
  puts(msg);
  return 0;
}
```

- What variables and functions are declared globally?

  `msg`, `main()`, `puts()`, others in `stdio.h`

Functions and variables as symbols

- Let's compile, but not link, the file hello.c to create hello.o:
  `athena%`[1] `gcc -Wall -c hello.c -o hello.o`
  - `-c`: compile, but do not link hello.c; result will compile the code into machine instructions but not make the program executable
  - addresses for lines of code and static and global variables not yet assigned
  - need to perform *link* step on `hello.o` (using `gcc` or `ld`) to assign memory to each symbol
  - linking resolves symbols defined elsewhere (like the C standard library) and makes the code executable

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Functions and variables as symbols

- Let’s look at the symbols in the compiled file hello.o:
  `athena%`[1] `nm hello.o`
- Output:

```
0000000000000000 T main
0000000000000000 R msg
                 U puts
```

- ’T’ – (text) code; ’R’ – read-only memory; ’U’ - undefined symbol
- Addresses all zero before linking; symbols not allocated memory yet
- Undefined symbols are defined externally, resolved during linking

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Functions and variables as symbols

- Why aren't symbols listed for other declarations in `stdio.h`?
- Compiler doesn't bother creating symbols for unused function prototypes (saves space)
- What happens when we link?
  `athena%`[1] `gcc -Wall hello.o -o hello`
  - Memory allocated for defined symbols
  - Undefined symbols located in external libraries (like `libc` for C standard library)

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Functions and variables as symbols

- Let's look at the symbols now:
  `athena%`[1] `nm hello`
- Output:
  (other default symbols)
  ⋮

```
0000000000400524 T main
000000000040062c R msg
                 U puts@@GLIBC_2.2.5
```

- Addresses for static (allocated at compile time) symbols
- Symbol `puts` located in shared library GLIBC_2.2.5 (GNU C standard library)
- Shared symbol `puts` not assigned memory until run time

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Static and dynamic linkage

- Functions, global variables must be allocated memory before use
- Can allocate at compile time (static) or at run time (shared)
- Advantages/disadvantages to both
- Symbols in same file, other `.o` files, or static libraries (archives, `.a` files) – static linkage
- Symbols in shared libraries (`.so` files) – dynamic linkage
- `gcc` links against shared libraries by default, can force static linkage using `-static` flag

Static linkage

- What happens if we statically link against the library?

  `athena%`[1] `gcc -Wall -static hello.o -o hello`
- Our executable now contains the symbol `puts`:

```
⋮
00000000004014c0 W puts
⋮
0000000000400304 T main
⋮
000000000046cd04 R msg
⋮
```

- 'W': linked to another defined symbol

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Static linkage

- At link time, statically linked symbols added to executable
- Results in much larger executable file (static – 688K, dynamic – 10K)
- Resulting executable does not depend on locating external library files at run time
- To use newer version of library, have to recompile

Dynamic linkage

- Dynamic linkage occurs at run-time
- During compile, linker just looks for symbol in external shared libraries
- Shared library symbols loaded as part of program startup (before `main()`)
- Requires external library to define symbol exactly as expected from header file declaration
  - changing function in shared library can break your program
  - version information used to minimize this problem
  - reason why common libraries like `libc` rarely modify or remove functions, even broken ones like `gets()`

Linking external libraries

- Programs linked against C standard library by default
- To link against library `lib`*`namespec`*`.so` or `lib`*`namespec`*`.a`, use compiler flag `-l`*`namespec`* to link against library
- Library must be in library path (standard library directories + directories specified using `-L` *`directory`* compiler flag
- Use `-static` for force static linkage
- This is enough for static linkage; library code will be added to resulting executable

Loading shared libraries

- Shared library located during compile-time linkage, but needs to be located again during run-time loading
- Shared libraries located at run-time using linker library `ld.so`
- Whenever shared libraries on system change, need to run `ldconfig` to update links seen by `ld.so`
- During loading, symbols in dynamic library are allocated memory and loaded from shared library file

Loading shared libraries on demand

- In Linux, can load symbols from shared libraries on demand using functions in `dlfcn.h`
- Open a shared library for loading:

  ```
  void * dlopen(const char *file, int mode);
  ```

  values for mode: combination of `RTLD_LAZY` (lazy loading of library), `RTLD_NOW` (load now), `RTLD_GLOBAL` (make symbols in library available to other libraries yet to be loaded), `RTLD_LOCAL` (symbols loaded are accessible only to your code)

Loading shared libraries on demand

- Get the address of a symbol loaded from the library:

  ```c
  void * dlsym(void * handle, const char * symbol_name);
  ```

  handle from call to dlopen; returned address is pointer to variable or function identified by `symbol_name`
- Need to close shared library file handle after done with symbols in library:

  ```c
  int dlclose(void * handle);
  ```
- These functions are not part of C standard library; need to link against library `libdl`: `-ldl` compiler flag

Symbol resolution issues

- Symbols can be defined in multiple places
- Suppose we define our own `puts()` function
- But, `puts()` defined in C standard library
- When we call `puts()`, which one gets used?

Symbol resolution issues

- Symbols can be defined in multiple places
- Suppose we define our own `puts()` function
- But, `puts()` defined in C standard library
- When we call `puts()`, which one gets used?
- Our `puts()` gets used since ours is static, and `puts()` in C standard library not resolved until run-time
- If statically linked against C standard library, linker finds two `puts()` definitions and aborts (multiple definitions not allowed)

Symbol resolution issues

- How about if we define `puts()` in a shared library and attempt to use it within our programs?
- Symbols resolved in order they are loaded
- Suppose our library containing `puts()` is `libhello.so`, located in a standard library directory (like `/usr/lib`), and we compile our `hello.c` code against this library:

  `athena%`[1] `gcc -g -Wall hello.c -lhello -o hello.o`
- Libraries specified using `-l` flag are loaded in order specified, and before C standard library
- Which `puts()` gets used here?

  `athena% gcc -g -Wall hello.c -lc -lhello -o hello.o`

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

6.087 Lecture 9 – January 22, 2010

- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Creating libraries

- Libraries contain C code like any other program
- Static or shared libraries compiled from (un-linked) object files created using `gcc`
- Compiling a static library:
  - compile, but do not link source files:
    athena%[1] gcc -g -Wall -c *infile.c* -o *outfile.o*
  - collect compiled (unlinked) files into an archive:
    athena% ar -rcs lib*name*.a *outfile1.o outfile2.o* ...

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Creating shared libraries

- Compile and do not link files using `gcc`:
  `athena%`[1] `gcc -g -Wall -fPIC -c` *`infile.c`* `-o` *`outfile.o`*
- `-fPIC` option: create position-independent code, since code will be repositioned during loading
- Link files using `ld` to create a shared object (`.so`) file:
  `athena% ld -shared -soname lib`*`name`*`.so -o lib`*`name`*`.so.`*`version`* `-lc` *`outfile1.o outfile2.o`* `...`
- If necessary, add directory to `LD_LIBRARY_PATH` environment variable, so `ld.so` can find file when loading at run-time
- Configure `ld.so` for new (or changed) library:
  `athena% ldconfig -v`

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

6.087 Lecture 9 – January 22, 2010

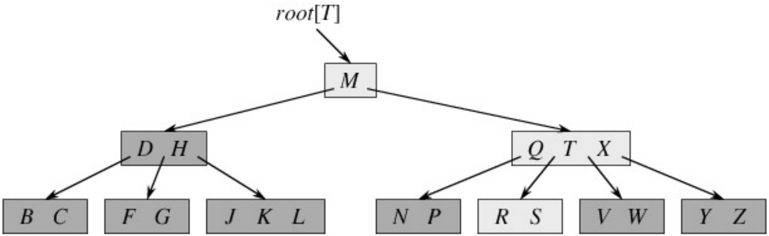
- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Data structures

- Many data structures designed to support certain algorithms
- B-tree - generalized binary search tree, used for databases and file systems
- Priority queue - ordering data by “priority,” used for sorting, event simulation, and many other algorithms

B-tree structure

- Binary search tree with variable number of children (at least t, up to $2t$)
- Tree is balanced – all leaves at same level
- Node contains list of “keys” – divide range of elements in children



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Initializing a B-tree

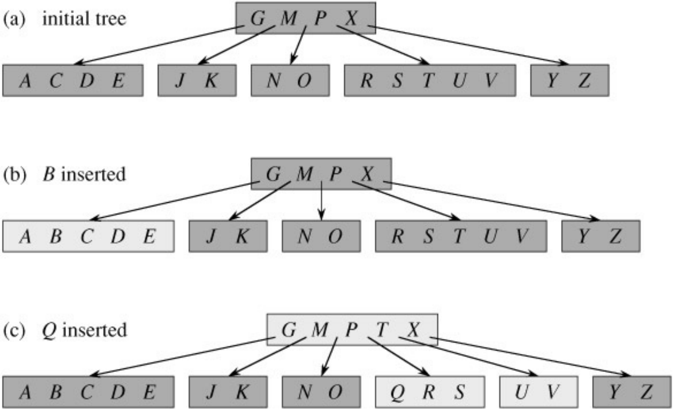
- Initially, B-tree contains root node with no children (leaf node), no keys
- Note: root node exempt from minimum children requirement

Inserting elements

- Insertion complicated due to maximum number of keys
- At high level:
  1. traverse tree down to leaf node
  2. if leaf already full, split into two leaves:
     (a) move median key element into parent (splitting parent already full)
     (b) split remaining keys into two leaves (one with lower, one with higher elements)
  3. add element to sorted list of keys
- Can accomplish in one pass, splitting full parent nodes during traversal in step 1

Inserting elements

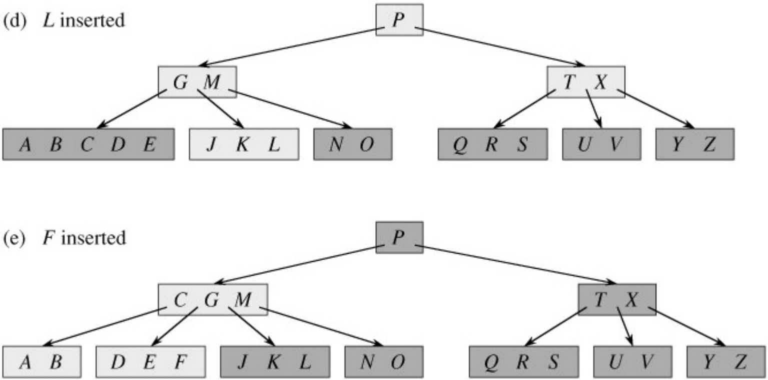
B-tree with $t = 3$ (nodes may have 2–5 keys):



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Inserting elements

More insertion examples:



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

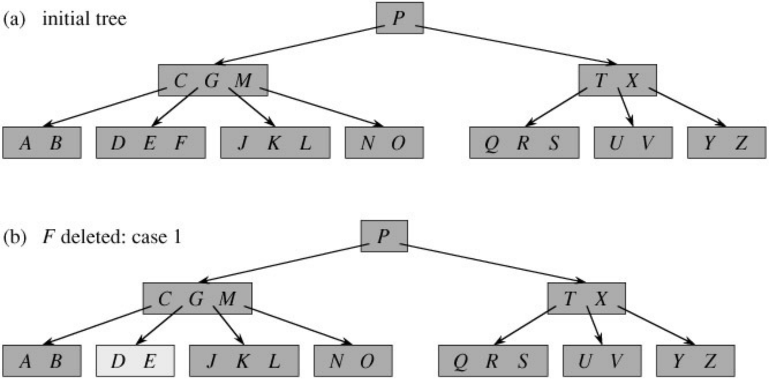
Searching a B-tree

- Search like searching a binary search tree:
    1. start at root.
    2. if node empty, element not in tree
    3. search list of keys for element (using linear or binary search)
    4. if element in list, return element
    5. otherwise, element between keys, and repeat search on child node for that range
- Tree is balanced – search takes O$(\log n)$ time

Deletion

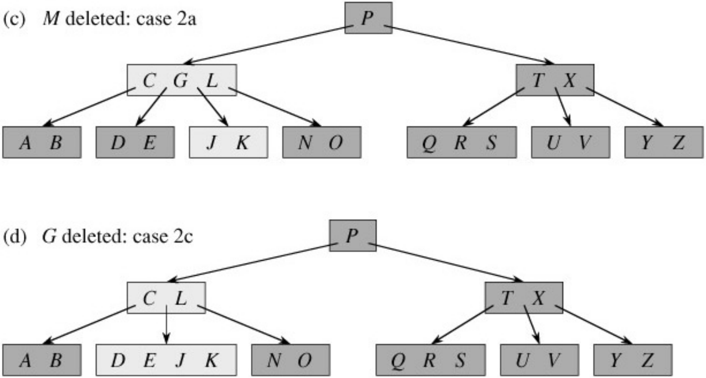
- Deletion complicated by minimum children restriction
- When traversing tree to find element, need to ensure child nodes to be traversed have enough keys
  - if adjacent child node has at least t keys, move separating key from parent to child and closest key in adjacent child to parent
  - if no adjacent child nodes have extra keys, merge child node with adjacent child
- When removing a key from a node with children, need to rearrange keys again
  - if child before or after removed key has enough keys, move closest key from child to parent
  - if neither child has enough keys, merge both children
  - if child not a leaf, have to repeat this process

Deletion examples



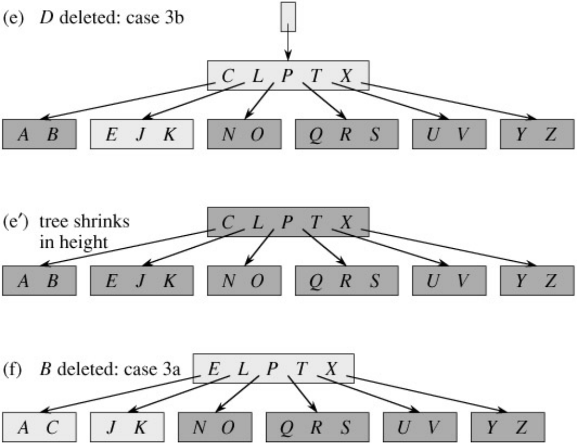
[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Deletion examples



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Deletion examples



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Priority queue

- Abstract data structure ordering elements by priority
- Elements enqueued with priority, dequeued in order of highest priority
- Common implementations: heap or binary search tree
- Operations: insertion, peek/extract max-priority element, increase element priority

Heaps

- Heap - tree with heap-ordering property: priority(child) $\leq$ priority(parent)
- More sophisticated heaps exist – e.g. binomial heap, Fibonacci heap
- We'll focus on simple binary heaps
- Usually implemented as an array with top element at beginning
- Can sort data using a heap – O$(n \log n)$ worst case in-place sort!

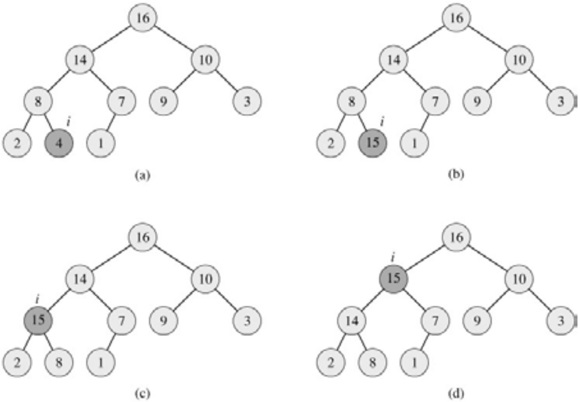
Extracting data

- Heap-ordering property $\Rightarrow$ maximum priority element at top of heap
- Can peek by looking at top element
- Can remove top element, move last element to top, and swap top element down with its children until it satisfies heap-ordering property:
  1. start at top
  2. find largest of element and left and right child; if element is largest, we are done
  3. otherwise, swap element with largest child and repeat with element in new position

Inserting data/increasing priority

- Insert element at end of heap, set to lowest priority $-\infty$
- Increase priority of element to real priority:
    1. start at element
    2. if new priority less than parent's, we are done
    3. otherwise, swap element with parent and repeat

Example of inserting data



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Summary

Topics covered:

- Using external libraries
  - symbols and linkage
  - static vs. dynamic linkage
  - linking to your code
  - symbol clashing
- Creating libraries
- Data structures
  - B-tree
  - priority queue

6.087 Practical Programming in C
January (IAP) 2010

Outline

- Review
- Standard Library
  - <stdio.h>
  - <ctype.h>
  - <stdlib.h>
  - <assert.h>
  - <stdarg.h>
  - <time.h>

6.087 Lecture 10 – January 25, 2010

- Review
- Standard Library
  - <stdio.h>
  - <ctype.h>
  - <stdlib.h>
  - <assert.h>
  - <stdarg.h>
  - <time.h>

Review: Libraries

- linking: binds symbols to addresses.
- static linkage: occurs at compile time (static libraries).
- dynamic linkage: occurs at run time (shared libraries).
- shared libraries:
  - ld.so - locates shared libraries
  - ldconfig - updates links seen by ld.so
  - `dlopen()`,`dlsym()`,`dlclose()` - load shared libraries on demand.
- compiling static libraries: gcc,ar
- compiling shared libraries: gcc,ldconfig

Review: BTree

- generalized search tree–multiple children.
- except for root, each node can have between t and $2t$ children.
- tree is **always** balanced.
- Used in file systems, databases etc.

Review: Priority Queue

- abstract data structure: many implementations
- common implementations: heaps,bst,linked list
- elements are queued and dequeued in order of priority.
- operations:
  `peek(),insert(),extract-max()/extract-min()`

6.087 Lecture 10 – January 25, 2010

- Review
- Standard Library
  - <stdio.h>
  - <ctype.h>
  - <stdlib.h>
  - <assert.h>
  - <stdarg.h>
  - <time.h>

<stdio.h>: Opening, closing files

```
FILE* fopen(const char* filename,const char* mode)
```

- mode can be "r"(read),"w"(write),"a"(append).
- "b" can be appended for binary input/output (unnecessary in *nx)
- returns NULL on error.

```
FILE* freopen(const char* filename,const char* mode,FILE* stream)
```

- redirects the stream to the file.
- returns NULL on error.
- Where can this be used? (redirecting stdin,stdout,stderr)

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

- flushes any unwritten data.
- if stream is NULL flushes all outputs streams.
- returns EOF on error.

<stdio.h>: File operations

```c
int remove(const char* filename)
```

- removes the file from the file system.
- retrn non-zero on error.

```c
int rename(const char* oldname,const char* newname)
```

- renames file
- returns non-zero on error (reasons?: permission, existence)

<stdio.h>:Temporary files

`FILE* tmpfile(void)`

- creates a temporary file with mode "wb+".
- the file is removed **automatically** when program terminates.

`char* tmpnam(char s[L_tmpnam])`

- creates a string that is not the name of an existing file.
- return reference to internal static array if s is NULL. Populate `s` otherwise.
- generates a new name every call.

<stdio.h>: Raw I/O

```
size_t fread(void* ptr, size_t size, size_t nobj, FILE* stream)
```

- reads at most `nobj` items of size `size` from stream into ptr.
- returns the number of items read.
- `feof` and `ferror` must be used to test end of file.

```
size_t fwrite (const void* ptr, size_t size, size_t nobj, FILE* stream)
```

- write at most `nobj` items of size `size` from `ptr` onto `stream`.
- returns number of objects written.

<stdio.h>: File position

int fseek(FILE* stream, **long** offset, **int** origin)

- sets file position in the stream. Subsequent read/write begins at this location
- origin can be `SEEK_SET`, `SEEK_CUR`, `SEEK_END`.
- returns non-zero on error.

long ftell (FILE* stream)

- returns the current position within the file. (limitation? long data type).
- returns -1L on error.

int rewind(FILE* stream)

- sets the file pointer at the beginning.
- equivalent to fseek(stream,0L,SEEK_SET);

<stdio.h>: File errors

`void clearerr(FILE* stream)`

- clears EOF and other error indicators on stream.

`int feof(FILE* stream)`

- return non-zero (TRUE) if end of file indicator is set for stream.
- only way to test end of file for functions such as `fwrite()`,`fread()`

`int ferror(FILE* stream)`

- returns non-zero (TRUE) if **any** error indicator is set for stream.

<ctype.h>: Testing characters

isalnum(c)	isalpha(c) || isdigit (c)
iscntrl (c)	control characters
isdigit (c)	0-9
islower(c)	'a'-'z'
isprint (c)	printable character (includes space)
ispunct(c)	punctuation
isspace(c)	space, tab or new line
isupper(c)	'A'-'Z'

<string.h>: Memory functions

```
void* memcpy(void* dst,const void* src,size_t n)
```

- copies `n` bytes from `src` to location `dst`
- returns a pointer to `dst`.
- `src` and `dst` **cannot overlap.**

```
void* memmove(void* dst,const void* src,size_t n)
```

- behaves same as `memcpy()` function.
- `src` and `dst` **can overlap.**

```
int memcmp(const void* cs,const void* ct,int n)
```

- compares first `n` bytes between `cs` and `ct`.

```
void* memset(void* dst,int c,int n)
```

- fills the first `n` bytes of `dst` with the value `c`.
- returns a pointer to `dst`

<stdlib.h>:Utility

```
double atof(const char* s)
int  atoi (const char* s)
long atol(const char* s)
```

- converts character to float,integer and long respectively.

```
int  rand()
```

- returns a pseduo-random numbers between 0 and RAND_MAX

```
void srand(unsigned int seed)
```

- sets the seed for the pseudo-random generator!

<stdlib.h>: Exiting

`void abort(void)`

- causes the program to terminate abnormally.

`void exit(int status)`

- causes normal program termination. The value `status` is returned to the operating system.
- 0 `EXIT_SUCCESS` indicates successful termination. Any other value indicates failure (`EXIT_FAILURE`)

<stdlib.h>:Exiting

void atexit (**void** (*fcn)(**void**))

- *registers* a function `fcn` to be called when the program terminates normally;
- returns non zero when registration cannot be made.
- After `exit()` is called, the functions are called in reverse order of registration.

int system(**const char*** cmd)

- executes the command in string `cmd`.
- if cmd is not null, the program executes the command and returns exit status returned by the command.

<stdlib.h>:Searchign and sorting

```
void* bsearch(const void* key,const void* base,
    size_t n,size_t size,
    int (*cmp)(const void* keyval,const void* datum));
```

- searches `base[0]` through `base[n-1]` for `*key`.
- function `cmp()` is used to perform comparison.
- returns a pointer to the matching item if it exists and NULL otherwise.

```
void qsort(void* base,size_t n,
           size_t sz,
           int (*cmp)(const void*,const void*))!
```

- sorts `base[0]` through `base[n-1]` in ascending/descending order.
- function `cmp()` is used to perform comparison.

<assert.h>:Diagnostics

```
void assert(int expression)
```

- used to check for invariants/code consistency during debugging.
- does nothing when expression is true.
- prints an error message indicating, expression, filename and line number.

Alternative ways to print filename and line number during execution is to use: `__FILE__`, `__LINE__` macros.

<stdarg.h>:Variable argument lists

Variable argument lists:

- functions can variable number of arguments.
- the data type of the argument can be different for each argument.
- atleast one mandatory argument is required.
- Declaration:

```
int printf (char* fmt ,...); /*fmt is last named argument*/
```

```
va_list ap
```

- `ap` defines an iterator that will point to the variable argument.
- before using, it has to be initialized using `va_start`.

<stdarg.h>:Variable argument list

$\boxed{\text{va\_start(va\_list ap, lastarg)}}$

- `ap` lastarg refers to the **name** of the last named argument.
- va_start is a macro.

$\boxed{\text{va\_arg(va\_list ap, type)}}$

- each call of `va_arg` points ap to the next argument.
- type has to be inferred from the fixed argument (e.g. printf) or determined based on previous argument(s).

$\boxed{\text{va\_end(va\_list ap)}}$

- must be called before the function is exited.

<stdarg.h>:Variable argument list(cont.)

```
int sum(int num,...)
{
    va_list ap;int total=0;
    va_start(ap,num);
    while(num>0)
    {
        total+=va_arg(ap,int);
        num--;
    }
    va_end(ap);
    return total;
}

int suma=sum(4,1,2,3,4);/*called with five args*/
int sumb=sum(2,1,2); /*called with three args*/
```

<time.h>

time_t, clock_t, **struct** tm data types associated with time.

struct tm:

int tm_sec	seconds
int tm_min	minutes
int tm_hour	hour since midnight (0,23)
int tm_mday	day of the month (1,31)
int tm_mon	month
int tm_year	years since 1900
int tm_wday	day since sunday (0,6)
int tm_yday	day since Jan 1 (0,365)
int tm_isdst	DST flag

<time.h>

```
clock_t clock()
```

- returns processor time used since beginning of program.
- divide by CLOCKS_PER_SEC to get time in seconds.

```
time_t time(time_t * tp)
```

- returns current time (seconds since Jan 1 1970).
- if tp is not NULL, also populates tp.

```
double difftime(time_t t1,time_t t2)
```

- returns difference in seconds.

```
time_t mktime(struct tm* tp)
```

- converts the structure to a time_t object.
- returns -1 if conversion is not possible.

<time.h>

char* asctime(**const struct** tm* tp)

- returns string representation of the form "Sun Jan 3 15:14:13 1988".
- returns static reference (can be overwritten by other calls).

struct tm* localtime(**const** time_t * tp)

- converts **calendar time** to local time".

char* ctime(**const** time_t * tp)

- converts **calendar time** to string representation of local time".
- equivalent to sctime(locltime(tp))!

<time.h>

size_t strftime (**char*** s,size_t smax,**const char*** fmt,**const struct** tm* tp)

- returns time in the desired format.
- does not write more than `smax` characters into the string `s`.

%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%d	day of the month
%H	hour (0-23)
%I	hour (0-12)
%m	month
%M	minute
%p	AM/PM
%S	second

6.087 Practical Programming in C
January (IAP) 2010

6.087 Lecture 11 – January 26, 2010

- Review
- Dynamic Memory Allocation
  - Designing the `malloc()` Function
  - A Simple Implementation of `malloc()`
  - A Real-World Implementation of `malloc()`
- Using `malloc()`
  - Using `valgrind`
- Garbage Collection

Review: C standard library

- I/O functions: `fopen()`, `freopen()`, `fflush()`, `remove()`, `rename()`, `tmpfile()`, `tmpnam()`, `fread()`, `fwrite()`, `fseek()`, `ftell()`, `rewind()`, `clearerr()`, `feof()`, `ferror()`
- Character testing functions: `isalpha()`, `isdigit()`, `isalnum()`, `iscntrl()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`
- Memory functions: `memcpy()`, `memmove()`, `memcmp()`, `memset()`

Review: C standard library

- Conversion functions: `atoi()`, `atol()`, `atof()`, `strtol()`, `strtoul()`, `strtod()`
- Utility functions: `rand()`, `srand()`, `abort()`, `exit()`, `atexit()`, `system()`, `bsearch()`, `qsort()`
- Diagnostics: `assert()` function, `__FILE__`, `__LINE__` macros

Review: C standard library

- Variable argument lists:
  - Declaration with `...` for variable argument list (may be of any type):
    int printf (**const char** * fmt, ...);
  - Access using data structure `va_list ap`, initialized using `va_start()`, accessed using `va_arg()`, destroyed at end using `va_end()`
- Time functions: `clock()`, `time()`, `difftime()`, `mktime()`, `asctime()`, `localtime()`, `ctime()`, `strftime()`

- Review
- Dynamic Memory Allocation
  - Designing the `malloc()` Function
  - A Simple Implementation of `malloc()`
  - A Real-World Implementation of `malloc()`
- Using `malloc()`
  - Using `valgrind`
- Garbage Collection

Dynamic memory allocation

- Memory allocated during runtime
- Request to map memory using `mmap()` function (in `<sys/mman.h>`)
- Virtual memory can be returned to OS using `munmap()`
- Virtual memory either backed by a file/device or by *demand-zero* memory:
  - all bits initialized to zero
  - not stored on disk
  - used for stack, heap, uninitialized (at compile time) globals

Mapping memory

- Mapping memory:

```
void *mmap(void *start, size_t length, int prot,
           int flags, int fd, off_t offset);
```

  - asks OS to map virtual memory of specified length, using specified physical memory (file or demand-zero)
  - `fd` is file descriptor (integer referring to a file, not a file stream) for physical memory (i.e. file) to load into memory
  - for demand-zero, including the heap, use `MMAP_ANON` flag
  - `start` – suggested starting address of mapped memory, usually NULL

- Unmap memory:

```
int munmap(void *start, size_t length);
```

The heap

- Heap – private section of virtual memory (demand-zero) used for dynamic allocation
- Starts empty, zero-sized
- `brk` – OS pointer to top of heap, moves upwards as heap grows
- To resize heap, can use `sbrk()` function:

```
void *sbrk(int inc); /* returns old value of brk_ptr */
```

- Functions like `malloc()` and `new` (in C++) manage heap, mapping memory as needed
- Dynamic memory allocators divide heap into blocks

Requirements

- Must be able to allocate, free memory in any order
- Auxiliary data structure must be on heap
- Allocated memory cannot be moved
- Attempt to minimize fragmentation

Fragmentation

- Two types – internal and external
- Internal – block size larger than allocated variable in block
- External – free blocks spread out on heap
- Minimize external fragmentation by preferring fewer larger free blocks

Design choices

- Data structure to track blocks
- Algorithm for positioning a new allocation
- Splitting/joining free blocks

Tracking blocks

- Implicit free list: no data structure required
- Explicit free list: heap divided into fixed-size blocks; maintain a linked list of free blocks
  - allocating memory: remove allocated block from list
  - freeing memory: add block back to free list
- Linked list iteration in linear time
- Segregated free list: multiple linked lists for blocks of different sizes
- Explicit lists stored within blocks (pointers in payload section of free blocks)

Block structures

Figure removed due to copyright restrictions. Please see http://csapp.cs.cmu.edu/public/1e/public/figures.html, Figure 10.37, Format of a simple heap block.

Block structures

Figure removed due to copyright restrictions. Please see
http://csapp.cs.cmu.edu/public/1e/public/figures.html,
Figure 10.50, Format of heap blocks that use doubly-linked free lists.

Positioning allocations

- Block must be large enough for allocation
- First fit: start at beginning of list, use first block
- Next fit: start at end of last search, use next block
- Best fit: examines entire free list, uses smallest block
- First fit and next fit can fragment beginning of heap, but relatively fast
- Best fit can have best memory utilization, but at cost of examining entire list

Splitting and joining blocks

- At allocation, can use entire free block, or part of it, splitting the block in two
- Splitting reduces internal fragmentation, but more complicated to implement
- Similarly, can join adjacent free blocks during (or after) freeing to reduce external fragmentation
- To join (coalesce) blocks, need to know address of adjacent blocks
- Footer with pointer to head of block – enable successive block to find address of previous block

A simple memory allocator

- Code in *Computer Systems: A Programmer's Perspective*
- Payload 8 byte alignment; 16 byte minimum block size
- Implicit free list
- Coalescence with boundary tags; only split if remaining block space ≥ 16 bytes

Initialization

1. Allocate 16 bytes for padding, prologue, epilogue
2. Insert 4 byte padding and prologue block (header + footer only, no payload) at beginning
3. Add an epilogue block (header only, no payload)
4. Insert a new free chunk (extend the heap)

Allocating data

1. Compute total block size (header+payload+footer)
2. Locate free block large enough to hold data (using first or next fit for speed)
3. If block found, add data to block and split if padding ≥ 16 bytes
4. Otherwise, insert a new free chunk (extending the heap), and add data to that
5. If could not add large enough free chunk, out of memory

Freeing data

1. Mark block as free (bit flag in header/footer)
2. If previous block free, coalesce with previous block (update size of previous)
3. If next block free, coalesce with next block (update size)

Explicit free list

- Maintain pointer to head, tail of free list (not in address order)
- When freeing, add free block to end of list; set pointer to next, previous block in free list at beginning of payload section of block
- When allocating, iterate through free list, remove from list when allocating block
- For segregated free lists, allocator maintains array of lists for different sized free blocks

`malloc()` for the real world

- Used in GNU libc version of `malloc()`
- Details have changed, but nice general discussion can be found at
  `http://g.oswego.edu/dl/html/malloc.html`
- Chunks implemented as in segregated free list, with pointers to previous/next chunks in free list in payload of free blocks
- Lists segregated into bins according to size; bin sizes spaced logarithmically
- Placement done in best-fit order
- Deferred coalescing and splitting performed to minimize overhead

6.087 Lecture 11 – January 26, 2010

- Review
- Dynamic Memory Allocation
  - Designing the `malloc()` Function
  - A Simple Implementation of `malloc()`
  - A Real-World Implementation of `malloc()`
- Using `malloc()`
  - Using `valgrind`
- Garbage Collection

Using `malloc()`

- Minimize overhead – use fewer, larger allocations
- Minimize fragmentation – reuse memory allocations as much as possible
- Growing memory – using `realloc()` can reduce fragmentation
- Repeated allocation and freeing of variables can lead to poor performance from unnecessary splitting/coalescing (depending on implementation of `malloc()`)

Using `valgrind` to detect memory leaks

- A simple tutorial: `http://cs.ecs.baylor.edu/~donahoo/tools/valgrind/`
- `valgrind` program provides several performance tools, including `memcheck`:

  ```
  athena%[1] valgrind --tool=memcheck
  --leak-check=yes program.o
  ```

- `memcheck` runs program using virtual machine and tracks memory leaks
- Does not trigger on out-of-bounds index errors for arrays on the stack

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Other `valgrind` tools

- Can use to profile code to measure memory usage, identify execution bottlenecks
- `valgrind` tools (use name in `-tool=` flag):
  - `cachegrind` – counts cache misses for each line of code
  - `callgrind` – counts function calls and costs in program
  - `massif` – tracks overall heap usage

6.087 Lecture 11 – January 26, 2010

- Review
- Dynamic Memory Allocation
  - Designing the `malloc()` Function
  - A Simple Implementation of `malloc()`
  - A Real-World Implementation of `malloc()`
- Using `malloc()`
  - Using `valgrind`
- Garbage Collection

Garbage collection

- C implements no garbage collector
- Memory not freed remains in virtual memory until program terminates
- Other languages like Java implement garbage collectors to free unreferenced memory
- When is memory unreferenced?

Garbage collection

- C implements no garbage collector
- Memory not freed remains in virtual memory until program terminates
- Other languages like Java implement garbage collectors to free unreferenced memory
- When is memory unreferenced?
  - Pointer(s) to memory no longer exist
  - Tricky when pointers on heap or references are circular (think of circular linked lists)
  - Pointers can be masked as data in memory; garbage collector may free data that is still referenced (or not free unreferenced data)

Garbage collection and memory allocation

- Program relies on garbage collector to free memory
- Garbage collector calls `free()`
- `malloc()` may call garbage collector if memory allocation above a threshold

Mark and sweep garbage collector

- Simple tracing garbage collector
- Starts with list of known in-use memory (e.g. the stack)
- Mark: trace all pointers, marking data on the heap as it goes
- Sweep: traverse entire heap, freeing unmarked data
- Requires two complete traversals of memory, takes a lot of time
- Implementation available at `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

Mark and sweep garbage collector

Figure removed due to copyright restrictions. Please see
http://csapp.cs.cmu.edu/public/1e/public/figures.html,
Figure 10.51, A garbage collector's view of memory as a directed graph.

Mark and sweep garbage collector

Figure removed due to copyright restrictions. Please see http://csapp.cs.cmu.edu/public/1e/public/figures.html, Figure 10.54, Mark and sweep example.

Copying garbage collector

- Uses a duplicate heap; copies live objects during traversal to the duplicate heap (the *to-space*)
- Updates pointers to point to new object locations in duplicate heap
- After copying phase, entire old heap (the *from-space*) is freed
- Code can only use half the heap

Cheney's (not Dick's) algorithm

- Method for copying garbage collector using breadth-first-search of memory graph
- Start with empty to-space
- Examine stack; move pointers to to-space and update pointers to to-space references
- Items in from-space replaced with pointers to copy in to-space
- Starting at beginning of to-space, iterate through memory, doing the same as pointers are encountered
- Can accomplish in one pass

Summary

Topics covered:

- Dynamic memory allocation
  - the heap
  - designing a memory allocator
  - a real world allocator
- Using `malloc()`
- Using `valgrind`
- Garbage collection
  - mark-and-sweep collector
  - copying collector

6.087 Practical Programming in C
January (IAP) 2010

Outline

- Review
- Multithreaded programming
  - Concepts
- Pthread
  - API
  - Mutex
  - Condition variables

6.087 Lecture 12 – January 27, 2010

- Review

- Multithreaded programming
  - Concepts

- Pthread
  - API
  - Mutex
  - Condition variables

Review: malloc()

- Mapping memory: `mmap()`, `munmap()`. Useful for demand paging.
- Resizing heap: `sbrk()`
- Designing `malloc()`
  - implicit linked list,explicit linked list
  - best fit,first fit,next fit
- Problems:
  - fragmentation
  - memory leaks
  - valgrind –tool=memcheck, checks for memory leaks.

Garbage collection

- C does not have any garbage collectors
- Implementations available
- Types:
  - Mark and sweep garbage collector (depth first search)
  - Cheney's algorithm (breadth first search)
  - Copying garbage collector

6.087 Lecture 12 – January 27, 2010

- Review
- Multithreaded programming
  - Concepts
- Pthread
  - API
  - Mutex
  - Condition variables

Preliminaries: Parallel computing

- Parallelism: Multiple computations are done simultaneously.
  - Instruction level (pipelining)
  - Data parallelism (SIMD)
  - Task parallelism (embarrassingly parallel)
- Concurrency: Multiple computations that **may** be done in parallel.
- Concurrency vs. Parallelism

Process vs. Threads

- Process: An instance of a program that is being executed in its **own** address space. In POSIX systems, each process maintains its own heap, stack, registers, file descriptors etc.
  Communication:
  - Shared memory
  - Network
  - Pipes, Queues
- Thread: A light weight process that shares its address space with others.In POSIX systems, each thread maintains the bare essentials: registers, stack, signals.
  Communication:
  - shared address space.

Multithreaded concurrency

Serial execution:

- All our programs so far has had a single thread of execution: main thread.
- Program exits when the main thread exits.

Multithreaded:

- Program is organized as multiple and concurrent threads of execution.
- The main thread *spawns* multiple threads.
- The thread **may** communicate with one another.
- Advantages:
  - Improves performance
  - Improves responsiveness
  - Improves utilization
  - less overhead compared to multiple processes

Multithreaded programming

Even in C, multithread programming may be accomplished in several ways

- Pthreads: POSIX C library.
- OpenMP
- Intel threading building blocks
- Cilk (from CSAIL!)
- Grand central despatch
- CUDA (GPU)
- OpenCL (GPU/CPU)

Not all code can be made parallel

```
float params[10];
for(int i=0;i<10;i++)
  do_something(params[i]);
```

```
float params[10];
float prev=0;
for(int i=0;i<10;i++)
{
  prev=complicated(params[i],prev);
}
```

paralleizable	not parallelizable

Not all multi-threaded code is safe

```c
int balance=500;
void deposit(int sum){
  int currbalance=balance;/*read balance*/
  ...
  currbalance+=sum;
  balance=currbalance;/*write balance*/
}

void withdraw(int sum){
  int currbalance=balance;/*read balance*/
  if(currbalance>0)
    currbalance-=sum;
  balance=currbalance; /*write balance*/
}
  ..
  deposit(100);/*thread 1*/
  ..
  withdraw(50);/thread 2*/
  ..
  withdraw(100);/*thread 3*/
    ...
```

- minimize use of global/static memory
- Scenario: T1(read),T2(read,write),T1(write) ,balance=600
- Scenario: T2(read),T1(read,write),T2(write) ,balance=450

6.087 Lecture 12 – January 27, 2010

- Review
- Multithreaded programming
  - Concepts
- Pthread
  - API
  - Mutex
  - Condition variables

Pthread

API:

- Thread management: creating, joining, attributes

  pthread_

- Mutexes: create, destroy mutexes

  pthread_mutex_

- Condition variables: create,destroy,wait,signal

  pthread_cond_

- Synchronization: read/write locks and barriers

  pthread_rwlock_, pthread_barrier_

API:

- **#include** <pthread.h>
- gcc –Wall –O0 –o <output> file.c –pthread (no –l prefix)

Creating threads

```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void *(*start_routine)(void*), void * arg);
```

- creates a new thread with the attributes specified by `attr`.
- Default attributes are used if `attr` is NULL.
- On success, stores the thread it into `thread`
- calls function `start_routine(arg)` on a separate thread of execution.
- returns zero on success, non-zero on error.

```
void pthread_exit(void *value_ptr);
```

- called implicitly when thread function exits.
- analogous to `exit()`.

Example

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```


code: `https://computing.llnl.gov/tutorials/pthreads/`

Output

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #2!
Hello World! It's me, thread #3!
In main: creating thread 4
Hello World! It's me, thread #4!
```

```
In main: creating thread 0
Hello World! It's me, thread #0!
In main: creating thread 1
Hello World! It's me, thread #1!
In main: creating thread 2
Hello World! It's me, thread #2!
In main: creating thread 3
Hello World! It's me, thread #3!
In main: creating thread 4
Hello World! It's me, thread #4!
```

Synchronization: joining

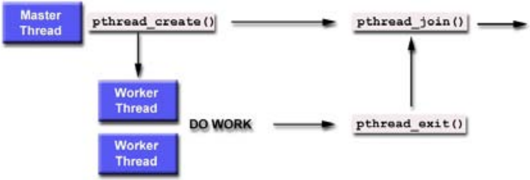


Figure: https://computing.llnl.gov/tutorials/pthreads


```
int pthread_join(pthread_t thread, void **value_ptr);
```

- pthread_join() blocks the calling thread until the specified thread terminates.
- If value_ptr is not null, it will contain the return status of the called thread

Other ways to synchronize: mutex,condition variables

Example

```c
#define NELEMENTS 5000
#define BLK_SIZE 1000
#define NTHREADS (NELEMENTS/BLK_SIZE)

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;long t; void *status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, work, (void *)(t*BLK_SIZE));
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc); exit(-1);
       }
  }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);exit(-1);
      }
   }
   printf("Main: program completed. Exiting.\n");
}
```

Mutex

- Mutex (mutual exclusion) acts as a "lock" protecting access to the shared resource.
- Only one thread can "own" the mutex at a time. Threads must take turns to lock the mutex.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t * mutex,
                const pthread_mutexattr_t * attr);
thread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- pthread_mutex_init() initializes a mutex. If attributes are NULL, default attributes are used.
- The macro PTHREAD_MUTEX_INITIALIZER can be used to initialize static mutexes.
- pthread_mutex_destroy() destroys the mutex.
- Both function return return 0 on success, non zero on error.

Mutex

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- pthread_mutex_lock() locks the given mutex. If the mutex is locked, the function is blocked until it becomes available.
- pthread_mutex_trylock() is the non-blocking version. If the mutex is currently locked the call will return immediately.
- pthread_mutex_unlock() unlocks the mutex.

Example revisited

```
int balance=500;
void deposit(int sum){
  int currbalance=balance;/*read balance*/
  ...
  currbalance+=sum;
  balance=currbalance;/*write balance*/
}

void withdraw(int sum){
  int currbalance=balance;/*read balance*/
  if(currbalance>0)
    currbalance−=sum;
  balance=currbalance; /*write balance*/
}
  ..
  deposit(100);/*thread 1*/
  ..
  withdraw(50);/thread 2*/
  ..
  withdraw(100);/*thread 3*/
    ...
```

- Scenario: T1(read),T2(read,write),T1(write),balance=600
- Scenario: T2(read),T1(read,write),T2(write),balance=450

Using mutex

```
int balance=500;
pthread_mutex_t mutexbalance=PTHREAD_MUTEX_INITIALIZER;

void deposit(int sum){
  pthread_mutex_lock(&mutexbalance);
  {
  int currbalance=balance;/*read balance*/
  ...
  currbalance+=sum;
  balance=currbalance;/*write balance*/
  }
  pthread_mutex_unlock(&mutexbalance);
}
void withdraw(int sum){
  pthread_mutex_lock(&mutexbalance);
  {
  int currbalance=balance;/*read balance*/
  if(currbalance>0)
    currbalance-=sum;
  balance=currbalance; /*write balance*/
  }
  pthread_mutex_unlock(&mutexbalance);
}
..  deposit(100);/*thread 1*/
..  withdraw(50);/thread 2*/
..  withdraw(100);/*thread 3*/
```

- Scenario: T1(read,write),T2(read,write),balance=550
- Scenario: T2(read),T1(read,write),T2(write),balance=550

Condition variables

Sometimes locking or unlocking is based on a run-time condition (examples?).Without condition variables, program would have to poll the variable/condition continuously.

Consumer:

(a) lock mutex on global item variable

(b) wait for (item>0) signal from producer (mutex unlocked automatically).

(c) wake up when signalled (mutex locked again automatically), unlock mutex and proceed.

Producer:

(1) produce something

(2) Lock global item variable, update item

(3) signal waiting (threads)

(4) unlock mutex

Condition variables

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t * cond, const pthread_condattr_t * attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- pthread_cond_init() initialized the condition variable. If attr is NULL, default attributes are sed.
- pthread_cond_destroy() will destroy (uninitialize) the condition variable.
- destroying a condition variable upon which other threads are currently blocked results in undefined behavior.
- macro PTHREAD_COND_INITIALIZER can be used to initialize condition variables. No error checks are performed.
- Both function return 0 on success and non-zero otherwise.

Condition variables

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t * cond,const pthread_condattr_t * attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- pthread_cond_init() initialized the condition variable. If attr is NULL, default attributes are sed.
- pthread_cond_destroy() will destroy (uninitialize) the condition variable.
- destroying a condition variable upon which other threads are currently blocked results in undefined behavior.
- macro PTHREAD_COND_INITIALIZER can be used to initialize condition variables. No error checks are performed.
- Both function return 0 on success and non-zero otherwise.

Condition variables

```
int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex);
```

- blocks on a condition variable.
- must be called with the mutex already locked otherwise behavior undefined.
- automatically releases mutex
- upon successful return, the mutex will be automatically locked again.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- unblocks threads waiting on a condition variable.
- `pthread_cond_broadcast()` unlocks **all** threads that are waiting.
- `pthread_cond_signal()` unlocks **one of** the threads that are waiting.
- both return 0 on success, non zero otherwise.

Example

```c
#include<pthread.h>
pthread_cond_t  cond_recv=PTHREAD_COND_INITIALIZER;
pthread_cond_t  cond_send=PTHREAD_COND_INITIALIZER;
pthread_mutex_t cond_mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t count_mutex=PTHREAD_MUTEX_INITIALIZER;

int full=0;
int count=0;
```

```c
void* produce(void*)
{
  while(1)
  {
      pthread_mutex_lock(&cond_mutex);
    while(full)
  {
    pthread_cond_wait(&cond_recv,
              &cond_mutex);
  }
      pthread_mutex_unlock(&cond_mutex);
      pthread_mutex_lock(&count_mutex);
      count++;full=1;
      printf("produced(%d):%d\n",
      pthread_self(),count);
      pthread_cond_broadcast(&cond_send);
      pthread_mutex_unlock(&count_mutex);
      if(count>=10) break;
    }
```

```c
void* consume(void*)
{
  while(1)
    {
      pthread_mutex_lock(&cond_mutex);
      while(!full)
  {
    pthread_cond_wait(&cond_send,
    &cond_mutex);
  }
      pthread_mutex_unlock(&cond_mutex);
      pthread_mutex_lock(&count_mutex);
      full=0;
      printf("consumed(%ld):%d\n",
      pthread_self(),count);
      pthread_cond_broadcast(&cond_recv);
      pthread_mutex_unlock(&count_mutex);
      if(count>=10)break;
    }
```

Example

```c
int main()
{
  pthread_t cons_thread, prod_thread;
  pthread_create(&prod_thread, NULL, produce, NULL);
  pthread_create(&cons_thread, NULL, consume, NULL);

  pthread_join(cons_thread, NULL);
  pthread_join(prod_thread, NULL);
  return 0;
}
```

Output:

```
produced(3077516144):1
consumed(3069123440):1
produced(3077516144):2
consumed(3069123440):2
produced(3077516144):3
consumed(3069123440):3
produced(3077516144):4
consumed(3069123440):4
produced(3077516144):5
consumed(3069123440):5
produced(3077516144):6
consumed(3069123440):6
produced(3077516144):7
consumed(3069123440):7
```

Summary

- Parallel programming concepts
- Multithreaded programming
- Pthreads
- Syncrhonization
- Mutex
- Condition variables

6.087 Practical Programming in C
January (IAP) 2010

6.087 Lecture 13 – January 28, 2010

- Review
- Multithreaded Programming
  - Race Conditions
  - Semaphores
  - Thread Safety, Deadlock, and Starvation
- Sockets and Asynchronous I/O
  - Sockets
  - Asynchronous I/O

Review: Multithreaded programming

- Thread: abstraction of parallel processing with shared memory
- Program organized to execute multiple threads in parallel
- Threads *spawned* by main thread, communicate via shared resources and *joining*
- `pthread` library implements multithreading

  - ```
    int pthread_create(pthread_t * thread, const pthread_attr_t * attr,
                       void *(*start_routine)(void *), void * arg);
    ```
  - `void pthread_exit(void *value_ptr);`
  - `int pthread_join(pthread_t thread, void **value_ptr);`
  - `pthread_t pthread_self(void);`
    ```

Review: Resource sharing

- Access to shared resources need to be controlled to ensure deterministic operation
- Synchronization objects: mutexes, semaphores, read/write locks, barriers
- Mutex: simple single lock/unlock mechanism
  - `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t * attr);`
  - `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

Review: Condition variables

- Lock/unlock (with mutex) based on run-time condition variable
- Allows thread to wait for condition to be true
- Other thread signals waiting thread(s), unblocking them
  - `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`
  - `int pthread_cond_destroy(pthread_cond_t *cond);`
  - `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - `int pthread_cond_signal(pthread_cond_t *cond);`

6.087 Lecture 13 – January 28, 2010

- Review
- Multithreaded Programming
  - Race Conditions
  - Semaphores
  - Thread Safety, Deadlock, and Starvation
- Sockets and Asynchronous I/O
  - Sockets
  - Asynchronous I/O

Multithreaded programming

- OS implements scheduler – determines which threads execute when
- Scheduling may execute threads in arbitrary order
- Without proper synchronization, code can execute non-deterministically
- Suppose we have two threads: 1 reads a variable, 2 modifies that variable
- Scheduler may execute 1, then 2, or 2 then 1
- Non-determinism creates a *race condition* – where the behavior/result depends on the order of execution

Race conditions

- Race conditions occur when multiple threads share a variable, without proper synchronization
- Synchronization uses special variables, like a mutex, to ensure order of execution is correct
- Example: thread T_1 needs to do something before thread T_2
    - condition variable forces thread T_2 to wait for thread T_1
    - producer-consumer model program
- Example: two threads both need to access a variable and modify it based on its value
    - surround access and modification with a mutex
    - mutex groups operations together to make them *atomic* – treated as one unit

Race conditions in assembly

Consider the following program `race.c`:

```
unsigned int cnt = 0;

void *count(void *arg) { /* thread body */
  int i;
  for (i = 0; i < 100000000; i++)
    cnt++;
  return NULL;
}

int main(void) {
  pthread_t tids[4];
  int i;
  for (i = 0; i < 4; i++)
    pthread_create(&tids[i], NULL, count, NULL);
  for (i = 0; i < 4; i++)
    pthread_join(tids[i], NULL);
  printf("cnt=%u\n",cnt);
  return 0;
}
```

What is the value of `cnt`?

[Bryant and O'Halloran. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.]

Race conditions in assembly

Ideally, should increment `cnt` 4×100000000 times, so `cnt` $= 400000000$. However, running our code gives:

```
athena%[1] ./race.o
cnt=137131900
athena% ./race.o
cnt=163688698
athena% ./race.o
cnt=163409296
athena% ./race.o
cnt=170865738
athena% ./race.o
cnt=169695163
```

So, what happened?

1 Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Race conditions in assembly

- C not designed for multithreading
- No notion of atomic operations in C
- Increment `cnt++;` maps to three assembly operations:
    1. load `cnt` into a register
    2. increment value in register
    3. save new register value as new `cnt`
- So what happens if thread interrupted in the middle?
- Race condition!

Race conditions in assembly

Let's fix our code:

```
pthread_mutex_t mutex;
unsigned int cnt = 0;

void *count(void *arg) { /* thread body */
  int i;
  for (i = 0; i < 100000000; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}

int main(void) {
  pthread_t tids[4];
  int i;
  pthread_mutex_init(&mutex, NULL);
  for (i = 0; i < 4; i++)
    pthread_create(&tids[i], NULL, count, NULL);
  for (i = 0; i < 4; i++)
    pthread_join(tids[i], NULL);
  pthread_mutex_destroy(&mutex);
  printf("cnt=%u\n",cnt);
  return 0;
}
```

Race conditions

- Note that new code functions correctly, but is much slower
- C statements not atomic – threads may be interrupted at assembly level, in the middle of a C statement
- Atomic operations like mutex locking must be specified as atomic using special assembly instructions
- Ensure that all statements accessing/modifying shared variables are synchronized

Semaphores

- *Semaphore* – special nonnegative integer variable `s`, initially 1, which implements two atomic operations:
  - `P(s)` – wait until $s > 0$, decrement s and return
  - `V(s)` – increment s by 1, unblocking a waiting thread
- Mutex – locking calls `P(s)` and unlocking calls `V(s)`
- Implemented in `<semaphore.h>`, part of library `rt`, not `pthread`

Using semaphores

- Initialize semaphore to `value`:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- Destroy semaphore:

```
int sem_destroy(sem_t *sem);
```

- Wait to lock, blocking:

```
int sem_wait(sem_t *sem);
```

- Try to lock, returning immediately (0 if now locked, -1 otherwise):

```
int sem_trywait(sem_t *sem);
```

- Increment semaphore, unblocking a waiting thread:

```
int sem_post(sem_t *sem);
```

Producer and consumer revisited

- Use a semaphore to track available slots in shared buffer
- Use a semaphore to track items in shared buffer
- Use a semaphore/mutex to make buffer operations synchronous

Producer and consumer revisited

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t mutex, slots, items;

#define SLOTS 2
#define ITEMS 10

void* produce(void* arg)
{
  int i;
  for (i = 0; i < ITEMS; i++)
  {
    sem_wait(&slots);
    sem_wait(&mutex);
    printf("produced(%ld):%d\n",
           pthread_self(), i+1);
    sem_post(&mutex);
    sem_post(&items);
  }
  return NULL;
}

void* consume(void* arg)
{
  int i;
  for (i = 0; i < ITEMS; i++) {
    sem_wait(&items);
    sem_wait(&mutex);
    printf("consumed(%ld):%d\n",
           pthread_self(), i+1);
    sem_post(&mutex);
    sem_post(&slots);
  }
  return NULL;
}

int main()
{
  pthread_t tcons, tpro;

  sem_init(&mutex, 0, 1);
  sem_init(&slots, 0, SLOTS);
  sem_init(&items, 0, 0);

  pthread_create(&tcons,NULL,consume,NULL);
  pthread_create(&tpro,NULL,produce,NULL);
  pthread_join(tcons,NULL);
  pthread_join(tpro,NULL);

  sem_destroy(&mutex);
  sem_destroy(&slots);
  sem_destroy(&items);
  return 0;
}
```

[Bryant and O'Halloran. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.]

Other challenges

- Synchronization objects help solve race conditions
- Improper use can cause other problems
- Some common issues:
  - thread safety and reentrant functions
  - deadlock
  - starvation

Thread safety

- Function is *thread safe* if it always behaves correctly when called from multiple concurrent threads
- Unsafe functions fal in several categories:
  - accesses/modifies unsynchronized shared variables
  - functions that maintain state using static variables – like `rand()`, `strtok()`
  - functions that return pointers to static memory – like `gethostbyname()`
  - functions that call unsafe functions may be unsafe

Reentrant functions

- Reentrant function – does not reference any shared data when used by multiple threads
- All reentrant functions are thread-safe (are all thread-safe functions reentrant?)
- Reentrant versions of many unsafe C standard library functions exist:

Unsafe function	Reentrant version
`rand()`	`rand_r()`
`strtok()`	`strtok_r()`
`asctime()`	`asctime_r()`
`ctime()`	`ctime_r()`
`gethostbyaddr()`	`gethostbyaddr_r()`
`gethostbyname()`	`gethostbyname_r()`
`inet_ntoa()`	(none)
`localtime()`	`localtime_r()`

Thread safety

To make your code thread-safe:

- Use synchronization objects around shared variables
- Use reentrant functions
- Use synchronization around functions returning pointers to shared memory (*lock-and-copy*):
    1. lock mutex for function
    2. call unsafe function
    3. dynamically allocate memory for result; (deep) copy result into new memory
    4. unlock mutex

Deadlock

- Deadlock – happens when every thread is waiting on another thread to unblock
- Usually caused by improper ordering of synchronization objects
- Tricky bug to locate and reproduce, since schedule-dependent
- Can visualize using a progress graph – traces progress of threads in terms of synchronization objects

Deadlock

Figure removed due to copyright restrictions. Please see
http://csapp.cs.cmu.edu/public/1e/public/figures.html,
Figure 13.39, Progress graph for a program that can deadlock.

Deadlock

- Defeating deadlock extremely difficult in general
- When using only mutexes, can use the “mutex lock ordering rule” to avoid deadlock scenarios:
  A program is deadlock-free if, for each pair of mutexes (s, t) in the program, each thread that uses both s and t simultaneously locks them in the same order.

[Bryant and O’Halloran. *Computer Systems: A Programmer’s Perspective* Prentice Hall, 2003.]

Starvation and priority inversion

- Starvation similar to deadlock
- Scheduler never allocates resources (e.g. CPU time) for a thread to complete its task
- Happens during priority inversion
  - example: highest priority thread T_1 waiting for low priority thread T_2 to finish using a resource, while thread T_3, which has higher priority than T_2, is allowed to run indefinitely
  - thread T_1 is considered to be in starvation

6.087 Lecture 13 – January 28, 2010

- Review
- Multithreaded Programming
  - Race Conditions
  - Semaphores
  - Thread Safety, Deadlock, and Starvation
- Sockets and Asynchronous I/O
  - Sockets
  - Asynchronous I/O

Sockets

- *Socket* – abstraction to enable communication across a network in a manner similar to file I/O
- Uses header `<sys/socket.h>` (extension of C standard library)
- Network I/O, due to latency, usually implemented asynchronously, using multithreading
- Sockets use client/server model of establishing connections

Creating a socket

- Create a socket, getting the file descriptor for that socket:

```c
int socket(int domain, int type, int protocol);
```

  - `domain` – use constant `AF_INET`, so we're using the internet; might also use `AF_INET6` for IPv6 addresses
  - `type` – use constant `SOCK_STREAM` for connection-based protocols like TCP/IP; use `SOCK_DGRAM` for connectionless datagram protocols like UDP (we'll concentrate on the former)
  - `protocol` – specify 0 to use default protocol for the socket type (e.g. TCP)
  - returns nonnegative integer for file descriptor, or -1 if couldn't create socket
- Don't forget to close the file descriptor when you're done!

Connecting to a server

- Using created socket, we connect to server using:

```
int connect(int fd, struct sockaddr *addr, int addr_len);
```

  - `fd` – the socket's file descriptor
  - `addr` – the address and port of the server to connect to; for internet addresses, cast data of type `struct sockaddr_in`, which has the following members:
    - `sin_family` – address family; always `AF_INET`
    - `sin_port` – port in network byte order (use `htons()` to convert to network byte order)
    - `sin_addr.s_addr` – IP address in network byte order (use `htonl()` to convert to network byte order)
  - `addr_len` – size of `sockaddr_in` structure
  - returns 0 if successful

Associate server socket with a port

- Using created socket, we bind to the port using:

```c
int bind(int fd, struct sockaddr *addr, int addr_len);
```

  - `fd`, `addr`, `addr_len` – same as for `connect()`
  - note that address should be IP address of desired interface (e.g. `eth0`) on local machine
  - ensure that port for server is not taken (or you may get “address already in use” errors)
  - return 0 if socket successfully bound to port

Listing for clients

- Using the bound socket, start listening:

```c
int listen(int fd, int backlog);
```

  - `fd` – bound socket file descriptor
  - `backlog` – length of queue for pending TCP/IP connections; normally set to a large number, like 1024
  - returns 0 if successful

Accepting a client's connection

- Wait for a client's connection request (may already be queued):

```
int accept(int fd, struct sockaddr *addr, int *addr_len);
```

  - `fd` – socket's file descriptor
  - `addr` – pointer to structure to be filled with client address info (can be NULL)
  - `addr_len` – pointer to int that specifies length of structure pointed to by `addr`; on output, specifies the length of the stored address (stored address may be truncated if bigger than supplied structure)
  - returns (nonnegative) file descriptor for connected client socket if successful

Reading and writing with sockets

- Send data using the following functions:

```
int write(int fd, const void *buf, size_t len);

int send(int fd, const void *buf, size_t len, int flags);
```

- Receive data using the following functions:

```
int read(int fd, void *buf, size_t len);

int recv(int fd, void *buf, size_t len, int flags);
```

  - `fd` – socket's file descriptor
  - `buf` – buffer of data to read or write
  - `len` – length of buffer in bytes
  - `flags` – special flags; we'll just use 0
  - all these return the number of bytes read/written (if successful)

Asynchronous I/O

- Up to now, all I/O has been synchronous – functions do not return until operation has been performed
- Multithreading allows us to read/write a file or socket without blocking our main program code (just put I/O functions in a separate thread)
- Multiplexed I/O – use `select()` or `poll()` with multiple file descriptors

I/O multiplexing with `select()`

- To check if multiple files/sockets have data to read/write/etc: (include `<sys/select.h>`)

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout);
```

  - `nfds` – specifies the total range of file descriptors to be tested (0 up to `nfds`-1)
  - `readfds`, `writefds`, `errorfds` – if not NULL, pointer to set of file descriptors to be tested for being ready to read, write, or having an error; on output, set will contain a list of only those file descriptors that are ready
  - `timeout` – if no file descriptors are ready immediately, maximum time to wait for a file descriptor to be ready
  - returns the total number of set file descriptor bits in all the sets
- Note that `select()` is a blocking function

I/O multiplexing with `select()`

- `fd_set` – a mask for file descriptors; bits are set (“1”) if in the set, or unset (“0”) otherwise
- Use the following functions to set up the structure:
  - FD_ZERO(&fdset) – initialize the set to have bits unset for all file descriptors
  - FD_SET(fd, &fdset) – set the bit for file descriptor `fd` in the set
  - FD_CLR(fd, &fdset) – clear the bit for file descriptor `fd` in the set
  - FD_ISSET(fd, &fdset) – returns nonzero if bit for file descriptor `fd` is set in the set

I/O multiplexing using `poll()`

- Similar to `select()`, but specifies file descriptors differently: (include `<poll.h>`)

```
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

  - `fds` – an array of `pollfd` structures, whose members `fd`, `events`, and `revents`, are the file descriptor, events to check (OR-ed combination of flags like `POLLIN`, `POLLOUT`, `POLLERR`, `POLLHUP`), and result of polling with that file descriptor for those events, respectively
  - `nfds` – number of structures in the array
  - `timeout` – number of milliseconds to wait; use 0 to return immediately, or -1 to block indefinitely

Summary

- Multithreaded programming
  - race conditions
  - semaphores
  - thread safety
  - deadlock and starvation
- Sockets, asynchronous I/O
  - client/server socket functions
  - `select()` and `poll()`

Outline

- Review
- Inter process communication
  - Signals
  - Fork
  - Pipes
  - FIFO
- Spotlights

6.087 Lecture 14 – January 29, 2010

- Review
- Inter process communication
  - Signals
  - Fork
  - Pipes
  - FIFO
- Spotlights

Review: multithreading

- Race conditions
  - non-determinism in thread order.
  - can be prevented by synchronization
  - atomic operations necessary for synchronization
- Mutex: Allows a single thread to own it
- Semaphores: Generalization of mutex, allows N threads to acquire it at a time.
  - P(s) : acquires a lock
  - V(s) : releases lock
  - `sem_init()`,`sem_destroy()`
  - `sem_wait()`,`sem_trywait()`,`sem_post()`
- Other problems: deadlock, starvation

Sockets

- <sys/socket.h>
- enables client-server computing
- Client: `connect()`
- Server: `bind()`, `listen()`, `accept()`
- I/O: `write()`, `send()`, `read()`, `recv()`

6.087 Lecture 14 – January 29, 2010

- Review
- Inter process communication
  - Signals
  - Fork
  - Pipes
  - FIFO
- Spotlights

Preliminaries

- Each process has its own address space. Therefore, individual processes cannot communicate unlike threads.
- Interprocess communication: Linux/Unix provides several ways to allow communications
  - **signal**
  - **pipes**
  - **FIFO queues**
  - shared memory
  - semaphores
  - **sockets**

<signals.h>

- Unix/Linux allows us to handle exceptions that arise during execution (*e.g.,* interrupt, floating point error, segmentation fault etc.).
- A process recieves a *signal* when such a condition occurs.

```
void (*signal(int sig,void(*handler)(int)))(int)
```

- determines how subsequent signals will be handled.
- pre-defined behavior: SIG_DFL (default), SIG_IGN (ignore)
- returns the previous handler.

<signal.h>

Valid signals:

SIGABRT	abnormal termination
SIGFPE	floating point error
SIGILL	illegal instruction
SIGINT	interrupt
SIGSEGV	segmentation fault
SIGTERM	termination request
SIGBUS	bus error
SIGQUIT	quit

The two signals SIGSTOP,SIGKILL cannot be handled.

<signal.h>

`int raise(int sig)` can be used to send signal `sig` to the program.

Notes:

- There can be race conditions.
- signal handler itself can be interrupted.
- use of non-reentrant functions unsafe.
- `sigprocmask` can be used to prevent interruptions.
- handler is **reset** each time it is called.

Example

```
#include <stdio.h>

void sigproc()
{       signal(SIGINT, sigproc); /*   */
     printf("you have pressed ctrl-c \n");
}

void quitproc()
{       printf("ctrl-\\ pressed to quit");
     exit(0); /* normal exit status */
}
main()
{
     signal(SIGINT, sigproc);
     signal(SIGQUIT, quitproc);
     printf(``ctrl-c disabled use ctrl-\\ to quitn'');
     for(;;); /* infinite loop */
}
```

Fork

```
pid_t fork(void)
```

- `fork()` is a system call to create a new **process**
- In the child process, it returns 0
- In the parent process, it returns the PID (process id) of the child.
- The child PID can be used to send signals to the child process.
- returns -1 on failure (invalid PID)

Example

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

int main() {
        /*some code*/
        pid_t pid=fork();
    int i;
        if(pid) {
        for(i=0;i<5;i++){
            sleep(2);
                    printf("parent process:%d\n",i);
        }
        }
        else    {
        for(i=0;i<5;i++){
            sleep(1);
                    printf("child process:%d\n",i);
        }

        }/*end child*/
}/*end main*/


parent process:0
child process:1
child process:2
parent process:1
child process:3
child process:4
parent process:2
parent process:3
parent process:4
```

Fork

- `fork()` makes a full copy of the parents address space.
- `pid_t getpid()` returns PID of the current process.
- `pid_t getppid()` returns PID of the parent process.
- `wait(int*)` is used to wait for the child to finish.
- `waitpid()` is used to wait for a specific child.

Zombies:

- the child process can exit before the parent
- stray process is marked as <defunct>
- `preap` can be used to reap zombie processes.

Pipes

Pipes are used in unix to redirect output of one command to another. Pipes also allow parent processes to communicate with its children. Examples

- `ls | more` - displays results of ls one screen at a time
- `cat file.txt | sort` -displays contents of file.txt in sorted order

```
int pipe(int FILEDES[2])
```

- A pipe can be thought of as a pair of file descriptors
- no physical file is associated with the file descriptor
- one end is opened in write mode.
- other end is opened in read mode.

Example

```c
/* source: http://beej.us/guide */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h> /*ipc*/

int main(void)
{
    int pfds[2];
    char buf[30];
    pipe(pfds);
    if (!fork()) {
        printf(" CHILD: writing to the pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }
    return 0;
}
```

FIFO

- FIFO queues may be thought of as named pipes.
- Multiple processes can read and write from a FIFO.
- Unlike pipes, the processes can be unrelated.
- FIFOs can be created using `mknod` system call.

```
int mknod (const char *path,mode_t mode,dev_t dev)
```

- `<sys/stat.h>` contains the declaration for mknod.
- `mknod` used to create *special* files - devices,fifos etc.
- mode can have special bits such as `S_IFIFO | 0644`
- dev is interpreted based on the mode.

Example: `mknod("myfifo", S_IFIFO | 0644 , 0);`

Example

```c
/* source: http://beej.us/guide */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "fifo"

int main(void) {
  char s[300];
  int num, fd;
  mknod(FIFO_NAME, S_IFIFO | 0666, 0);
  printf("waiting for readers...\n");
  fd = open(FIFO_NAME, O_WRONLY);
  printf("got a reader\n");

  while (gets(s), !feof(stdin)) {
   num = write(fd, s, strlen(s));
   if (num == -1)
        perror("write");
   else
        printf("wrote %d bytes\n", num);
   }
   return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#define FIFO_NAME "fifo"
int main(void) {
    char s[300];
    int num, fd;
    mknod(FIFO_NAME, S_IFIFO | 0666, 0);
    printf("waiting for writers...\n");
    fd = open(FIFO_NAME, O_RDONLY);
    printf("got a writer\n");

    do {
      num = read(fd, s, 300);
        if (num == -1)
            perror("read");
        else {
            s[num] = '\0';
            printf("read %d bytes:\"%s\"\n",
                num, s);
        }
    } while (num > 0);
    return 0;
}
```

6.087 Lecture 14 – January 29, 2010

- Review
- Inter process communication
  - Signals
  - Fork
  - Pipes
  - FIFO
- Spotlights

Project spotlights

- Face finding with openCV
- Barcode scanner
- ImageIC
- Image2DXF
- Library database
- Simple Audio Visualizer
- Non-linear oscillator
- NoteDeluxe
- CUDA
- Visual mouse
- Wallpaper downloader