YORK UNIVERSITÉ UNIVERSITY

Introduction to C Programming (Part C)

Overview (King Ch. 13, 22, 16-17)

- Strings (Ch. 13)
- Input/Output (Ch. 22)
- Structures (Ch. 16)
- Dynamic memory Management/Linked Lists (Ch. 17)
- Makefile (Ch. 15)

Chapter 13

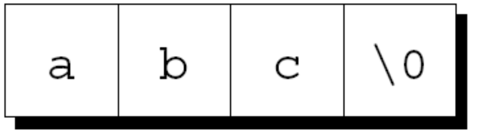
Strings

Introduction

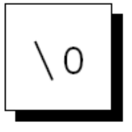
- This chapter covers
  - string *constants* (or *literals*)
  - string *variables*
- Strings are arrays of characters
- The C library provides a collection of functions for working with strings: `<string.h>`

How String Literals Are Stored

- The string literal `"abc"` is stored as an array of four characters:



- A special character—the ***null character***—marks the end of a string (i.e., a byte whose bits are all zero)
- The string `""` is stored as a single null character:



Operations on String Literals

- We can use a string literal wherever C allows a `char *` pointer:

  ```
  char *p;
  p = "abc";
  ```

  This assignment makes `p` point to the first character of the string.
- String literals can be subscripted:

  ```
  char ch;
  ch = "abc"[1];
  ```

  The new value of `ch` will be the letter `b`

Operations on String Literals

- Attempting to modify a string literal causes undefined behavior:

```
char *p = "abc";

*p = 'd';    /*** WRONG ***/
```

- A program that tries to change a string literal may crash or behave erratically.

String Variables

- If a string variable needs to hold 80 characters, it must be declared to have length 81:

```
#define STR_LEN 80
…
char str[STR_LEN+1];
```

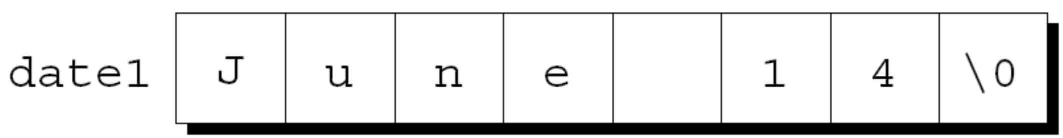
- Adding 1 to the desired length allows room for the null character at the end of the string.

Initializing a String Variable

- A string variable can be initialized at the same time it's declared:

```
char date1[8] = "June 14";
```

- The compiler will automatically add a null character so that `date1` can be used as a string:



- We may omit its length:

```
char date1[] = "June 14";
```

Initializing a String Variable

- If the initializer is too short to fill the string variable, the compiler adds extra null characters:

```
char date2[9] = "June 14";
```

date2	J	u	n	e		1	4	\0	\0

- An initializer can’t be longer than the variable size:

```
char date3[7] = "June 14”;
```

date3	J	u	n	e		1	4

Character Arrays vs Character Pointers

- The declaration

  ```
  char date[] = "June 14";
  ```

  declares `date` to be an *array,*
- The similar-looking

  ```
  char *date = "June 14";
  ```

  declares `date` to be a *pointer.*
- Thanks to the close relationship between arrays and pointers, either version can be used as a string.

Character Arrays vs Character Pointers

- However, there are significant differences between the two versions of `date`.
  - In the array version, the characters stored in `date` can be modified. In the pointer version, `date` points to a string literal that **shouldn't** be modified.
    - String literals are stored at **read-only memory** and trying to modify this memory leads to undefined behavior (memory access violation)
  - In the array version, `date` is an array name. In the pointer version, `date` is a pointer variable that can point to other strings.

Character Arrays vs Character Pointers

- The declaration

  ```
  char *p;
  ```

  does not allocate space for a string.
- Before we can use `p` as a string, it must point to an array of characters.
- One possibility is to make `p` point to a string variable:

  ```
  char str[STR_LEN+1], *p;
  p = str;
  ```

- Another possibility is to make `p` point to a dynamically allocated string.

Character Arrays vs Character Pointers

- Using an uninitialized pointer variable as a string is a serious error.
- An attempt at building the string `"abc"`:

```
char *p;

p[0] = 'a';      /*** WRONG ***/
p[1] = 'b';      /*** WRONG ***/
p[2] = 'c';      /*** WRONG ***/
p[3] = '\0';     /*** WRONG ***/
```

- Since `p` hasn't been initialized, this causes undefined behavior.

Reading and Writing Strings

- Writing a string is easy using `printf` or `puts`.
- Reading a string is easy using `scanf` or `gets`.
- We can read strings one character at a time, using a function that
  - (1) doesn't skip white-space characters
  - (2) stops reading at the first new-line character (which isn't stored in the string)
  - (3) discards extra character

Reading Strings Char. by Char.

- `read_line` consists primarily of a loop that calls `getchar` to read a character and then stores the character in `str`, provided that there's room left:

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';    /* terminates string */
  return i;         /* number of characters stored */
}
```

- `ch` has `int` type rather than `char` type because `getchar` returns an `int` value.

Accessing the Characters in a String

- Since strings are stored as arrays, we can use subscripting to access the characters in a string.
- To process every character in a string `s`, we can set up a loop that increments a counter `i` and selects characters via the expression `s[i]`.

Accessing the Characters in a String

- A function that counts the number of spaces in a string:

```
int count_spaces(const char s[])
{
  int count = 0, i;

  for (i = 0; s[i] != '\0'; i++)
    if (s[i] == ' ')
      count++;
  return count;
}
```

Accessing the Characters in a String

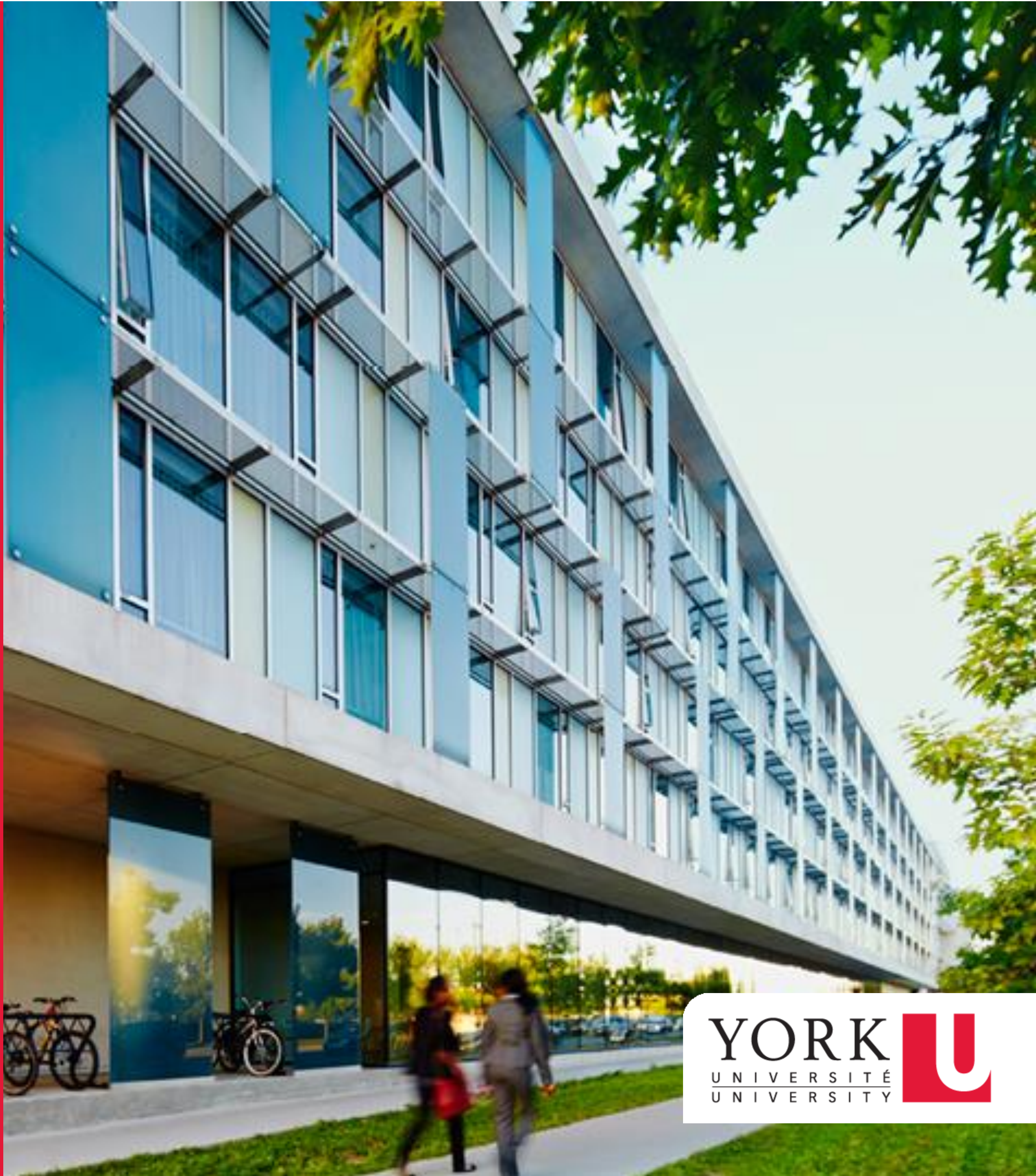
- A version that uses pointer arithmetic instead of array subscripting :

```
int count_spaces(const char *s)
{
  int count = 0;

  for (; *s != '\0'; s++)
    if (*s == ' ')
      count++;
  return count;
}
```

Accessing the Characters in a String

- Questions raised by the `count_spaces` example:
  - ***Q1**: Is it better to use array operations or pointer operations to access the characters in a string?*
    - **A1**: We can use either or both. Traditionally, C programmers lean toward using pointer operations.
  - ***Q2**: Should a string parameter be declared as an array or as a pointer?*
    - **A2**: There's no difference between the two.
  - ***Q3**: Does the form of the parameter (`s[]` or `*s`) affect what can be supplied as an argument?*
    - **A3**: No.

Array of Strings



Array of Strings

- There is more than one way to store an array of strings.
- One option is to use a two-dimensional array of characters, with one string per row:

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

- The number of rows in the array can be omitted, but we must specify the number of columns.

Arrays of Strings

- Unfortunately, the `planets` array contains a fair bit of wasted space (extra null characters):

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

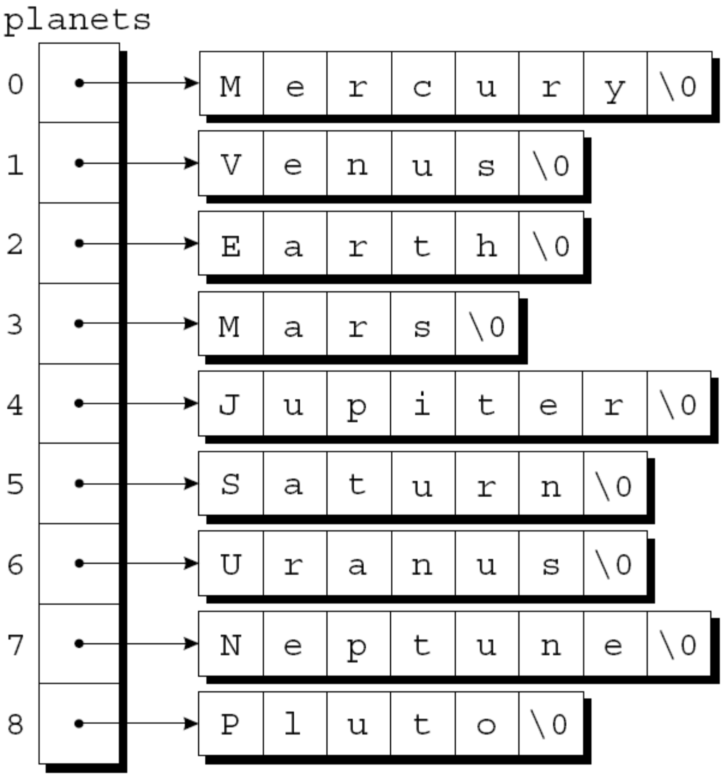
Arrays of Strings

- Most collections of strings will have a mixture of long strings and short strings.
- What we need is a ***ragged array,*** whose rows can have different lengths.
- We can simulate a ragged array in C by creating an array whose elements are *pointers* to strings:

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

Arrays of Strings

- This small change has a dramatic effect on how `planets` is stored:



Arrays of Strings

- To access one of the planet names, all we need do is subscript the `planets` array.
- Accessing a character in a planet name is done in the same way as accessing an element of a two-dimensional array.
- A loop that searches the `planets` array for strings beginning with the letter M:

```
for (i = 0; i < 9; i++)
  if (planets[i][0] == 'M')
    printf("%s begins with M\n", planets[i]);
```

Command-Line Arguments

- When we run a program, we'll often need to supply it with information.
- This may include a file name or a switch that modifies the program's behavior.
- Examples of the UNIX `ls` command:

```
ls
ls –l
ls -l remind.c
```

Command-Line Arguments

- Command-line information is available to all programs, not just operating system commands.
- To obtain access to ***command-line arguments***, `main` must have two parameters:

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

- Command-line arguments are called ***program parameters*** in the C standard.

Command-Line Arguments

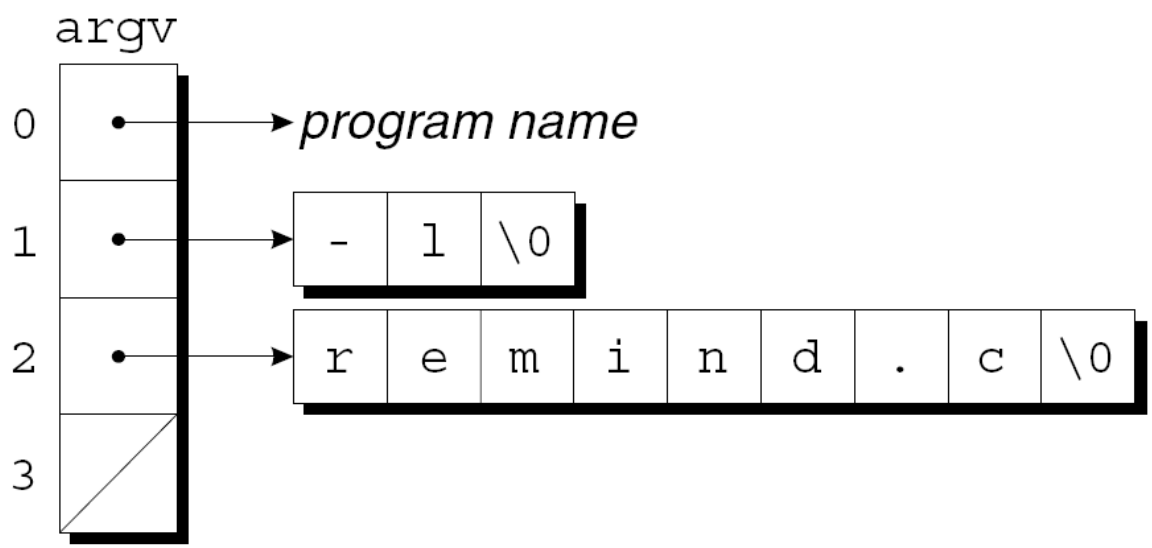
- `argc` ("argument count") is the number of command-line arguments.
- `argv` ("argument vector") is an array of pointers to the command-line arguments (stored as strings).
- `argv[0]` points to the name of the program, while `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments.
- `argv[argc]` is always a ***null pointer***—a special pointer that points to nothing.
  - The macro `NULL` represents a null pointer.

Command-Line Arguments

- If the user enters the command line

  `ls -l remind.c`

  then `argc` will be 3, and `argv` will have the following appearance:



Command-Line Arguments

- Since `argv` is an array of pointers, accessing command-line arguments is easy.
- Typically, a program that expects command-line arguments will set up a loop that examines each argument in turn.
- One way to write such a loop is to use an integer variable as an index into the `argv` array:

```
int i;

for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

YORK UNIVERSITÉ UNIVERSITY

Command-Line Arguments

- Another technique is to set up a pointer to `argv[1]`, then increment the pointer repeatedly:

```
char **p;

for (p = &argv[1]; *p != NULL; p++)
  printf("%s\n", *p);
```

Program: Checking Planet Names

- The `planet.c` program illustrates how to access command-line arguments.
- The program is designed to check a series of strings to see which ones are names of planets.
- The strings are put on the command line:

```
$ ./planet Jupiter venus Earth fred
```

- The program will indicate whether each string is a planet name and, if it is, display the planet's number:

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

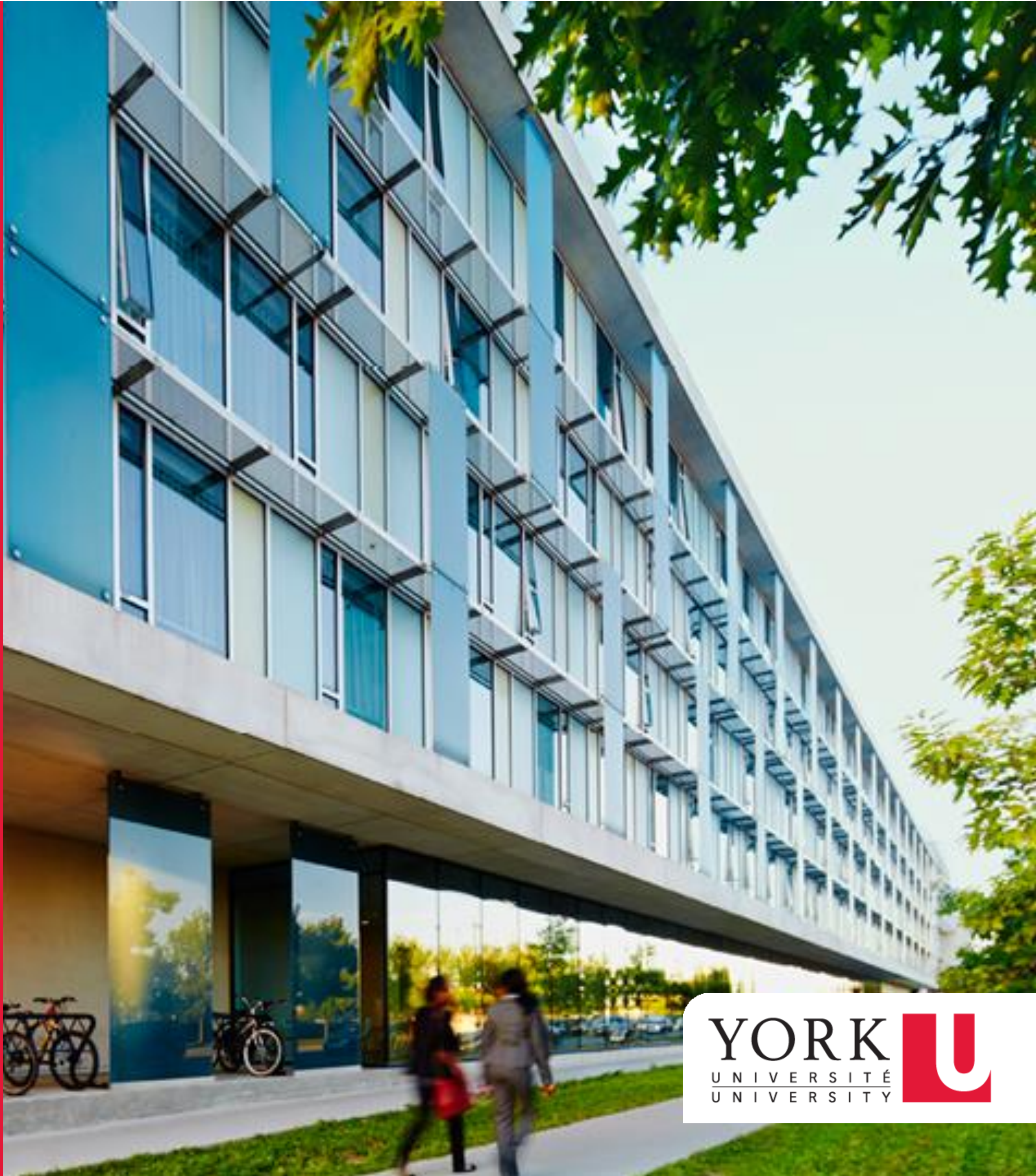
planet.c

```
/* Checks planet names */

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

#define NUM_PLANETS 9

int main(int argc, char *argv[])
{
  char *planets[] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
  int i, j;
```

planet.c (cont.)

```
  for (i = 1; i < argc; i++) {
    for (j = 0; j < NUM_PLANETS; j++)
      if (strcmp(argv[i], planets[j]) == 0) {
        printf("%s is planet %d\n", argv[i], j + 1);
        break;
      }
    if (j == NUM_PLANETS)
      printf("%s is not a planet\n", argv[i]);
  }

  return 0;
}
```

C String Library & Code Examples



Chapter 22

Input/Output

Streams

- In C, the term ***stream*** means any source of input or any destination for output.
- Many small programs obtain all their input from one stream (the keyboard) and write all their output to another stream (the screen).
- Larger programs may need additional streams.
- Streams often represent files stored on various media.
- However, they could just as easily be associated with devices such as network ports and printers.

File Pointers

- Accessing a stream is done through a ***file pointer,*** which has type `FILE *`.
- The `FILE` type is declared in `<stdio.h>`.
- Certain streams are represented by file pointers with standard names.
- Additional file pointers can be declared as needed:

```
FILE *fp1, *fp2;
```

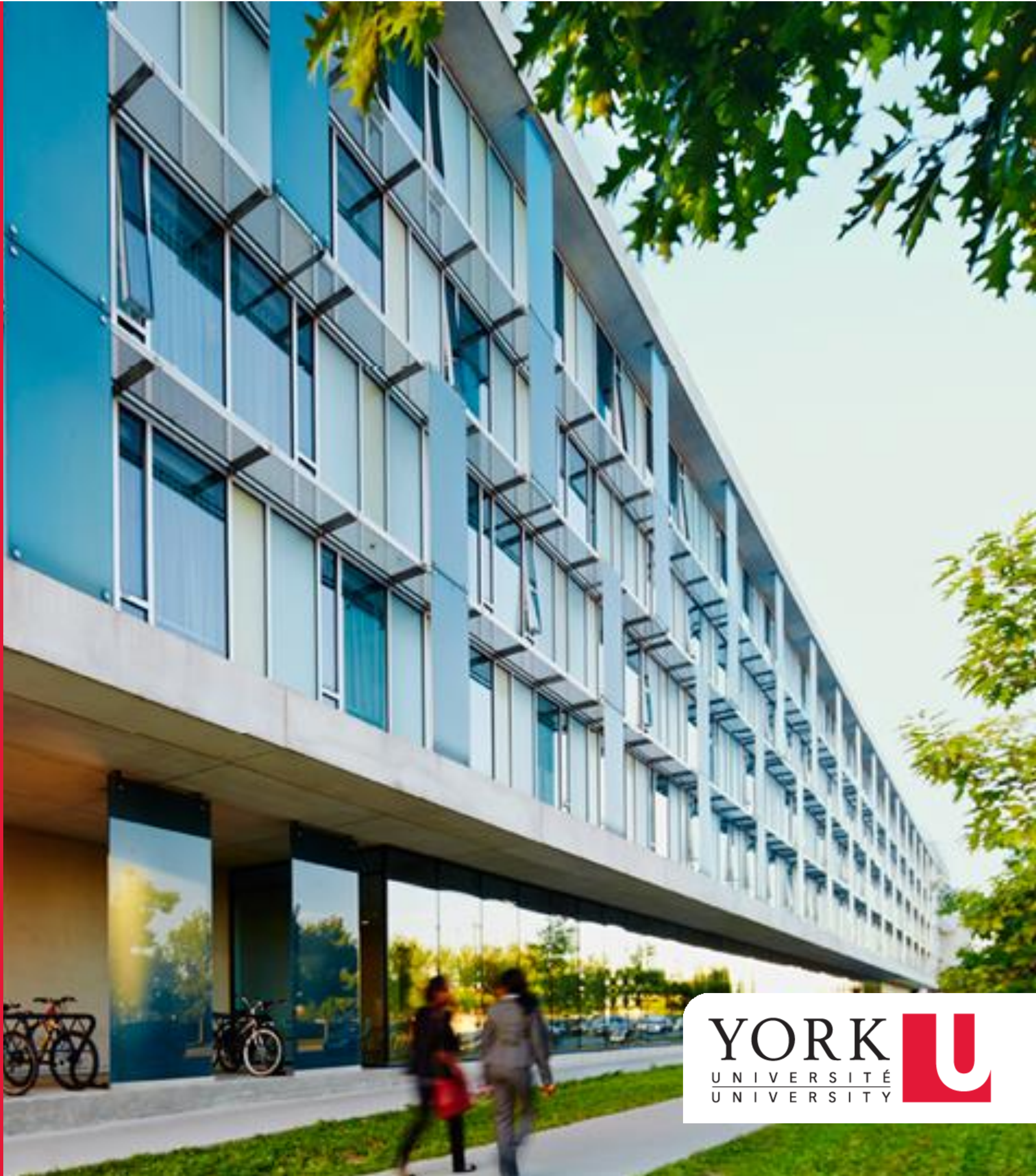
Standard Streams and Redirection

- `<stdio.h>` provides three standard streams:

File Pointer	***Stream***	***Default Meaning***
`stdin`	Standard input	Keyboard
`stdout`	Standard output	Screen
`stderr`	Standard error	Screen

- These streams are ready to use—we don't declare them, and we don't open or close them.
- File operations are provided by `<stdio.h>`.

File Operations



Opening a File

- Opening a file for use as a stream requires a call of the `fopen` function. Prototype for `fopen`:

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

- `filename` is the name of the file to be opened.
  - This argument may include information about the file's location, such as a drive specifier or path.
- `mode` is a "mode string" that specifies what operations we intend to perform on the file.

Opening a File

- `fopen` returns a file pointer that the program saves in a variable or returns a null pointer.

```
fp = fopen("in.dat", "r");
  /* opens in.dat for reading */
```

- Or combined with the declaration of `fp`:

```
FILE *fp = fopen(FILE_NAME, "r");
```

- Or combined with test against `NULL`:

```
if ((fp = fopen(FILE_NAME, "r")) == NULL) …
```

Opening a File

- In Windows, be careful when the file name in a call of `fopen` includes the `\` character.
- The following call will fail:

```
fopen("c:\project\test1.dat", "r”)
```

- One way to avoid the problem is to use `\\` instead of `\`:

```
fopen("c:\\project\\test1.dat", "r")
```

- An alternative is to use the `/` character instead of `\`:

```
fopen("c:/project/test1.dat", "r")
```

Modes

- Factors that determine which mode string to pass to `fopen`:
  - Which operations are to be performed on the file
  - Whether the file contains text or binary data

Modes

- Mode strings for text files:

String	***Meaning***
`"r"`	Open for reading
`"w"`	Open for writing (file need not exist)
`"a"`	Open for appending (file need not exist)
`"r+"`	Open for reading and writing, starting at beginning
`"w+"`	Open for reading and writing (truncate if file exists)
`"a+"`	Open for reading and writing (append if file exists)

Closing a File

- The `fclose` function allows a program to close a file that it’s no longer using.
- The argument to `fclose` must be a file pointer obtained from a call of `fopen` or `freopen`.
- `fclose` returns zero if the file was closed successfully.
- Otherwise, it returns the error code `EOF` (a macro defined in `<stdio.h>`).

Opening/Closing a File

- The outline of a program that opens a file for reading:

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

#define FILE_NAME "example.dat"

int main(void)
{
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Can't open %s\n", FILE_NAME);
    exit(EXIT_FAILURE);
  }
  …
  fclose(fp);
  return 0;
}
```

Writing to a File

- The `fprintf` and `printf` functions write a variable number of data items to an output stream, using a format string to control the appearance of the output.
- The prototypes for both functions end with the `...` symbol (an ***ellipsis***), which indicates a variable number of additional arguments:

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

- Both functions return the number of characters written; a negative return value indicates that an error occurred.

Writing to a File

- `printf` always writes to `stdout`, whereas `fprintf` writes to the stream indicated by its first argument:

```
printf("Total: %d\n", total);
  /* writes to stdout */
fprintf(fp, "Total: %d\n", total);
  /* writes to fp */
```

- A call of `printf` is equivalent to a call of `fprintf` with `stdout` as the first argument.

Writing to a File

- `fprintf` works with any output stream.
- One of its most common uses is to write error messages to `stderr`:

```
fprintf(stderr, "Error: data file can't be opened.\n");
```

- Writing a message to `stderr` guarantees that it will appear on the screen even if the user redirects `stdout`.

Reading from a File

- `fscanf` and `scanf` read data items from an input stream, using a format string to indicate the layout of the input.
- After the format string, any number of pointers—each pointing to an object—follow as additional arguments.
- Input items are converted (according to conversion specifications in the format string) and stored in these objects.

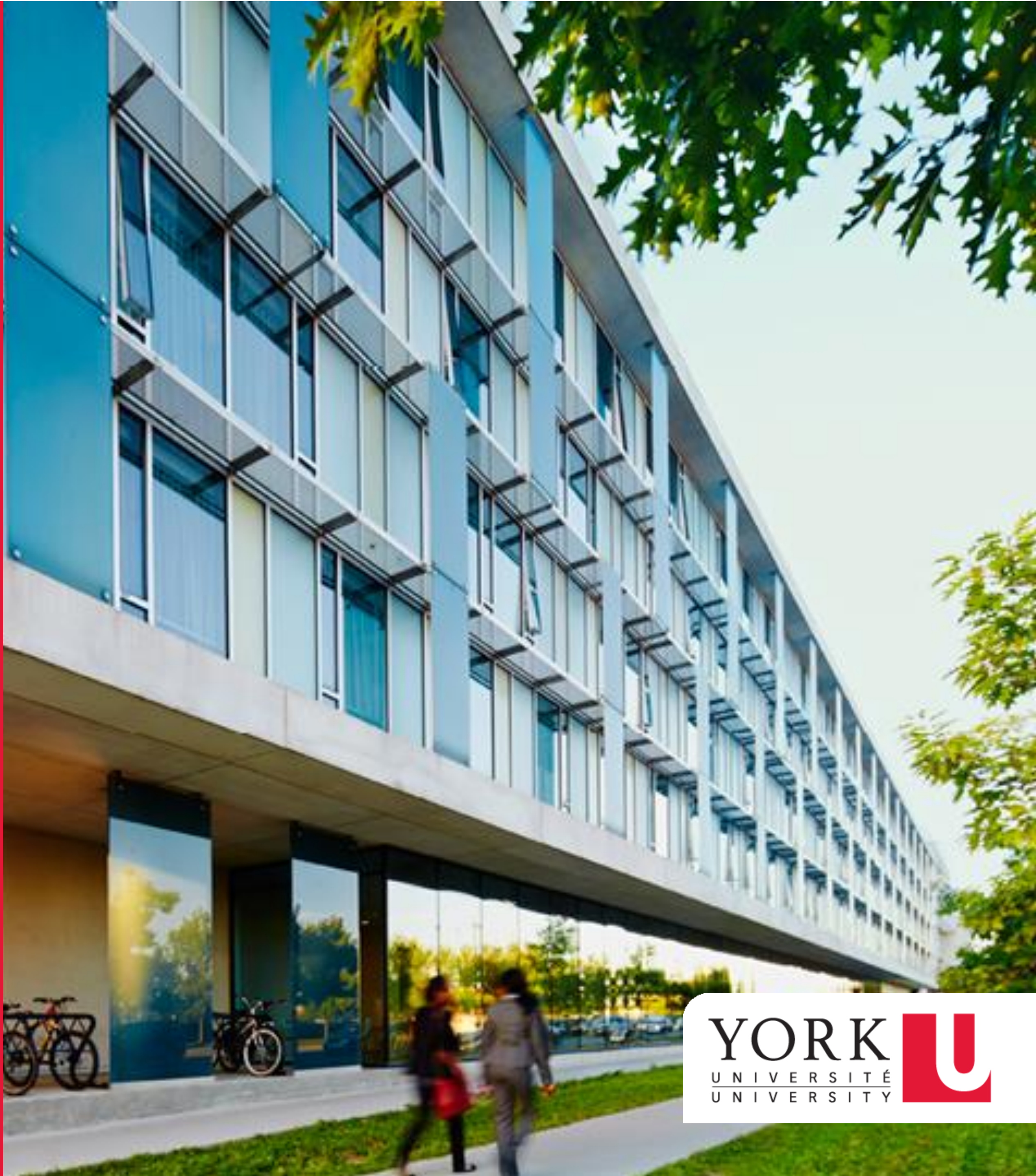
Detecting End-of-File (EOF) and Error Conditions

- If we ask a …`scanf` function to read and store *n* data items, we expect its return value to be *n*.
- If the return value is less than *n*, something went wrong:
  - ***End-of-file.*** The function encountered end-of-file before matching the format string completely.
  - ***Read error.*** The function was unable to read characters from the stream.
  - ***Matching failure.*** A data item was in the wrong format.

Detecting End-of-File and Error Conditions

- Every stream has two indicators associated with it: an ***error indicator*** and an ***end-of-file indicator.***
- These indicators are cleared when the stream is opened.
- Encountering end-of-file sets the end-of-file indicator, and a read error sets the error indicator.
  - The error indicator is also set when a write error occurs on an output stream.
- A matching failure doesn’t change either indicator.

Code Examples



Chapter 16

Structures

Structure Variables

- The properties of a ***structure*** are different from those of an array.
  - The elements of a structure (its ***members***) aren't required to have the same type.
  - The members of a structure have names; to select a particular member, we specify its name, not its position.

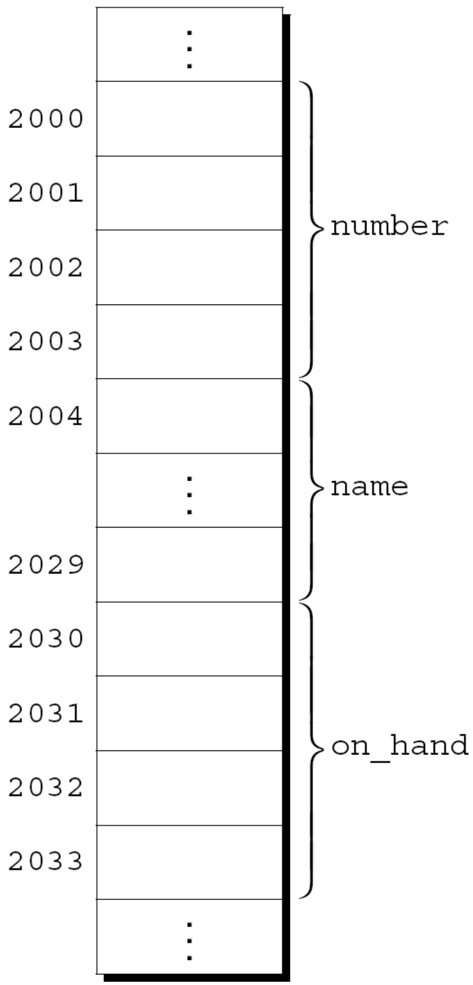
Declaring Structure Variables

- A structure is a logical choice for storing a collection of related data items.
- A declaration of two structure variables that store information about parts in a warehouse:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

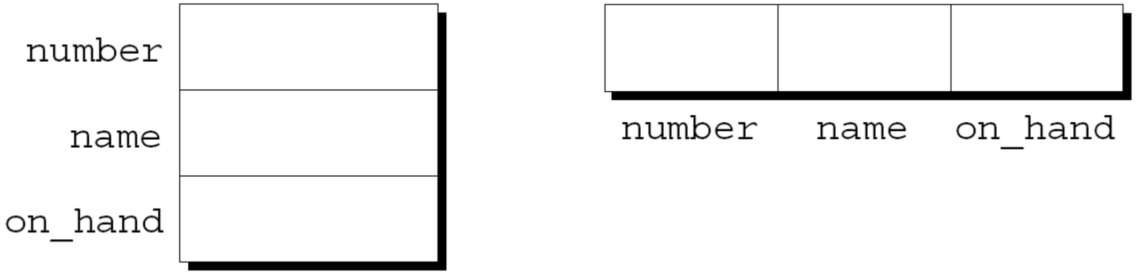
Declaring Structure Variables

- The members of a structure are stored in memory in the order in which they're declared.
- Appearance of `part1`
- Assumptions:
  - `part1` is located at address 2000.
  - Integers occupy four bytes.
  - `NAME_LEN` has the value 25.
  - There are no gaps between the members.



Declaring Structure Variables

- Abstract representations of a structure:



- Member values will go in the boxes later.

Declaring Structure Variables

- Each structure represents a new **scope**.
- Any names declared in that scope won't conflict with other names in a program.
- In C terminology, each structure has a separate ***name space*** for its members.

Declaring Structure Variables

- For example, the following declarations can appear in the same program:

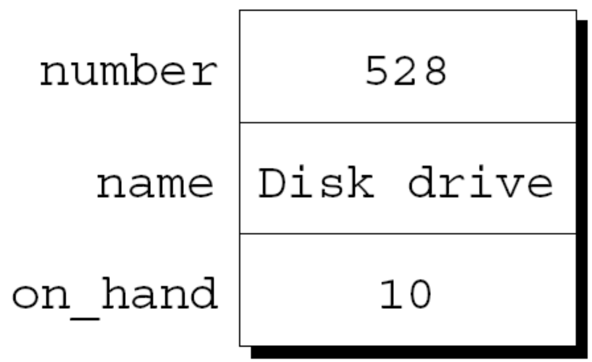
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

struct {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;
```

Initializing Structure Variables

- A structure declaration may include an initializer:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```

- Appearance of `part1` after initialization:



Operations on Structures

- To access a member within a structure, we write the name of the structure first, then a period, then the name of the member.
- Statements that display the values of `part1`'s members:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

Operations on Structures

- The members of a structure are **lvalues**.
- They can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258;
  /* changes part1's part number */
part1.on_hand++;
  /* increments part1's quantity on hand */
```

Operations on Structures

- The period used to access a structure member is actually a C operator.
- It takes precedence over nearly all other operators.
- Example:

```
scanf("%d", &part1.on_hand);
```

The `.` operator takes precedence over the `&` operator, so `&` computes the address of `part1.on_hand`.

Operations on Structures

- The other major structure operation is assignment:

  `part2 = part1;`

- The effect of this statement is to copy `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.

Operations on Structures

- Arrays can’t be copied using the = operator, but an array embedded within a structure is copied when the enclosing structure is copied.
- Some programmers exploit this property by creating “dummy” structures to enclose arrays that will be copied later:

```
struct { int a[10]; } a1, a2;
a1 = a2;
  /* legal, since a1 and a2 are structures */
```

Structure Types

- Suppose that a program needs to declare several structure variables with identical members.
- We need a name that represents a *type* of structure, not a particular structure *variable*.
- Ways to name a structure:
  - Declare a “structure tag”
  - Use `typedef` to define a type name

Declaring a Structure Tag

- A ***structure tag*** is a name used to identify a particular kind of structure.
- The declaration of a structure tag named `part`:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

- Note that a semicolon must follow the right brace.

Declaring a Structure Tag

- The `part` tag can be used to declare variables:

```
struct part part1, part2;
```

- We can't drop the word `struct`:

```
part part1, part2;    /*** WRONG ***/
```

`part` isn't a type name; without the word `struct`, it is meaningless.

Declaring a Structure Tag

- The declaration of a structure *tag* can be combined with the declaration of structure *variables:*

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Defining a Structure Type

- As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name.
- A definition of a type named `Part`:

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

- `Part` is used in the same way as the built-in types:

```
Part part1, part2;
```

Nested Arrays and Structures

- Structures and arrays can be combined without restriction.
- Arrays may have structures as their elements, and structures may contain arrays and structures as members.

Nested Structures

- Nesting one structure inside another is often useful.
- Suppose that `person_name` is the following structure:

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```

Nested Structures

- We can use `person_name` as part of a larger structure:

```
struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

- Accessing `student1`'s first name, middle initial, or last name requires two applications of the . operator:

```
strcpy(student1.name.first, "Fred");
```

Arrays of Structures

- One of the most common combinations of arrays and structures is an array whose elements are structures.
- This kind of array can serve as a simple database.
- An array of `part` structures capable of storing information about 100 parts:

```
struct part inventory[100];
```

Arrays of Structures

- Accessing a part in the array is done by using subscripting:

```
print_part(inventory[i]);
```

- Accessing a member within a `part` structure requires a combination of subscripting and member selection:

```
inventory[i].number = 883;
```

- Accessing a single character in a part name requires subscripting, followed by selection, followed by subscripting:

```
inventory[i].name[0] = '\0';
```

Structures as Arguments

- Functions may have structures as **arguments** and **return values**.
- A function with a structure argument:

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

- A call of `print_part`:

```
print_part(part1);
```

Structures as Arguments

- A function that returns a `part` structure:

```
struct part build_part(int number,
                       const char *name,
                       int on_hand)
{
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

- A call of `build_part`:

```
part1 = build_part(528, "Disk drive", 10);
```

Structures as Arguments

- Passing a structure to a function and returning a structure from a function both require making a **copy of all members in the structure**.
- To avoid this overhead, it's sometimes advisable to **pass a pointer to a structure** or return a pointer to a structure.
- Chapter 17 gives examples of functions that have a pointer to a structure as an argument and/or return a pointer to a structure (see tutorial).

Pointers to Structures

- When we create a new part, we'll need a variable that can point to the new part temporarily:

```
struct part *new_part;
```

- We'll use `malloc` to allocate memory for the new node, saving the return value in `new_part`:

```
new_part = malloc(sizeof(struct part));
```

- `new_part` now points to a block of memory just large enough to hold a `part` structure:

Pointers to Structures

- Next, we'll store data in the `number` member of the new node:

```
(*new_part).number = 10;
```

- The parentheses around `*new_part` are mandatory because the `.` operator would otherwise take precedence over the `*` operator.

The `->` Operator

- Accessing a member of a structure using a pointer is so common that C provides a special operator for this purpose.
- This operator, known as ***right arrow selection,*** is a minus sign followed by `>`.
- Using the `->` operator, we can write

  ```
  new_part->number = 10;
  ```

  instead of

  ```
  (*new_part).number = 10;
  ```

The -> Operator

- The `->` operator produces an **lvalue**, so we can use it wherever an ordinary variable would be allowed.
- A `scanf` example:

```
scanf("%d", &new_part->number);
```

- The `&` operator is still required, even though `new_part` is a pointer.

Chapter 17

Dynamic Memory Management and Linked Lists

Dynamic Storage Allocation

- C’s data structures, including arrays, are normally fixed in size.
- Fixed-size data structures can be a problem, since we’re forced to choose their sizes when writing a program.
- Fortunately, C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution.
- Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

Dynamic Storage Allocation

- Dynamic storage allocation is used most often for **strings**, **arrays**, and **structures**.
- Dynamically allocated structures can be linked together to form **lists**, **trees**, and **other data structures**.
- Dynamic storage allocation is done by calling a memory allocation function.

Memory Allocation Functions

- The `<stdlib.h>` header declares three memory allocation functions:

  `malloc`—Allocates a block of memory but doesn’t initialize it.

  `calloc`—Allocates a block of memory and clears it.

  `realloc`—Resizes a previously allocated block of memory.
- These functions return a value of type `void *` (a “generic” pointer).

Null Pointers

- If a memory allocation function can't locate a memory block of the requested size, it returns a ***null pointer.***
- A null pointer is a special value that can be distinguished from all valid pointers.
- After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

Null Pointers

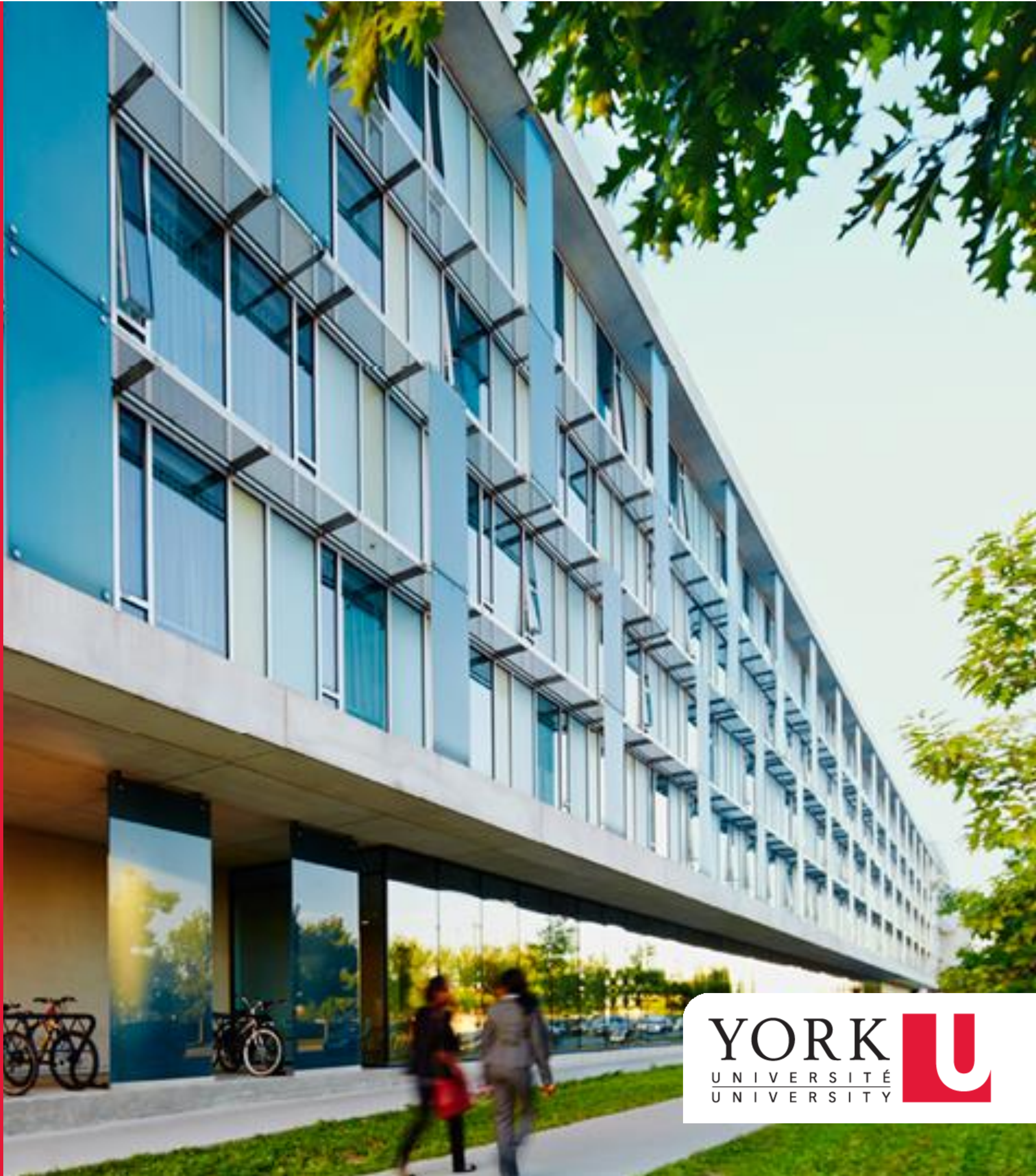
- An example of testing `malloc`'s return value:

```
p = malloc(10000);
if (p == NULL) {
  /* allocation failed; take appropriate action */
}
```

- `NULL` is a macro (defined in various library headers) that represents the null pointer.
- Some programmers combine the call of `malloc` with the `NULL` test:

```
if ((p = malloc(10000)) == NULL) {
  /* allocation failed; take appropriate action */
}
```

Dynamically Allocated Strings



Dynamically Allocated Strings

- Dynamic storage allocation is often useful for working with strings.
- Strings are stored in character arrays, and it can be hard to anticipate how long these arrays need to be.
- By allocating strings dynamically, we can postpone the decision until the program is running.

Using **`malloc`** to Allocate Storage for a String

- Prototype for the `malloc` function:

  ```
  void *malloc(size_t size);
  ```
- `malloc` allocates a block of `size` bytes and returns a pointer to it.
- `size_t` is an unsigned integer type defined in the library.

YORK U UNIVERSITÉ UNIVERSITY

Using **`malloc`** to Allocate Storage for a String

- A call of `malloc` that allocates memory for a string of `n` characters:

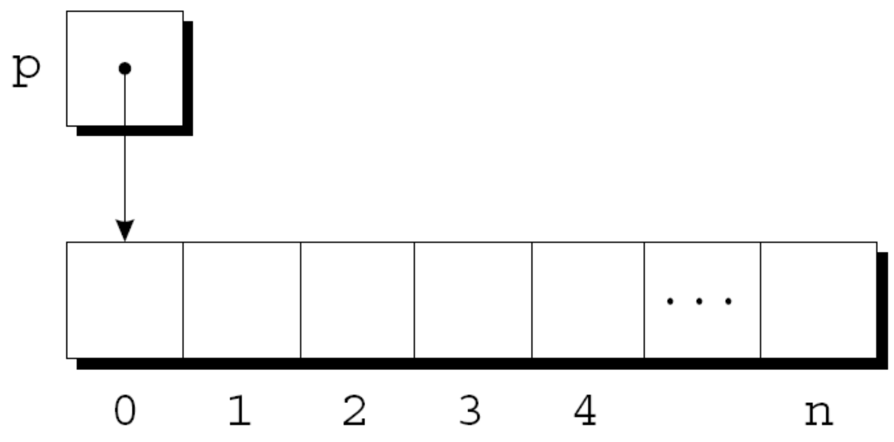
```
char * p;
p = malloc(n + 1);
```

- Some programmers prefer to cast `malloc`'s return value, although the cast is not required:

```
p = (char *) malloc(n + 1);
```

Using `malloc` to Allocate Storage for a String

- Memory allocated using `malloc` isn't cleared, so `p` will point to an uninitialized array of `n` + 1 characters:

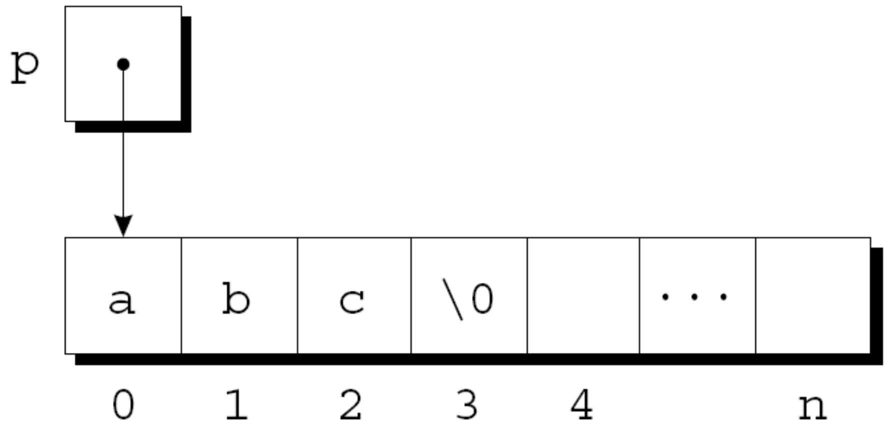


Using **`malloc`** to Allocate Storage for a String

- Calling `strcpy` is one way to initialize this array:

```
strcpy(p, "abc");
```

- The first four characters in the array will now be `a`, `b`, `c`, and `\0`:



Example function: concat

- Dynamic storage allocation makes it possible to write functions that return a pointer to a “new” string.
- Consider the problem of writing a function that concatenates two strings without changing either one.
- The function will measure the lengths of the two strings to be concatenated, then call `malloc` to allocate the right amount of space for the result.

Example function: concat

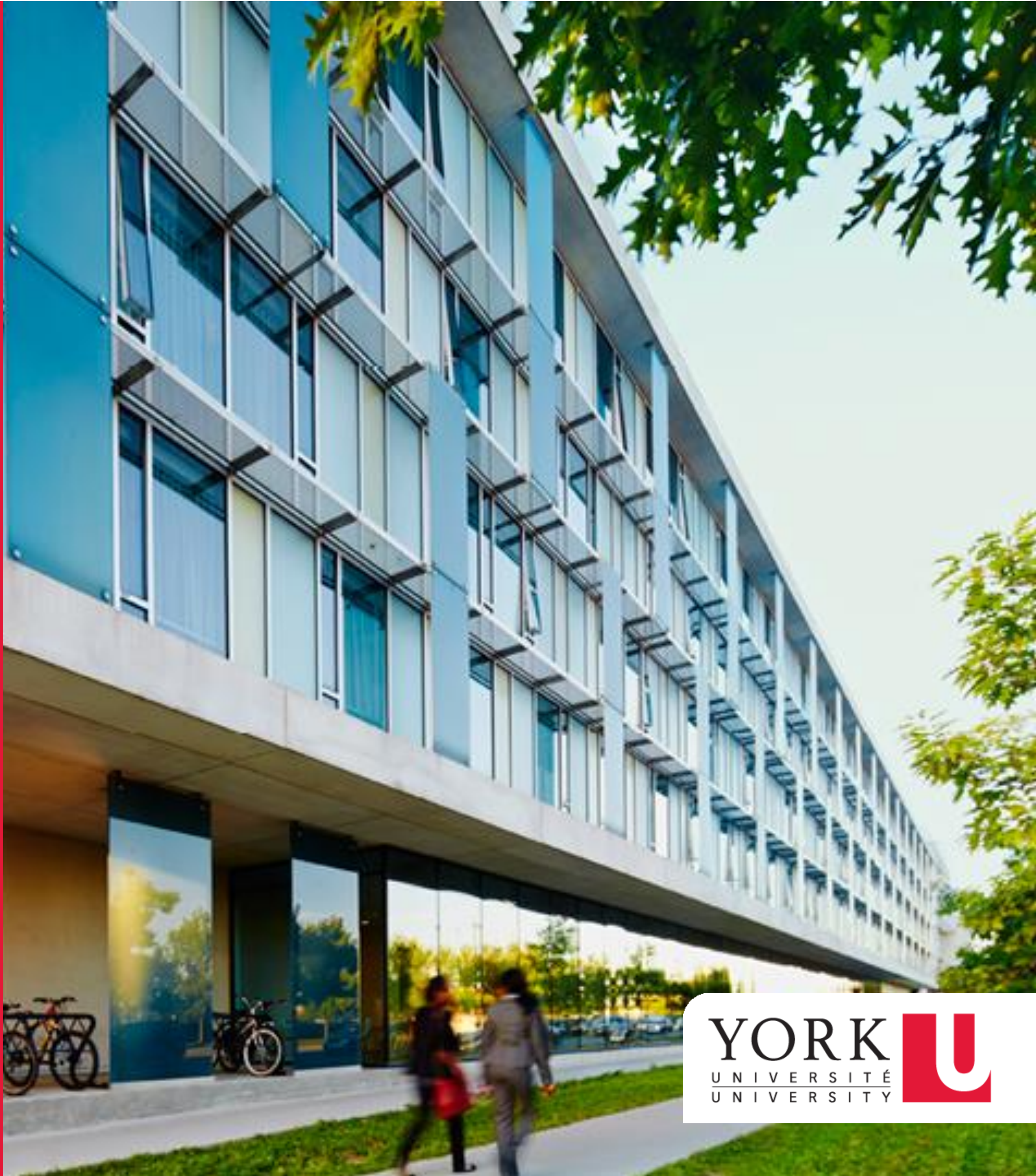
```
char *concat(const char *s1, const char *s2)
{
  char *result;

  result = malloc(strlen(s1) + strlen(s2) + 1);
  if (result == NULL) {
    printf("Error: malloc failed in concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

Example function: concat

- A call of the `concat` function:

  ```
  p = concat("abc", "def");
  ```
- After the call, `p` will point to the string `"abcdef"`, which is stored in a dynamically allocated array.

Dynamically Allocated Arrays



Dynamically Allocated Arrays

- Dynamically allocated arrays have the same advantages as dynamically allocated strings.
- The close relationship between arrays and pointers makes a dynamically allocated array as easy to use as an ordinary array.
- Although `malloc` can allocate space for an array, the `calloc` function is sometimes used instead, since it initializes the memory that it allocates.
- The `realloc` function allows us to make an array “grow” or “shrink” as needed.

Using `malloc` to Allocate Storage for an Array

- Suppose a program needs an array of `n` integers, where `n` is computed during program execution.
- We'll first declare a pointer variable:

```
int *a;
```

- Once the value of `n` is known, the program can call `malloc` to allocate space for the array:

```
a = malloc(n * sizeof(int));
```

- Always use the `sizeof` operator to calculate the amount of space required for each element.

Using `malloc` to Allocate Storage for an Array

- We can now ignore the fact that `a` is a pointer and use it instead as an array name, thanks to the relationship between arrays and pointers in C.
- For example, we could use the following loop to initialize the array that `a` points to:

```
for (i = 0; i < n; i++)
  a[i] = 0;
```

The **calloc** Function

- The `calloc` function is an alternative to `malloc`.
- Prototype for `calloc`:

```
void *calloc(size_t nmemb, size_t size);
```

- Properties of `calloc`:
  - Allocates space for an array with `nmemb` elements, each of which is `size` bytes long.
  - Returns a null pointer if the requested space isn't available.
  - Initializes allocated memory by setting all bits to 0.

The **`calloc`** Function

- A call of `calloc` that allocates space for an array of `n` integers:

```
a = calloc(n, sizeof(int));
```

The `realloc` Function

- The `realloc` function can resize a dynamically allocated array.
- Prototype for `realloc`:

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

- `ptr` must point to a memory block obtained by a previous call of `malloc`, `calloc`, or `realloc`.
- `size` represents the new size of the block, which may be larger or smaller than the original size.

The **`realloc`** Function

- Properties of `realloc`:
  - When it expands a memory block, `realloc` doesn’t initialize the bytes that are added to the block.
  - If `realloc` can’t enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged.
  - If `realloc` is called with a null pointer as its first argument, it behaves like `malloc`.
  - If `realloc` is called with 0 as its second argument, it frees the memory block.

The **`realloc`** Function

- We expect `realloc` to be reasonably efficient:
  - When asked to reduce the size of a memory block, `realloc` should shrink the block "in place."
  - `realloc` should always attempt to expand a memory block without moving it.
- If it can't enlarge a block, `realloc` will allocate a new block elsewhere, then copy the contents of the old block into the new one.
- Once `realloc` has returned, be sure to update all pointers to the memory block in case it has been moved.

YORK UNIVERSITÉ UNIVERSITY

Deallocating Storage

- `malloc` and the other memory allocation functions obtain memory blocks from a storage pool known as the ***heap.***
- Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.
- To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space.

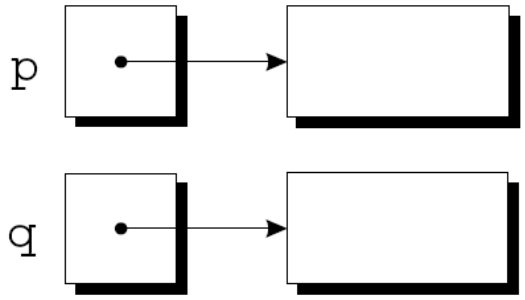
YORK UNIVERSITÉ UNIVERSITY

Deallocating Storage

- Example:

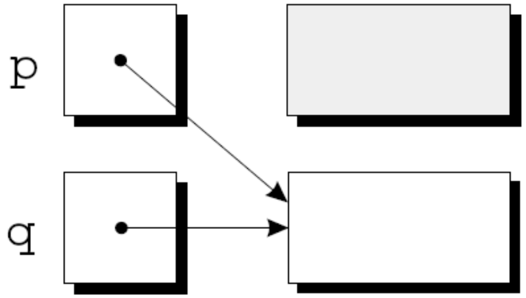
```
p = malloc(…);
q = malloc(…);
p = q;
```

- A snapshot after the first two statements have been executed:



Deallocating Storage

- After `q` is assigned to `p`, both variables now point to the second memory block:



- There are no pointers to the first block, so we'll never be able to use it again.

Deallocating Storage

- A block of memory that's no longer accessible to a program is said to be ***garbage.***
- A program that leaves garbage behind has a ***memory leak.***
- Some languages provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't.
- Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

The `free` Function

- Prototype for `free`:

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

- `free` will be passed a pointer to an unneeded memory block:

```
p = malloc(…);
q = malloc(…);
free(p);
p = q;
```

- Calling `free` releases the block of memory that `p` points to.

The “Dangling Pointer” Problem

- Using `free` leads to a new problem: ***dangling pointers.***
- `free(p)` deallocates the memory block that `p` points to, but doesn’t change `p` itself.
- If we forget that `p` no longer points to a valid memory block, chaos may ensue:

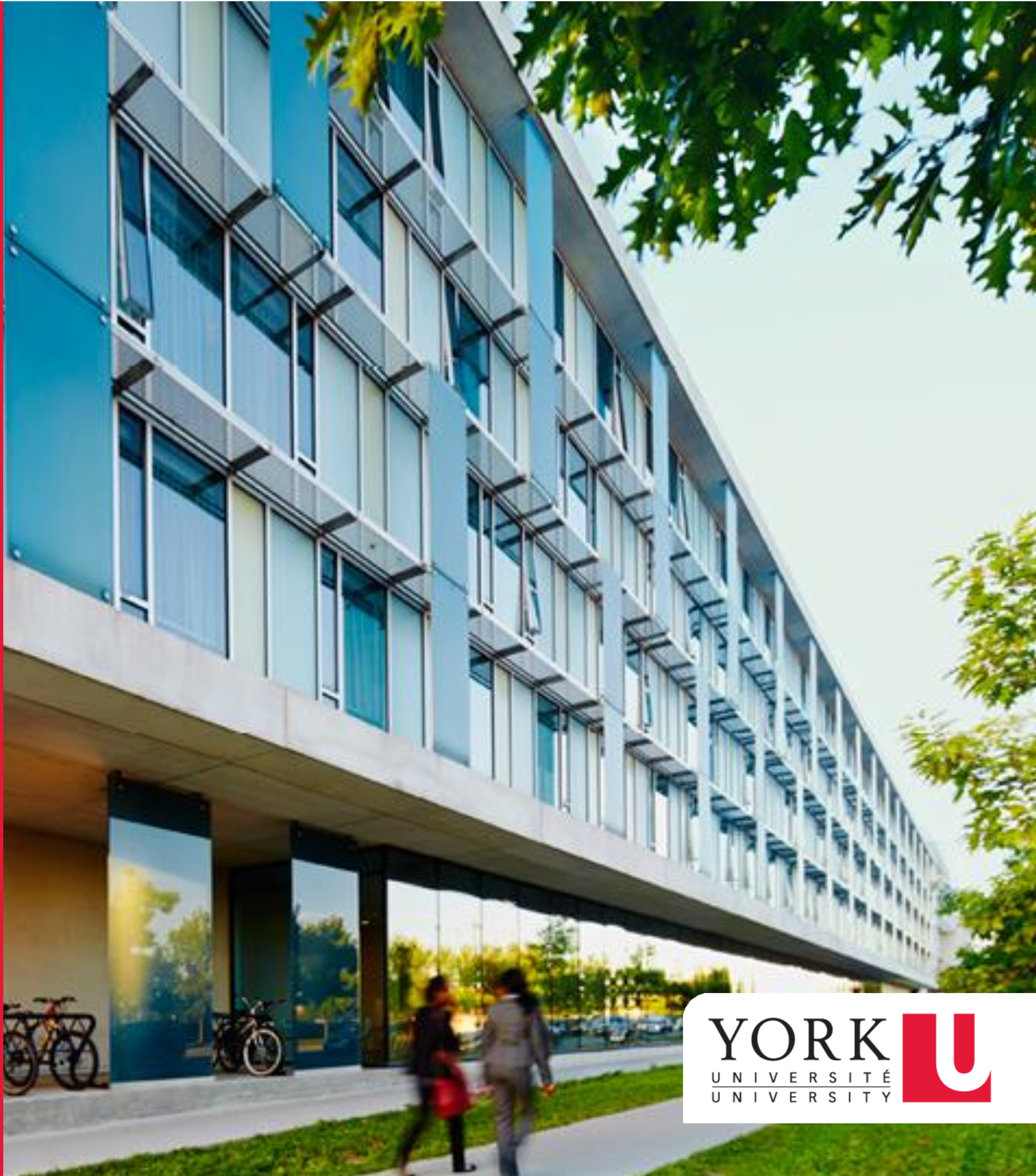
```
char *p = malloc(4);
…
free(p);
…
strcpy(p, "abc");   /*** WRONG ***/
```

- Modifying the memory that `p` points to is a serious error.

The "Dangling Pointer" Problem

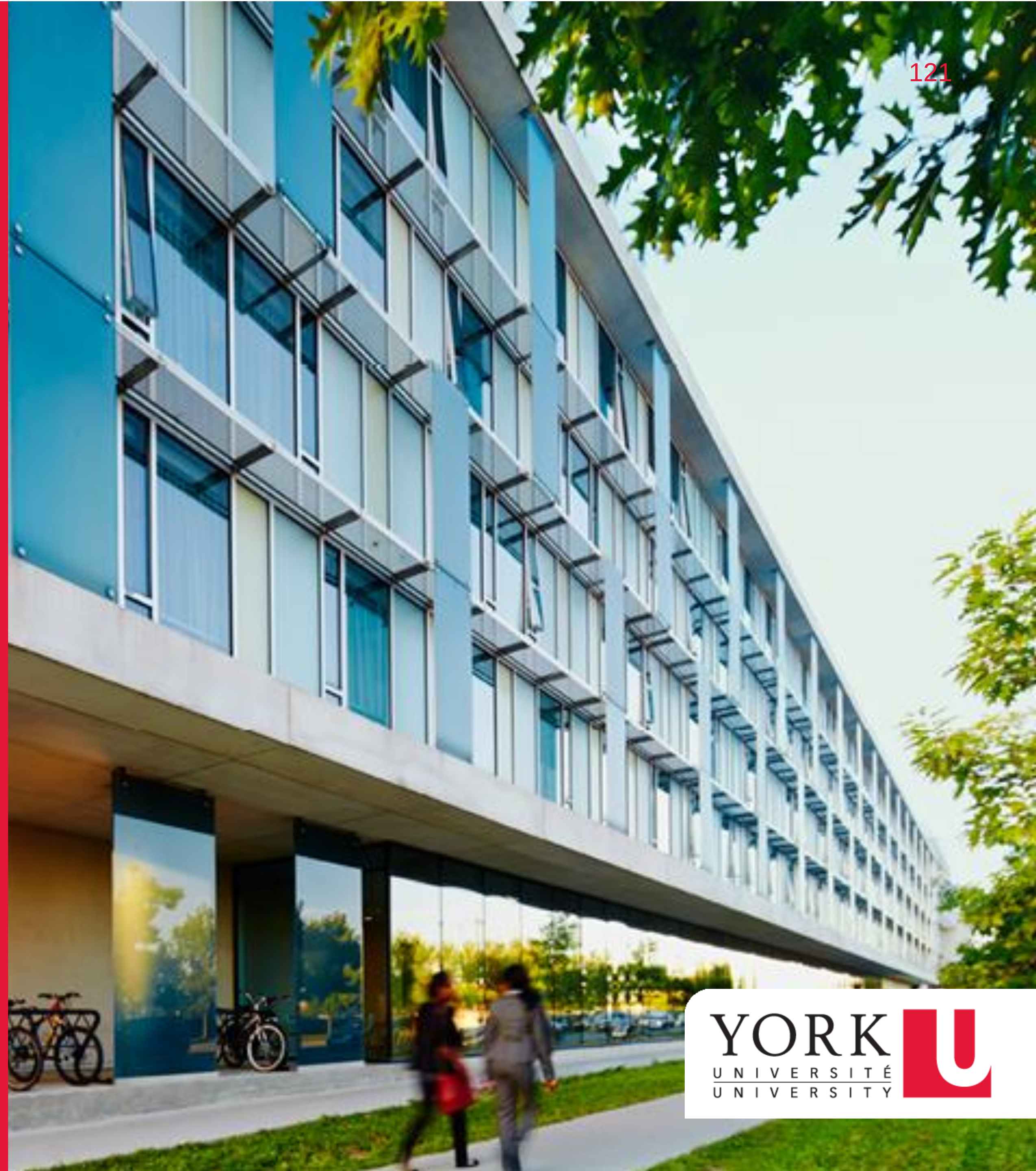
- Dangling pointers can be hard to spot, since several pointers may point to the same block of memory.
- When the block is freed, all the pointers are left dangling.

Linked Lists



Makefile

(Extra Topic -
No exam material)



Makefiles

Makefiles were originally designed to support separate compilation of C files.

```
FLAGS = -g

all: query printlog

query: query.o message.o queue.o
	gcc ${FLAGS} -o $@ $^

printlog: printlog.o message.o queue.o
	gcc ${FLAGS} -o $@ $^

# Separately compile each C file
%.o : %.c message.h
	gcc ${FLAGS} -c $<

clean :
	-rm *.o query
```

Terminology

Target

Prerequiste(s)

```
reverse : reverse.c
	gcc -Wall -o reverse reverse.c
```

Rule

Actions(s)

- May be many prerequisites
- Rule may have many actions (one per line)

Running make

- `make`
  - with no options looks for a file called Makefile, and evaluates the **first rule**
- `make query`
  - Looks for a file called Makefile and looks for a rule with the target `query` and evaluates it

How it works

- Make looks at the target and when its prerequisites were last modified
  - It assumes targets are files and checks the dates of the files
- Make does nothing…
  - If the target exists, and
  - Is more recent than all its prerequisites
- Make executes the actions…
  - If the target doesn't exist, or
  - If any prerequisite is more recent than the target

YORK UNIVERSITÉ UNIVERSITY

Variables

```
CFLAGS= -g -Wall
reverse: reverse.c
	gcc ${CFLAGS} -o reverse reverse.c
```

Make defines **variables** to represent parts of rules

`$@`	Target
$<	First prerequisite
$?	All out of date prerequisites
$^	All prerequisites

Pattern rules

Most files are compiled the same way

- We can write a pattern rule for the general case

```
%.o: %.c
	gcc ${CFLAGS} -c $<
```

- Use % to mark the stem of the file's name
- Like using * in commands in Unix

-c flag in gcc does compilation of file without linking

Multiple Targets and Phony Targets

- Often you want one command to build a number of other targets

```
all: query printlog
printlog: …
   …
query: …
```

- Or targets aren't building anything

```
clean:
   rm *.o query printlog
```