CS 3411 Systems Programming

Department of Computer Science
Michigan Technological University

C vs. C++
C Programming Language

- C may be a slightly different language than what you’re used to!
- Major differences from C++ include:
  1. No classes in C!
     - C is a procedural language.
  2. // comments not in C; must use non-nested /* . . . */
     - The newer C99 standard supports these, but they are not in the original specification.
  3. Function prototypes are not necessary in C.
  4. C is more lax in typing arguments to functions
  5. C passes values to functions using pass-by-value, there is no reference parameters.
     - References can be emulated by passing pointer by value.
  6. C has no stream based I/O, uses the relatively crude stdio.
  7. No new/delete style allocations in C; dynamic allocation through library functions required.
C Declarations

- C++ allows constructs like:
  ```
  for (int i = 0; i < array_size; i++)
      ...
  ```

- All declarations are global or at the beginning of a compound statement.
  ```
  somefunction(a, b)
  int a; char b;
  {
      int i;
      ...  
      for (i=0; i < array_size; i++) {
        ...
      }
  }
  ```
Consider the following C++ program, `test1.cc`:

```cpp
#include <iostream>
using namespace std;

int main() {
    int x;
    x = 3;
    foo(x);
}

void foo(int arg) {
    cout << arg << "\n";
}
```

Trying to compile it with `g++ -Wall test1.cc` gives us:

```plaintext
test1.cc: In function 'int main()':
test1.cc:9: error: 'foo' was not declared in this scope
```
Function Prototypes in C

- This is easy to fix!

```cpp
#include <iostream>
using namespace std;
void foo(int arg);
int main() {
    int x;
    x = 3;
    foo(x);
}
void foo(int arg) {
    cout << arg << " \n ";
}
```

- When the prototype is inserted, the code compiles and runs normally.
- C++ requires complete type info about a function and its arguments before its use.
- Function prototypes provide that info.
Function Prototypes in C

- C does not require explicit prototypes!

```c
#include <stdio.h>
main() {
    int x;
    x = 3;
    foo(x);
}
void foo(int arg) {
    printf("%d\n", arg);
}
```

- Compiling and running it gives the following warnings:

```
test1.c:8: warning: conflicting types for 'foo'
test1.c:6: note: previous implicit declaration of 'foo' was here
```

- But it still runs!
The original C specifications assume any undeclared function returns an `int` and does no checking of its arguments!

This may lead to interesting bugs.
No Reference Parameters in C!

▶ An example of reference parameters in C++ (*ref.cc*):

```cpp
#include <iostream>
using namespace std;

void swap(int& first, int& second) {
    int temp;
    temp = first; first = second; second = temp;
}
main() {
    int x, y;
    x = 3; y = 5;
    cout << "Before: \nx=" << x << " \ny=" << y << "\n";
    swap(x, y);
    cout << "After: \nx=" << x << " \ny=" << y << "\n";
}
```

▶ This gives us the output:

```
Before: x=3 y=5
After: x=5 y=3
```
No Reference Parameters in C!

- If we try the equivalent in C (`ref.c`):

```c
#include <stdio.h>
void swap(int first, int second) {
    int temp;
    temp = first; first = second; second = temp;
}
main() {
    int x, y;
    x = 3; y = 5;
    printf("Before: x=%d y=%d \n", x, y);
    swap(x, y);
    printf("After: x=%d y=%d \n", x, y);
}
```

- We get the output:

```
Before: x=3 y=5
After: x=3 y=5
```

- We use a workaround for this in C: pointer parameters.
Review of Pointers in C

- If \( x \) is an \texttt{int}, then \&\( x \) is the \textit{address of} \( x \).
- Pointers are declared like so:
  
  ```
  \texttt{int} \,*px;  
  px = \&x;  /* px is a pointer to x */
  ```

- The name of an array of type \( T \) is the address of the first byte of storage for the array, and may be assigned to a pointer of type \( (T *) \).
- The \& operator may be applied to variables or array elements.
- ...But not to expressions!
  - \&(x+1) and \&3 not allowed.
Review of Pointers in C

- `int *` is like a new type - pointer to `int`.
- If part of an expression, `*` is called the *indirection operator*.  
  ```c
  int x, y, *px;
  px = &x;
  y = *px; /* same as y = x; */
  ```
- `*` treats its argument as the address of the target, and accesses that address to fetch the contents.
Pointers can be used in expressions! (*px is okay where x is okay)

- \( y = *px + 1; \) /* \( y = x + 1; \) */
- \( printf("%d\n", *px); \) /* \( printf("%d\n", x); \) */
- \( d = sqrt((\textbf{double}) *px); \) /* \( d = sqrt((\textbf{double}) *px); \) */
- \( *px = 0; \) /* \( x = 0; \) */
- \( *px += 1; \) /* \( x += 1; \) */
- \( (*px)++; \) /* parantheses necessary */

Pointer Assignments:

```c
int x, *px, *py;
px = &x;
py = px;
*py = 0; /* x = 0 */
```
Assume we have the following declaration:

```c
foo *ptr; /* ptr is addr of a "foo" */
```

What would we get from the following operations?

- `ptr = ptr + 1; /* or ptr++; */`
- `ptr = ptr + 6;`
- `*ptr++;`
- `(*ptr)++;`
References in C

- We can achieve the *effect* of call by reference by passing pointers by *value*.

```c
#include <stdio.h>

void swap(int *first, int *second) {
    int temp;
    temp = *first; *first = *second; *second = temp;
}

main() {
    int x, y;
    x = 3; y = 5;
    printf("Before: x=%d y=%d\n", x, y);
    swap(&x,&y);
    printf("After: x=%d y=%d\n", x, y);
}
```

- This gives us the output:

  Before: x=3 y=5
  After: x=5 y=3

- Which is what we want!
Examples of Pointer Use: Strings in C

- There is no string data type in C.
- Instead, a string is assumed to be a sequence of `char` terminated by a zero byte.
- A `char *` is generally used as a string; just a pointer to the first char in the zero-terminated sequence of chars.
- Careful when *declaring* a string:

```c
char *STR; /* Only memory allocated is to pointer variable */
char str[20]; /* 20 bytes allocated to hold contents of string */
```