CSC 373 Winter 2020 Prof Lytinen
Intro to Linux and C

- **I'll be using:** `cdmlinux.cdm.depaul.edu`. This is a 64-bit Linux machine including a 64-bit gcc. You have a free account on it, and it provides the Linux/64-bit environment which you can use to complete all of the assignments.

- **Connect to your account using a PC:**

  SSH (Secure Shell): recommended
  http://my.cdm.depaul.edu/resources/software/SSHSecureShellClient-3.2.9.zip

  or

  PuTTY and related tools
  http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Using a Mac for this course

- The Mac OS is a variant of Linux. For the first part of the course, you can simply use the Mac Terminal.

- However, later in the quarter you may need to use `cdmlinux.cdm.depaul.edu`

- Mac terminal has a command called `ssh`. To use:

```
ssh slytinen@cdmlinux.cdm.depaul.edu (but use
your campusconnect login)
```

- You will then be prompted for your password.

- Mac ssh is terminal-based.

Transferring files from a Mac to cdmlinux

- Mac terminal has a command called `sftp`. Be sure to **connect to the correct working directory** on your Mac before running `sftp`. Then:

```
sftp slytinen@cdmlinux.cdm.depaul.edu (but
your campusconnect login)
```

- You will then be prompted for your password.
- To change working directories on cdmlinux, type (assuming the directory already exists)

```
cd 373w20
```

- `put` transfers a file from your Mac to cdmlinux, `get` does the opposite.

Linux manual

- See http://www.linuxcommand.org. In particular, I recommend
  - Learning the Shell (http://www.linuxcommand.org/lc3_learning_the_shell.php)
  - Looking around (http://www.linuxcommand.org/lc3_lts0030.php)
  - Manipulating files (http://www.linuxcommand.org/lc3_lts0050.php)
  - Permissions (http://www.linuxcommand.org/lc3_lts0090.php)

Compiling C programs

To compile a single file C program in a file named `hello.c` use the gcc compiler.

```
// hello.c
#include <stdio.h>

int main() {
  printf("Hello\n");
}
$ gcc hello.c -o hello
```

This compiles the program *hello.c*, links it with any standard c library routines called (e.g. i/o routines) and produces the executable file named *hello*.   Now run the executable:

```
$ ./hello
```

Linux online Manual

- Linux `man` command
  - To view the Linux online manual, you need to know the exact name of a manual topic; e.g., `ls`.
  - In some cases you may not know the exact name.
  - You can try the man command with the `-k` option (for keyword)

```
$ man ls
```

Linux directory structure

- Many Linux machine are multi-user
- Each user gets their own **root** directory

  pwd

  /home/DPU/slytinen
- Each process has its own current **working directory** (pwd)

- Each directory can contain a mixture of files and other directories
- Therefore, the directories themselves form a tree structure

Basic Linux commands

- pwd: print working directory
- ls: list file/directories in a directory
- cd: change directories
- mkdir: create a new directory

```
[slytinen@cdmlinux ~]$ mkdir 373w20
[slytinen@cdmlinux ~]$ cd 373w20
[slytinen@cdmlinux 373w20]$
[slytinen@cdmlinux 373w20]$ ls
[slytinen@cdmlinux 373w20]$
[slytinen@cdmlinux 373w20]$ cd ..
[slytinen@cdmlinux]$
```

- cp: copy a file
- cp –r: copy a directory and all of its files/subdirectories

Basic Linux commands

- rm: remove a file
- rm –r: remove a directory and all of its files/subdirectories
- mv : move a file
- mv –r: move a directory and all of its file/subdirectories
- gcc: run C compiler
- rmdir: remove a directory

Wild card *

matches strings/substrings in file or folder names

```
        rm *     ← don't do this!!
```

Linux command flags

- Many Linux commands can be passed "flags" which refine how they behave
- Most flags start with -
- Examples:
- `rm`

  `-i` flag: system asks to confirm removal

  `-r` flag: remove a directory and its contents

- `gcc` flags:

  -m specifies target machine (32 vs. 64 bit – this does not work on `cdmlinux`)

  -o specifies output file name

  `-O` specifies level of optimization (`-O1, -O2, -O3`)

Directory structure

- Delimiter in path names is / (not \)
- Can use * as a wild card to refer to directories or files

```
$ ls hello*.c
hello2.c  hello.c
```

- .. means "one level up" in directory structure

```
[slytinen@cdmlinux 373w20]$ cd ..
[slytinen@cdmlinux ~]$
```

- `Relative vs. absolute paths`

```
[slytinen@cdmlinux ~]$ cd 373w20
   same as
[slytinen@cdmlinux ~]$ cd ~slytinen/373w20
[slytinen@cdmlinux ~]$ cd ~/373w20
```

Directory structure

- / means the root directory (it would be unusual for you to want to look at the root directory on `cdmlinux`)

```
# cd /
$ ls
bin   dev  home  lib64  mnt  proc  run
srv  tmp  var boot  etc  lib   media
opt  root  sbin  sys  usr  vmkeys
```

- ~ (or ~/) means the root of your directory structure

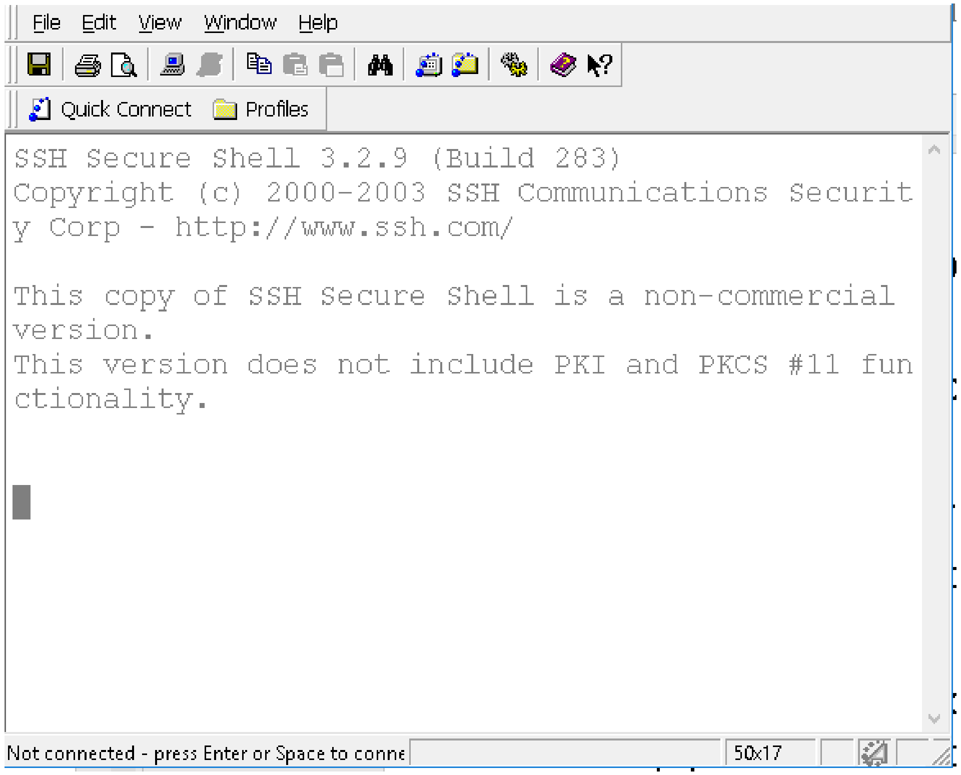
```
[slytinen@cdmlinux 373w20]$ cd ~
[slytinen@cdmlinux ~]$
```

Downloading code from D2L

- Depends on your laptop and the tools you have.
- Assume PC, graphic ssh.
- Save the .tar file to your PC/Mac
- On PC, start ssh, login to cdmlinux

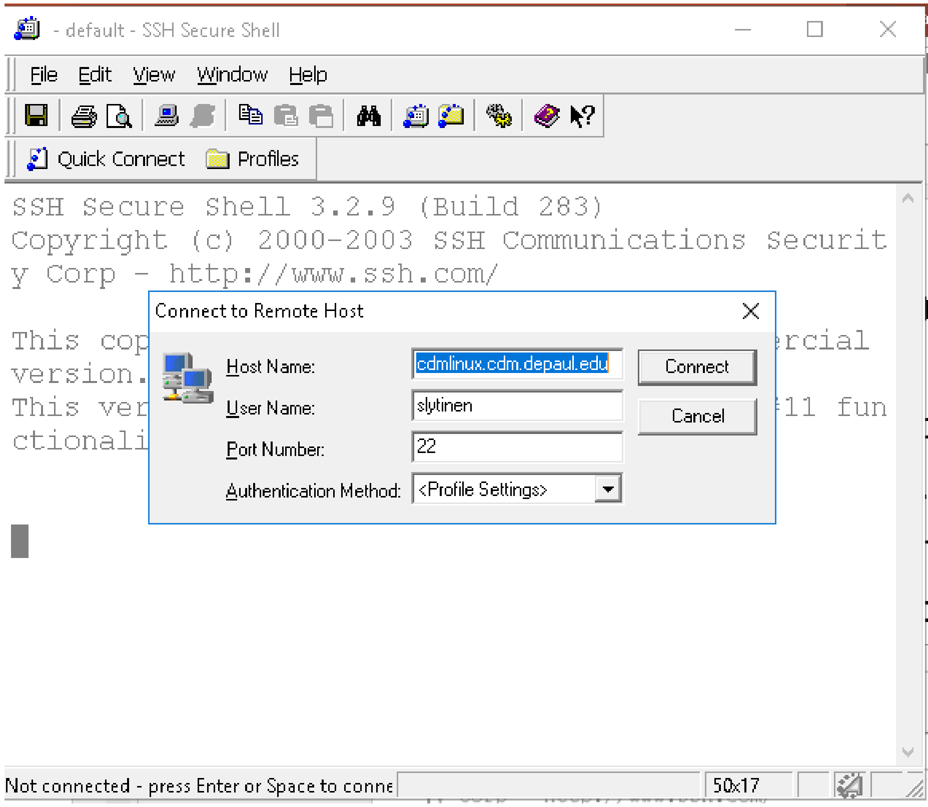
- Navigate (using ssh window, right ride) to the directory you want to store the tar file in
- Under the cdmlinux shell

```
    $ tar xvf *.tar
```



```
SSH Secure Shell 3.2.9 (Build 283)
Copyright (c) 2000-2003 SSH Communications Securit
y Corp - http://www.ssh.com/

This copy of SSH Secure Shell is a non-commercial
version.
This version does not include PKI and PKCS #11 fun
ctionality.
```

Not connected - press Enter or Space to conne 50x17

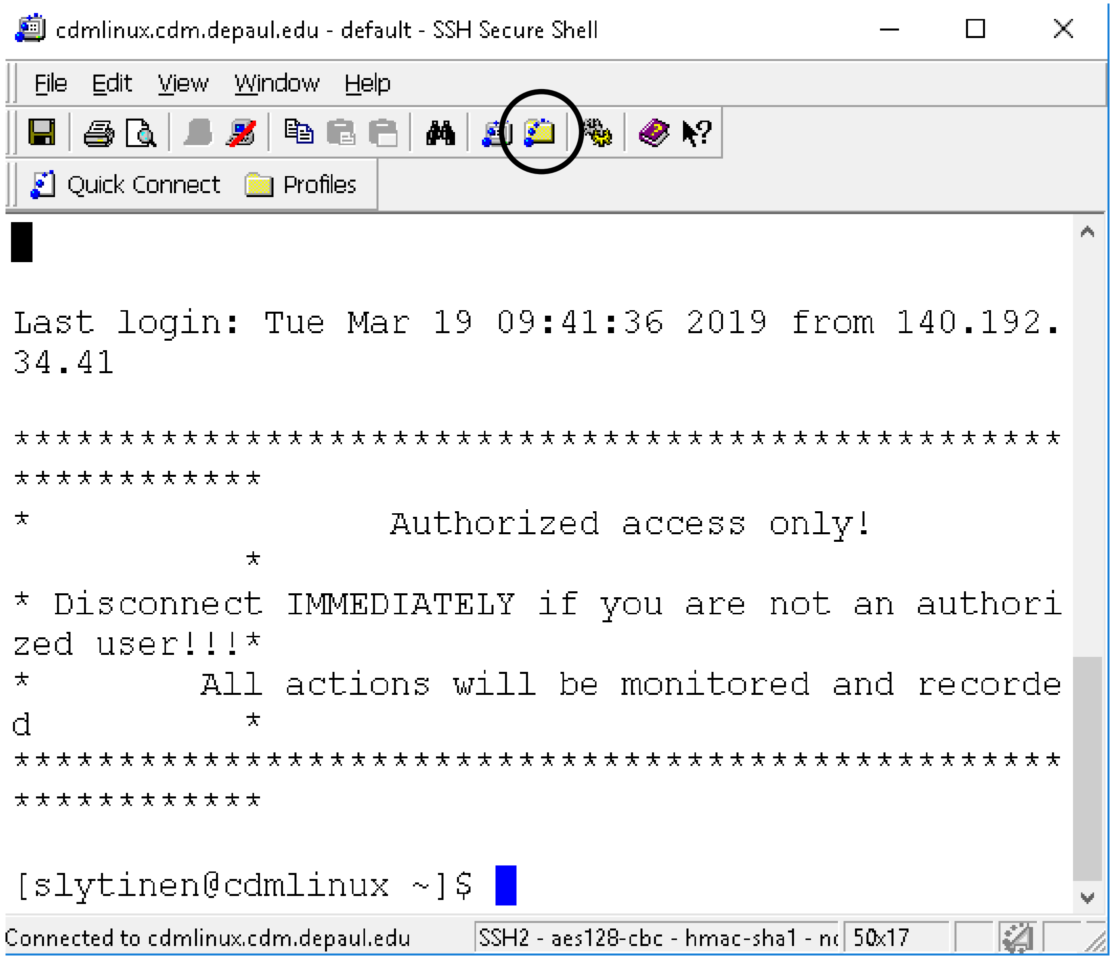


```
SSH Secure Shell 3.2.9 (Build 283)
Copyright (c) 2000-2003 SSH Communications Securit
y Corp - http://www.ssh.com/
```

Host Name: cdmlinux.cdm.depaul.edu
User Name: slytinen
Port Number: 22
Authentication Method: <Profile Settings>

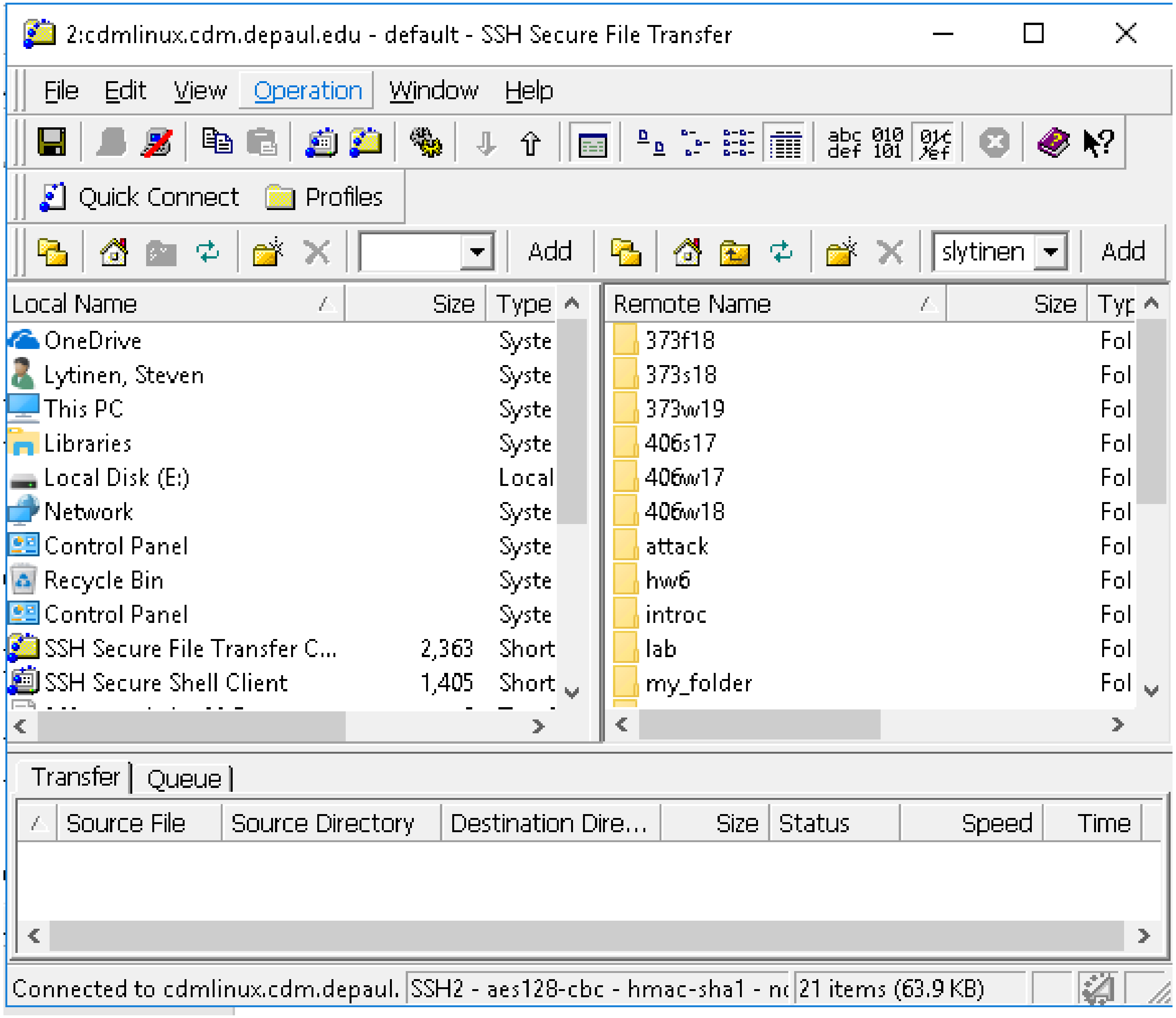
Not connected - press Enter or Space to conne 50x17

Hit Space key to be prompted for ssh. Type cdmlinux.cdm.depaul.edu,
And your user name. You'll then be prompted for your password.


cdmlinux.cdm.depaul.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect
Profiles
Connected to cdmlinux.cdm.depaul.edu
SSH2 - aes128-cbc - hmac-sha1 - nc
50x17

```
Last login: Tue Mar 19 09:41:36 2019 from 140.192.
34.41

**************************************************
************
*                 Authorized access only!
           *
* Disconnect IMMEDIATELY if you are not an authori
zed user!!!*
*        All actions will be monitored and recorde
d          *
**************************************************
************

[slytinen@cdmlinux ~]$
```

Click yellow folder icon for a FTP GUI


2:cdmlinux.cdm.depaul.edu - default - SSH Secure File Transfer
File Edit View Operation Window Help
Quick Connect
Profiles
Add
slytinen
Add
Transfer
Queue

Local Name	Size	Type
OneDrive		Syste
Lytinen, Steven		Syste
This PC		Syste
Libraries		Syste
Local Disk (E:)		Local
Network		Syste
Control Panel		Syste
Recycle Bin		Syste
Control Panel		Syste
SSH Secure File Transfer C...	2,363	Short
SSH Secure Shell Client	1,405	Short

Remote Name	Size	Typ
373f18		Fol
373s18		Fol
373w19		Fol
406s17		Fol
406w17		Fol
406w18		Fol
attack		Fol
hw6		Fol
introc		Fol
lab		Fol
my_folder		Fol

Source File	Source Directory	Destination Dire...	Size	Status	Speed	Time

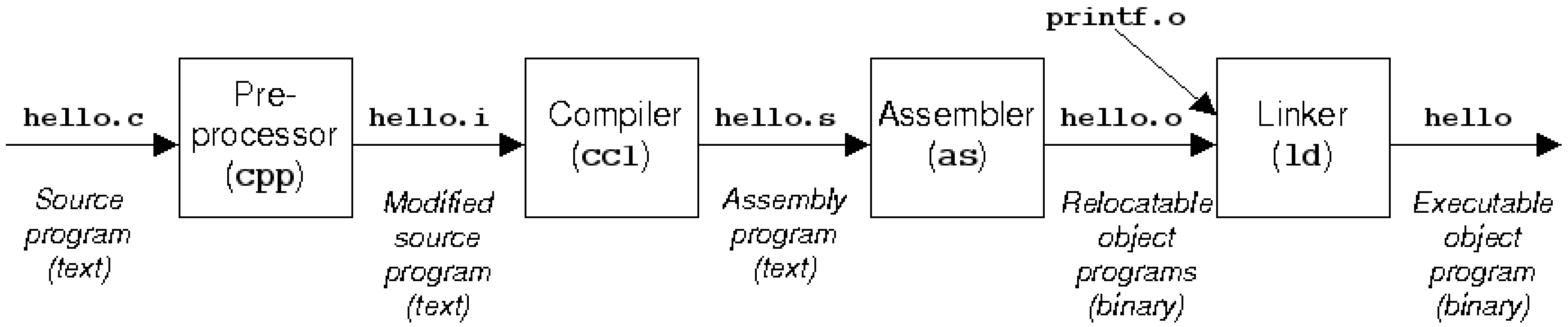
Connected to cdmlinux.cdm.depaul. | SSH2 - aes128-cbc - hmac-sha1 - nc | 21 items (63.9 KB)

Intro to C

- The obligatory "Hello world" program

```
/* Simple example program */
#include <stdio.h>

int main() {
    printf("Hello world\n");
}
```

- `#include` is a preprocessor directive; in essence, it pastes the C code in the header file into your own C code prior to compilation. Most of this code is either variable definitions or function prototypes.



Intro to C

- The obligatory “Hello world” program

```
/* Simple example program */
#include <stdio.h>

int main() {
   printf("Hello world\n");
}
```

- `main` is a function. It follows the function definition syntax:

```
<return-type> <name>(<parameters>) {
   <body>
}
```

- Execution of a C program always starts with `main`.

- printf is a built-in function in the <stdio.h> library.

One-pass compilers

- Older C compilers are “one-pass”, meaning that a function must be defined before it can be used (this is NOT the case with `gcc 4.8.5` on cdmlinux.cdm.depaul.edu)

- Example:

```
int main() {
    int y = 2;
    int s = square(y); // compiler will complain
    printf("%d\n", s); }

// To avoid this error, the definition of square must
// appear before the definition of main
int square(int x) {
    x *= x; // now x is 4
    return x;
}
```

One-pass compilers

Or, a “prototype” for square may be given prior to the definition of main

- Example of a prototype

```
int square(int);

int main() {
    int y = 2;
    int s = square(y); // compiler will complain
    printf("%d\n", s); }

int square(int x) {
    x *= x; // now x is 4
    return x;
}
```

Parameter passing

- In C, parameters are passed “by value”. This means that, for example, if main passes a parameter to square, the variable in main cannot be changed.

```
#include <stdio.h>

int main() {
  int x;
  printf("Type an integer\n");
  scanf("%d", &x);
  int sq =  square(x);
  printf("%d squared is %d\n", x, sq);
}

int square(int y) {
  y *= y;     // changes y, but not x in main
  return y;
}
```

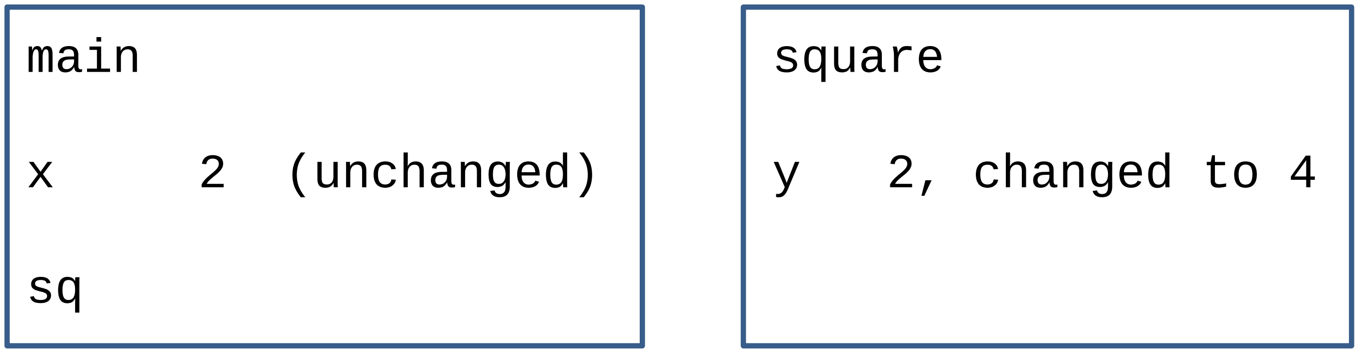
Parameter passing

- In C, parameters are passed “by value”. This means that, for example, if main passes a parameter to square, the variable in main cannot be changed.

```
#include <stdio.h> // square2.c

int main() {
  int x;
  printf("Type an integer\n");
  scanf("%d", &x);
  int sq =  square2(x);
  printf("%d squared is %d\n", x, sq);
}

int square2(int x) {
  x *= x;  // does not matter that the variable name in main
           // is the same as in square2
  return x;
}
```

```
int main() {
  int x;
  printf("Type an integer\n");
  scanf("%d", &x);
  int sq =  square(x);
  printf("%d squared is %d\n", x, sq);
}

int square(int y) {
  y *= y;     // changes y, but not x in main
  return y;
}
```

During execution of “square”



```
int main() {
  int x;
  printf("Type an integer\n");
  scanf("%d", &x);
  int sq =  square(x);
  printf("%d squared is %d\n", x, sq);
}

int square(int y) {
  y *= y;     // changes y, but not x in main
  return y;
}
```

After execution of "square"

```
main

x      2  (unchanged)
sq     4
```

Note that in some languages (e.g., Python and Java), some types of parameters might be modified when passed to a function as a parameter. This can happen in C, but only if a parameter is a **pointer** (to be discussed later)

```
int main() {
  int x;
  printf("Type an integer\n");
  scanf("%d", &x);
  int sq =  square2(x);
  printf("%d squared is %d\n", x, sq);
}

int square2(int x) {
  x *= x;     // does not change x in main
  return x;
}
```

During execution of “square”

```
main

x      2   (unchanged)

sq
```

```
square

x    2, changed to 4
```

```c
int main() {
  int x;
  printf("Type an integer\n");
  scanf("%d", &x);
  int sq =  square2(x);
  printf("%d squared is %d\n", x, sq);
}

int square2(int x) {
  x *= x;     // does not change x in main
  return x;
}
```

After execution of “square2”

```
main

x      2  (unchanged)

sq     4
```

- **C spacing**

- Spaces, line breaks, indentation, etc. make no difference to the C compiler, with the exception of comments

```
int

main(       )    { printf
(
"Hello, world\n"
      )
;          }
```

- **Variable declarations**

- Before a variable can be used in C, it must be **declared**

- Syntax of a declaration:

```
<type> <name>;
```

- For example

```
int x; // x can store any value of type int
```

- May initialize the variable while declaring it:

```
int x = 3;
```

- **Declarations are generally not necessary in Python**

C data types

types in C:

- Numbers

  ```
  int, long, short, byte
  unsigned int, unsigned long, …
  float, double
  ```

- `char` (individual character)
- void
- **pointers** (exist in other languages, but are not as explicit)
- structures, which we may discuss at the end of the quarter if there is time
- **There are no classes or objects in the C language.**

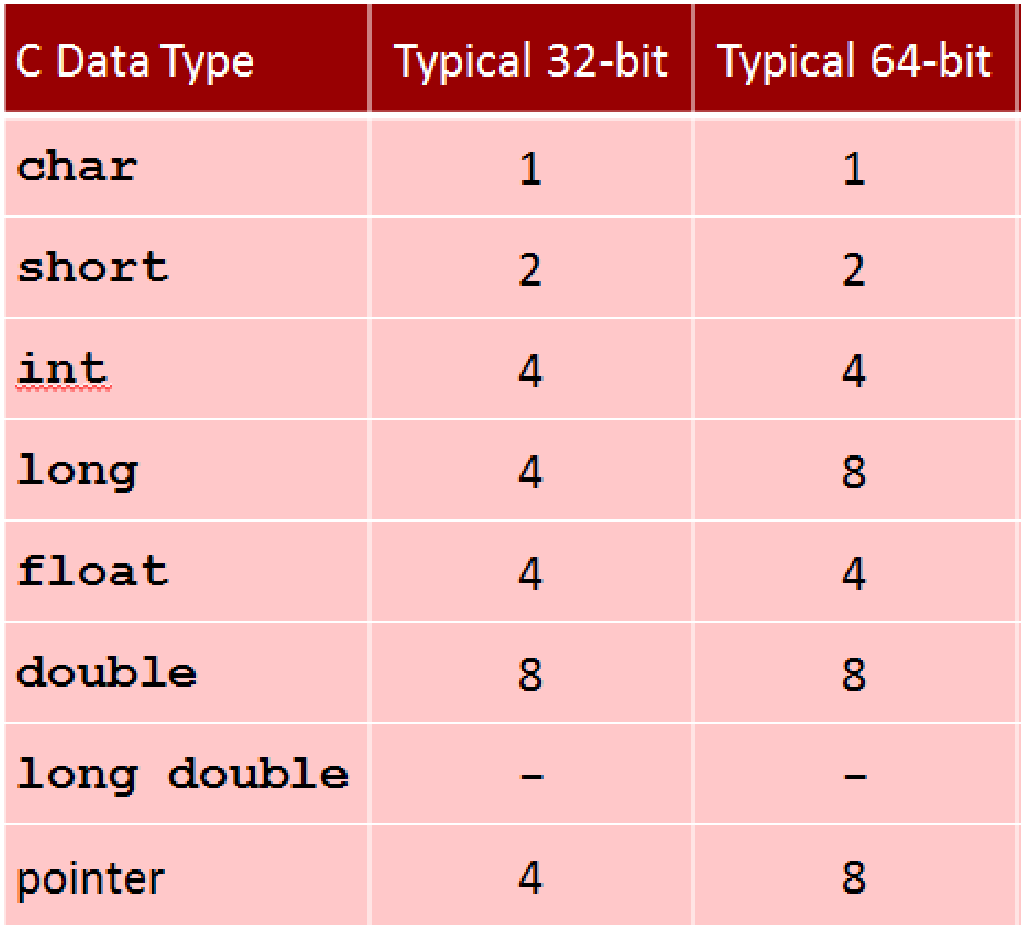
Illustration of Integer data types

```
// numbers.c
#include <stdio.h>
int main() {

  int i=1, j;
  long m=1, n;
  short s=1, t;
  char c=1, d;

  while (i > 0) {
    j = i;
    i *= 2; }

  printf("Largest int multiple of 2: %d\n", j);
  printf("%d * 2: %d\n\n", j, i);
  while (m > 0) {
    n = m;
    m *= 2;   }
  printf("Largest long multiple of 2: %ld\n", n);
  printf("%ld * 2: %ld\n\n", n, m);

  while (s > 0) {
    t = s;
    s *= 2; }
  printf("Largest short multiple of 2: %d\n", t);
  printf("%d * 2: %d\n\n", t, s);

  while (c > 0) {
    d = c;
    c *= 2;
  }
  printf("Largest char multiple of 2: %d\n", d);
  printf("%d * 2: %d\n", d, c);
}
```

```
[slytinen@cdmlinux 373s18]$ ./numbers

Largest int multiple of 2: 1073741824  (2^30)
1073741824 * 2: -2147483648

Largest long multiple of 2: 4611686018427387904   (2^62)
4611686018427387904 * 2: -9223372036854775808

Largest short multiple of 2: 16384  (2^14)
16384 * 2: -32768

Largest char multiple of 2: 64      (2^6)
64 * 2: -128
```

SIzes

Number of bytes (1 byte = 8 bits) required for each type of data



C Data Type	Typical 32-bit	Typical 64-bit
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8
`float`	4	4
`double`	8	8
`long double`	–	–
pointer	4	8

Integer representations

- Except for unsigned variables, the leftmost bit indicates the sign.
- Non-negative integers: leftmost bit = 0,
  the remaining bits represent the magnitude of the integer (in binary)
- Negative integers are represented using **2s complement**
  leftmostbit = 1; remaining bits to be discussed later
- Examples (positive integers):

Number	Binary
int x = 3;	00000000 00000000 00000000 00000011
long y = 10;	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001010

Integer maximum values

- int (32-bit): 2^{31} - 1: 2147483647. Why?
- long 64-bit: 2^{63} - 1: = 9223372036854775807
- Negative integers to be discussed later

- **See largest.c, unsigned_largest.c**

```
// compute largest int j
int i=1, j;
while (i > 0) {
    j = i;
    i = i*2 + 1;
  }
```

```
// compute largest unsigned int j
unsigned int i=1, j=0;
while (i > j) {
    j = i;
    i = i*2 + 1;
  }
```

Comments

- // means comment till the end of the line
- Example:

```
int main() {
    int x; // x is declared
    int y; // y is declared
    …
}
```

- `/*...*/` means comment appears between `/* and */`, and the comment can include line breaks

```
int main() {
    int x; /* x is declared
    int y; y is not declared */
    …
}
```

Arithmetic operations

`* / - + %  += -=  ...  ++ --`

- Standard precedence

```
int x = 1 + 2 * 3;   // x = 7, not 9
```

- `x++` means increment x by 1
- So does `++x`
- Difference

```
int x = 0;
printf("%d\n", x++); // prints 0
printf("%d\n", x);   // prints 1
printf("%d\n", ++x); // prints 2
printf("%d\n", x);   // prints 2
```

Mixing types

- Generally end up with the more "general" type. For example:

Type 1	Type 2	Result
Int	Long	Long
Int	Double	Double
Float	Double	Double

- Also may mix types in assignment statements; value "adapts" to the variable type

- Example:

```
int x;
double y;
x = 2;
y = x; // y is 2.0
```

Mixing types mix.c

- Another example:

```
int x;
x = 9.9; // x is 9
```

- Operator precedence may determine when a conversion (cast) takes place
- Any difference?

```
int x;
x = (9/2.0) * 2;
x = (9/2) * 2;
```

Mixing types

- Carelessly mixing types can sometimes lead to unexpected results

```
// test.c
int main() {
   int x = 1;
   printf("%f\n", x); }

[slytinen@cdmlinux 373s19]$ gcc -o test test.c
[slytinen@cdmlinux 373s19]$ ./test
-0.046982
```

Flow of control: `if` statements

- syntax of simple if statement:

```
if (<condition>) {
    <body> }
```

- The "condition" is evaluated; if it is "true" (see below) then the body is executed
- Example:

```
if (x == y){
    printf("%d\n", x);
}
```

- If body of the if statement is only 1 statement, then { } are optional

```
if (x == y)
    printf("%d\n", x);
```

- logical operators (used in conditions): ==, !=, &&, ||, ! (note that it's ==, not =)

```
if (x == y && x != z)
    printf("%d\n", x);
```

No boolean type in C (noboolean.c)

- And no keywords like `true` or `false`
- In context, 0 is interpreted to mean false and nonzero to mean true

```
int main() {
    int x = 2, y = 5, z = 2;
    int a = (x == y); // 0
    printf("%d %d\n", a, x == z); // prints 1s or 0s
    if (x - y) // if non-zero, it means true
        printf("x and y are not equal\n");
    else printf("x and y are equal\n");
    if (x – z)
       printf("x and z are not equal\n");
    else printf("x and z are equal\n");
}
```

```
int main() {
   int x = 2, y = 5, z = 2;
   int a = (x == y); // 0
   printf("%d %d\n", a, x == z); // prints 1s or 0s
   if (x - y) // if non-zero, it means true
      printf("x and y are not equal\n");
   else printf("x and y are equal\n");
   if (x – z)
      printf("x and z are not equal\n");
   else printf("x and z are equal\n");
}
```

```
0 1
x and y are not equal
x and z are equal
```

This can have unintended consequences

```
#include <stdio.h>

int main() {
   int x = 0;
   printf("The value of x is %d\n", x);
   printf("Does x equal 0? ");
   // this should be ==; compiler doesn’t catch it
   if (x = 0)
      printf("True\n");
   else printf("False\n"); }

$ gcc -o bad bad.c
$ ./bad
The value of x is 0
Does x equal 0? False
```

Flow of control: `if ... else`

- Syntax:

```
if (<condition>) {
   <body1>
}
else {
   <body2>
}
```

- If condition is "true" (see above), body1 is executed; otherwise body2 is executed
- If body1 or body2 are just 1 statement, { } are optional
- Example:

```
if (x == 2) {
   y = 3;
}
else {
   y = 1;
}
```

Flow of control: multiway `if ... else`

- Syntax:

```
if (<condition1>) {
   <body1>
}
else if (<condition2>) {
   <body2>
}
…
else {
   <bodyn>
}
```

Multiway if…else example

```
if (x == 1) {
   y = 10;
}
else if (x == 2) {
   y = 20;
}
else if (x == 3) {
   y = 30;
}
else {
   y = 100;
}
```

C switch statements

- Syntax:

```
switch (<expression>) {
    case <value1>:
        <code1>
        break;
    case <value2>:
        <code2>
        break;
    ...
    case <valuen>:
        <coden>
        break;
    default:
        <default-code>
}
```

Example: switch.c

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

void f(int x, char word[]) {
  switch (x) {
  case 1:  strcpy(word, "one"); break;
  case 2:  strcpy(word, "two"); break;
  case 3:  strcpy(word, "three"); break;
  default: strcpy(word, "greater than three");
  }
}

int main() {
  char word[20];
  int i;
  for (i=1; i<=4; i++) {
    f(i, word);
    printf("%d %s\n", i, word);
  }
}
```

Incorrect Example: switch_wrong.c (SKIP)

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

// this one is wrong
void *f(int x, char word[]) {
  switch(x) {
  case 1:  strcpy(word, "one"); // not ans = "one"
  case 2:  strcpy(word, "two");
  case 3:  strcpy(word, "three");
  default: strcpy(word, "greater than three");
  }
}

int main() {
  char word[20];
  int i;
  for (i=1; i<=4; i++) {
    f(i, word);
    printf("%d %s\n", i, word);
  }
}
```

```
$ ./switch_wrong
1 greater than three
2 greater than three
3 greater than three
4 greater than three
```

while loops (while.c)

- Syntax:

```
while (<condition>) {
   <body>
}
```

- Example:

```
#include <stdio.h>

int main() {
  int x = 10;
  while (x > 0) {
    printf("%d\n", x);
    x--; // same as x = x - 1;
  }
}
```

do … while (dowhile.c)

- Syntax:

```
do {
   <body>
}
while (<condition>);
```

  Example:

```
int i=0;
do {
   printf("%d\n", i++);
}
while (i <= 10);
```

for loops

- Syntax:

```
   for (<initialization>; <condition>; <incrementing>) {
      <body>
   }
```

- Example:

```
int i;
for (i=0; i<10; i++) {
   printf("%d\n", i);
 }
```

- Some C compilers will allow:

```
for (int i=0; i<10; i++) {
   printf("%d\n", i);
 }
```

Variable scope

- A **local** variable is declared (and used) within a function
- Local variables are not known outside of that function

```
int f() {
   x = 10; // syntax error
}
int main() {
   int x = 5;
   f();
}
```

Variable scope

- A **global** variable is declared outside of all functions

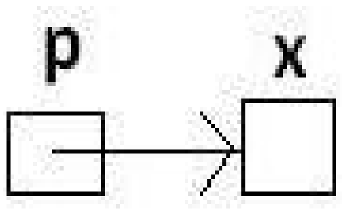
```
int x;
int f() {
   x = 10; // no syntax error
}
int main() {
   x = 5;
   f();
   printf("%d\n", x); // 10
}
```

Pointers

- A **pointer** is the address of a variable
- Created by using the & operator
- Example:

```
int x;
int *p = &x; // p contains the address
of x
```

- Represented graphically as



Passing pointers as parameters

- When a pointer is passed as a parameter, changes in the called function may affect variables in the calling function.

- Example: incr.c

```
void incr(int* x) {
   *x += 1; // now *x is 1 more than it was
}

int main() {
   int y = 1;
   incr(&y); // y is now 2
   printf("%d\n", y);
}
```

scanf (scanf.c)

- Input from console
- Format string similar to `printf` ; codes %d, %f, %s, etc.
- Pointers are passed as parameters, so as to allow `scanf` to store the values it reads
- Example: **scanf.c**

```
#include <stdio.h>

int main() {
  float f;
  float *fptr = &f;
  int x;
  printf("Type a floating point number and an integer\n");
  scanf("%f%d", fptr, &x);
  printf("You typed %f %d\n", f, x);
}
```

The address operator (&)

- & returns the **address** of a variable
- An address is a number between: 32-bit machine: 0 and 2^{32}-1 (4,294,967,295)
  64-bit machine: 0 and 2^{64}-1 ($4,294,967,296^{2}$-1, or 18,446,744,073,709,551,615)

- If a variable takes up more than 1 byte, then the address will be the first byte
  - For example, if the address of an `int` is the first byte of the 4-byte long `int`

The address operator (&) (address.c)

```
// address.c
#include <stdio.h>

int main() {
  int i, j;
  printf("%ld %ld\n", &i, &j);
  // addresses are usually displayed in hex
  printf("%#lx %#lx\n", &i, &j);
}
```

output (usually addresses are inspected as hex numbers, if at all)

```
$ ./address
140724074168588 140724074168584
0x7ffce073c10c 0x7ffce073c108
```

Pointer variables

- A **pointer variable** stores addresses
- Declaration of a pointer variable:

```
<type>* <name>;
```

Example:

```
int x;
int* p;
// p is a pointer variable, and now
// contains the address of x
p = &x;
```

The dereferencing operator (*)

- This operator is the opposite of the address operator, as illustrated below.

```
int x = 5;
printf("%d\n", *&x); // prints 5
```

- Likewise:

```
int x = 5;
int y = *&x; // y is 5
```

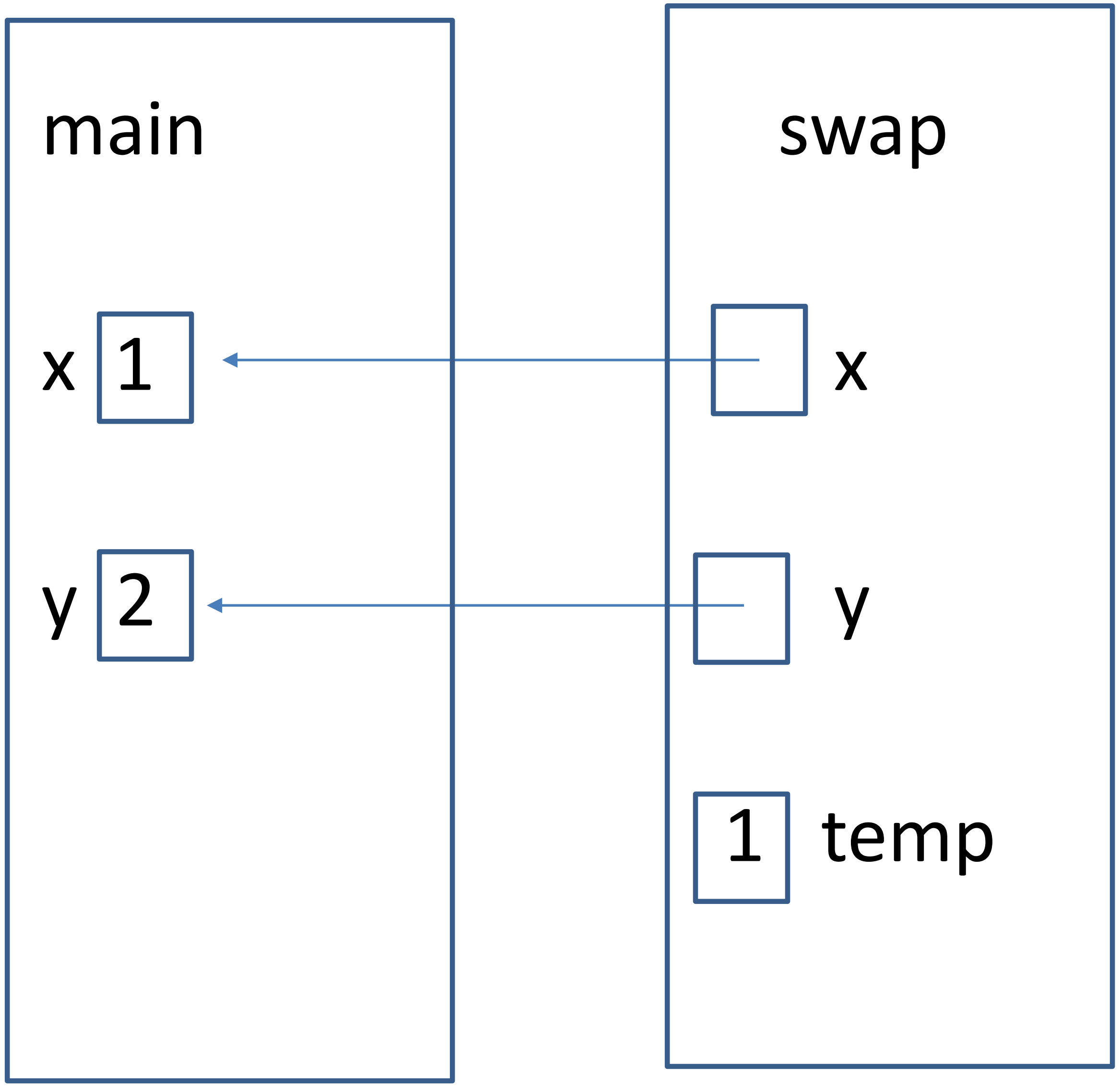
- Another example:

```
int x = 5;
int* p = &x;
*p = 10; // now x is 10
```

Example

```
// swap.c
#include <stdio.h>

void swap(int *i, int *j) {
  int temp = *i;
  *i = *j;
  *j = temp;
}

int main() {
  int x=1, y=2;
  int xptr = &x, yptr = &y;
  swap(xptr, yptr);
  printf("x = %d y = %d\n", x, y); // prints 2 1
}
```

main and swap functions



* Symbol has 3 meanings

1. **Multiplication**, as in **x * y;**

2. **Pointer variable declaration**, as in **int *x;**

3. **Deferencing,** as in

```
int x;
int *p = x;   // pointer variable declaration
*p = 10       // dereferencing
                // has the side effect of setting x to 10
```

C Arrays

- An **array** is a block of contiguous memory, which can store several pieces of data all of the same type
- An array is declared as follows: `<type> <name>[<size>];`
- The size cannot be a variable
- Examples

```
int x[10]; // x is 40 bytes, and can
contain 10 integers
char c[5]; // c is 5 bytes, and can contain
5 chars
int x[i];  // not allowed
```

C Arrays (array.c)

- Example:

```
int x[10];    // declares an array of size 10
x[0] = 10;    // assignment
x[1] = 20;    // assignment
int y = x[1]; // retrieval
```

- Another example: **array.c**

```
int arr[5];
int i;
for (i=0; i<5; i++)
      arr[i] = i*10;
for (i=4; i>=0; i--)
   print("%d\n", arr[i]);
```

Passing an array as a parameter

- Note that the length of the array must be passed as a separate parameter

```
// max.c
int max(int x[], int len) {
    int answer = x[0];
    int i;
    for (i=1; i<len; i++)
        if (x[i] > answer)
              answer = x[i];
    return answer;
}

int main() {
  int x[] = {2, 1, 4, 6, 3, 5};
  printf("%d\n", max(x, 6));
}
```

Array variables are pointers

- An array variable is a pointer, which points to the beginning of the block of memory allocated for the array.
- Example:

```
int x[10];
// prints the address at which x starts
printf("%#x\n", x);
```

- Therefore, one way to store or access data in an array is to use the dereferencing operator

Array variables are pointers

- An array variable is a **constant pointer**; it cannot be changed after it has been declared

```
int x[10];
int y;
int* z;
x = &y; // not allowed
z = x;  // OK
```

Array variables are pointers

- Example: **arrays.c** (the C file has more code)

```
int x[10];
int i;
for (i=0; i<10; i++)
   *(x+i) = 10*i; // this is "pointer arithmetic"
for (i=0; i<10; i++)
   printf("%d ", *(x+i)); //prints 0 10 20 30 ... 80 90
printf("\n");
```

- In pointer arithmetic, an integer value is added to a pointer so that it points further along (or further back) in memory. In this case, *(x+i) is equivalent to x[i].

- C compensates for the size (number of bytes) of the data in the array. So in third case, `(x + i)` under the hood is `(x + 4*i)` although we don't write it that way

Another way to write the same code

- Example:

```
int x[10];
int i;
for (i=0; i<10; i++)
   x[i] = 10*i;
for (i=0; i<10; i++)
   printf("%d ", x[i]); //prints 0 10 20 30 ... 80 90
printf("\n");
```

Exercises

- Write a function called range. It (sort of) emulates the Python range function. More precisely it should fill an array of type int[] with the values 0, 1, 2, …, n-1 (where n is one of the parameters passed to range). It returns a pointer to the array.

Exercises

- Write a function called `read_array.` It is passed an integer array and its length. The user should be prompted to enter the appropriate number of integers, and their values should be stored in the array. The prototype for this function is:

```
void read_array(int x[ ], int len);
```

- Write a function called `sum_array`. This function is passed an integer array, and the length of the array. It returns the sum of the integers. Write two versions of this function. In the first, use array syntax (i.e., use [ ] and not & or *). In the other version, use pointer syntax (i.e., no [ ]).

C Strings

- Strings in C are arrays of chars, with a **null character ('\0')** at the end. Because of this null character, the array's length is (at least) 1 greater than the number of characters in the string
- string arrays can be initialized with "..."
- Example:

```
int main() {
   // cstring -- s[3] is '\0'. Some space left.
   char s[10] = "yes";
   // cstring, with just enough space (including '\0')
   char t[4] = "yes";
   // NOT a string -- NO '\0' at a[3]
   char a[10] = {'y','e','s'};
}
```

Declaration and initialization

```
int main() {
    // spaces are allowed in C strings
    char s1[8] = "csc 373";
    // if the length is not specified,
    // compiler automatically decides the size needed
    char s2[] = "hello";
    // uninitialized
    char s3[20];
}
```

- The construction `char s2[] = "hello"` can only be used in a declaration. We cannot say, for example:

```
    char s3[20];
    s3 = "hello";
```

Things you can and cannot do with strings

```
int main() {
   char s1[5] = "john";
   char s2[5] = "jack";
   char s3[20];
   // Things you can NOT do:
   if (s1 == s2) // NO comparison by ==, same as any array
      ...
   s1 = s2;      // NO assignment by =, same as any array
   // Things you CAN do:
   // Different from regular arrays.
   scanf("%s", s3); // OK. string input until white space.
   // Null terminator is automatically added at the end.
   printf("%s", s3); // OK. string output until '\0'.
   // Passing string to a function without size -- why is it
   // possible?
   if (isPalindrome(s3)) ...
```

Strings as arguments to functions

hyphens1.c

```
int main() {
    char s1[20] = "off-campus and on-campus housing";
    int h = countHyphens1(s1);
    printf("%d\n", h);
}

int countHyphens1(char s[]) {
    int count = 0;
    int i;
    for (i = 0; s[i] != '\0'; i++) {
        if (s[i] == '-')
            count++; }
    return count;
}
```

Strings as arguments to functions

hyphens2.c

```
int main() {
   char s1[20] = "off-campus and on-campus housing"; /
   int h = countHyphens2(s1);
   printf("%d\n", h);
}

int countHyphens2(char* s) {
   int count = 0;
   for(; *s != '\0'; s++) {
      if (*s == '-')
         count++;
   }
   return count;
}
```

<string.h>

- There are some C predefined functions which deal with cstrings.
- You need to **#include <string.h>** to use those functions.

```
// stringlib.c
#include <stdio.h>
#include <string.h>

int main() {
  char s1[8] = "csc 373";
  char s2[15] = "c language";
  char s3[30];
  // string length, returns 7 not 8
  int len = strlen(s1);
  // continued on next slide
// string comparison, returns # of difference
// in lexicographic order
int diff = strcmp(s1, s2);
// normally false, but in this case means
// the strings are equal
if (diff == 0)
  printf("%s and %s are the same\n", s1, s2);
else if (diff < 0)
  printf("%s is before %s\n", s1, s2);
// this one will print
else printf("%s is before %s\n", s2, s1);
// continued on next slide
```

```
  strcpy(s3, s1);
  strcat(s3, " ");
  strcat(s3, s2);
  printf("%s\n", s3); // prints csc 373 c language
  char s4[] = "373";
  int i = atoi(s4);
  printf("%d\n", i); // prints 373
}
```

Exercise

- Write functions **strlen373_v1 and strlen373_v2**, which both mimic the behavior of the C string library's **strlen** function.

  - First write it using "array syntax" (i.e., use [ ]).

```
int strlen373_v1(char str[]) {
   return 0; // change this
}
```

  - Then write it using "pointer syntax" (no [ ]).

```
int strlen373_v2(char *str) {
   return 0; // change this
}
```

- Write in the file strlen373.c

Exercise

- Write functions **strcpy373_v1 and strcpy373_v2**, which mimic the behavior of the C string library's **strcpy** function.

  - First write it using "array syntax" (i.e., use [ ]).

```
void strcpy373_v1(char dest[], char src[]) {

}
```

  - Then write it using "pointer syntax" (no [ ]).

```
void strcpy373_v2(char *dest, char *src) {

}
```

  - Write it in strcpy373.c

A preview of data types

- Consider this program (**ascii.c**)

```
#include <stdio.h>
int main()
{
  int i;

  for (i = 0; i < 128; i++)
    printf("%c:  %d\n", i, i);
}
```

- If you run it, you will see that the program attempts to print each number as an ASCII character, according to the ascii code you can find at http://www.asciitable.com

- Each character is really just a 7-bit number, corresponding to its **ascii** encoding (a number between 0 and 127)

- Printing a number using %c tells printf to treat it like a character

Bit-Level Operations in C

- Operations &, |, ~, ^ available in C
- Apply to any “integral” data type, signed or unsigned
    - long, int, short, `char`
- View arguments as bit vectors

Bit-Level Operations in C

- View arguments as bit vectors
- Arguments applied bit-wise
- Examples (Char data type, 8 bits)

  $\sim 01000001_2$ --> 10111110_2
  $\sim 00000000_2$ --> 11111111_2
  01101001_2 & 01010101_2 --> 01000001_2
  01101001_2 | 01010101_2 --> 01111101_2
  01101001_2 ^ 01010101_2 --> 00111100_2

bitops.c

```
#include <string.h>
int main() {
  char op[10], n1, n2, n1_str[10], n2_str[10];
  while (1) {
    printf("Enter a bitwise expression\n");
    scanf("%s", n1_str);
    if (strcmp(n1_str, "q") == 0)
      return;
    scanf("%s %s", op, &n2_str);
   n1 = atoi(n1_str);
    n2 = atoi(n2_str);
}
```

bitops.c

```
   n1 = atoi(n1_str);
   n2 = atoi(n2_str);
   if (strcmp(op, "|") == 0)
      printf("%d | %d = %d\n", n1, n2, n1|n2);
    else if (strcmp(op, "&") == 0)
      printf("%d & %d = %d\n", n1, n2, n1&n2);
    else if (strcmp(op, "^") == 0)
      printf("%d ^ %d = %d\n", n1, n2, n1^n2);
    else printf("Invalid operator\n");
  }
}
```

Shift operations

- **Left Shift:** **x << y**
  - Shift bit-vector x left y positions
    - Throw away extra bits on left
    - Fill with 0's on right
- **Right Shift:** **x >> y**
  - Shift bit-vector x right y positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
    - Useful with two's complement integer representation (which we will talk about next week)

Examples of Shift operations

`x`	`01100010`
`x << 3`	`00010000`
`x >> 2`	`00011000`
`y`	`10100010`
`y >> 2 (Arithmetic)`	`11101000`
`y >> 2 (Logical)`	`00101000`

Example use of shifting: `print_binary.c`

```
#include <stdio.h>

void print_binary(int x) {
  // makes this machine-independent
  int size = (sizeof x) * 8;
  // 2's complement
  if (x<0) printf("1");
  else printf("0");
  int i;
  int mask = 1;
  // skip the sign bit
  mask = mask << (size-2);
  for (i=1; i<size; i++) {
    if (x & mask) printf("1");
    else printf("0");
    mask = mask >> 1;
  }
}
```

Examples of bitwise operators: `bitwise.c`

```
#include <stdio.h>
#include "print_binary.c"

int main() {
  int x,y; unsigned int z;
  printf("Enter 2 numbers\n");
  scanf("%d %d", &x, &y);
  printf("x in binary is ");   print_binary(x);
  printf("\ny in binary is "); print_binary(y);
  printf("\n\nx & y = ");      print_binary(x&y);
  printf("\nx | y = ");        print_binary(x|y);
  printf("\nx ^ y = ");        print_binary(x^y);
  printf("\n\nx = ");          print_binary(x);
  printf("\n~x = ");           print_binary(~x);
}
```

See also **bitops.c**

Examples of shift operators: see `shift.c`

Separate compilation

- C code can be broken into "modules"
- Separate modules can be compiled separately; makes for easier development by different programmers or teams
- Usual approach involves:
  - Header file (.h)
    - Typically uses #ifndef macro
  - Module source code (.c)
  - Code which uses the module: include the .h file
- Example
  - my_math.h
  - my_math.c
  - my_math_main.c

Linux commands

```
$ gcc -c my_math.c -o my_math.o
$ gcc –c my_math_main.c –o my_math_main.o
$ gcc –o main my_math_main.o my_math.o
$ ./main
```