# Introduction to C

This is not C++

# Overview

From C++ to C

C++ is a superset of C

- Everything from C is in C++, but extra stuff was added
- Most of what you know about C++ remains the same in C.

# Overview

C is a high-level assembly language:

- It is high-level since it has major features of a modern high-level programming language: data types & structures, control structures and advanced I/O
- It is an assembly language since by the overall philosophy the language is close to the assembly language level
    - Assembly programing can be included in C programs

Virtually anything that can be done in assembly can be done in C.

# Procedural Programming Language

- Never again will an entire course be used to teach a programming language
    - i.e. learning Java for 2631
- Therefore, you need to understand what elements compose a programming language
    - This allows you to move between programming languages more easily

# Elements of Procedural Programming Language

There are 6 elements of a procedural programming language:

1. Program Structure
2. Data Specifications
3. Control Structures
4. Data Manipulation
5. Sub-programs
6. Input/Output and Files

# Program Structure

- Separators - the item(s) that are between individual language tokens
  - 68000 Assembly - separators are whitespace and the comma
  - C - no separators, token specified so no need for separators
- Delimiters
  - 68000 Assembly -  each instruction on its own line (\n char)
  - C - semicolons and braces ({}) define parts of program
- Fixed/Free Format
  - C is free format -place stuff where  you want!

# Program Structure

Separators (Continued)

- Order
  - Definition - The place where variable is created (i.e. storage allocation)
  - Declaration - where the nature of a variable/function is given
    - Examples: includes, externs, defines, declarations, definitions, code
    - Note: includes & defines are NOT really C

# Data Specification

- Data declaration and attributes
  - implicit/explicit
  - initialization
  - ALL variables must be declared at the start of a block
    - immediately after the {
- Predefined data types
  - As per C++: int, long, float, double, char, pointers
  - No booleans: use ints with 0/-1 as values

# Data Specifications

- Predefined data structures
  - C - Arrays
- User-defined data types
  - enumerated types
  - structs (very different than in C
  - complex structures (e.g. arrays of records)
- classes, objects, messages - **DO NOT EXIST!**

# Pointers

- notation and operators, * & and -> are identical
- methods of allocation and de-allocation are significantly different
  - C++ - new and delete are built in keywords, i.e. part of the language
  - C corresponding operations are not built in, but rather come in the stdlib.h library

# Pointers

- allocation can be done in multiple ways
  - **(void *) malloc (long size)**
  - function takes size of desired memory and return a void* pointer
  - pointer **MUST** be cast to the appropriate type
  - e.g. int *p;
    p = (int *) malloc (sizeof(int));
- If malloc is unable to allocate a block of the requested size the returned pointer is NULL.
- De-allocation is done by the **free(ptr)** function.

# Structs

C++ Declaration:

```
struct <id> {
    fields;
};
```

The <id> can then be used as the type name.

# Structs

This is NOT true in C, whose declaration is:

```
struct <tag_name> {
    fields;
} <struct_name>;
```

the tag_name and struct_name are both optional, but one MUST appear.

# Structs

The **tag_name** has two purposes:

1. Self reference – this is typically used for linked structures since in C an item must be completely declared before it can be used. If this exception weren't included it would be impossible to have a link field.
2. As a portion of the type name – however, this is truly the ugliest way of doing this, as the type name becomes struct <tag_name>
    - ex. struct student joe_student;

# Structs

- The **struct_name** is the name of a variable of this type
- If done outside of a function this declares a single global variable of this type
- If done inside a function it is a completely local variable that can't be passed
    - Most structs are declared globally to be able to pass them around

# Structs

In order to declare a type of struct the typedef construct must be used. The format of a typedef is:

```
typedef actual type new name;
ex. typedef unsigned char BYTE;
```

for a struct:

```
typedef struct
{
    fields;
} <type_name>;
```

<type_name> can now be used as a variable type

# Control Structures & Data Manipulation

- Control structures (i.e. if, for, while, switch, etc.)
  - identical to C++
- Data Manipulation
  - relational operators
  - arithmetic operators
  - other operators (e.g. logical, bit, string, etc.)
  - built-in functions

  Often give hints on purpose, focus, best use of language

# C Operator Precedence

| Category | Operator | Associativity |
|---|---|---|
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)* & sizeof | Right to left |
| Multiplicative | * / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | \| | Left to right |
| Logical AND | && | Left to right |
| Logical OR | \|\| | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= *= /= %=>>= <<= &= ^= \|= | Right to left |
| Comma | , | Left to right |

# Sub-programs

This is a reference to functions:

- Existence
- Parameter passing (different to C++)
- External modules
- Scope
- Re-entrancy

Yes to all!!

# Sub-programs

- Identical in nature to C++
- Same two return categories
  - value returning
  - void
- Same two methods of parameter passing
  - by value
  - by reference

# Pass By Reference - C++

- Pass by reference determined by the function header
  - Specified by inclusion (or omission) of the & operator
    - e.g. `void copy(int x, int &y)`
  - No indication of pass by reference in function call
    - e.g. `copy(a, b);`
  - Within the body no difference in how variables are used
    - e.g. `y = x;`

# Pass By Reference - C

- Pass by reference is done using pointers
  - All reference parameters are pointer in the function header
    - e.g. `void copy(int x, int *y)`
  - Pointers are initialized by taking the address of a variable
    - Done in the function call
    - e.g. `copy(a, &b);`
  - Within the function, must dereference the pointer
    - e.g. `*y = x;`

# Example

```
void swap (int *one, int *two) {
    int temp;
    temp = *one;
    *one = *two;
    *two = temp;
}
// Called by:
swap (&a, &b);
```

# Pass by Reference - Extra

- With multiple levels of functions with by reference parameters:
  - You must clearly understand what is being passed in order to know whether to pass the value *ref_parameter or the address ref_parameter.

# Arrays as Parameters

- Are always passed by reference
- As such it is common to see code that treats an array parameter as a pointer, this is especially true for character arrays.
- The ability to treat an array as either an array or a pointer, upon which pointer math is possible, is ONLY allowed in a function. Since the array is passed as a pointer.

# Array as Parameter Example

```
int strcmp (char s1[], char s2[]) {

    while (*s1 != ‘\0’ && *s1 == *s2) {
        s1++;
        s2++;
    }

    return (*s1 - *s2);

}
```

# Input/Output and Files

- Interactive I/O
- Types of files supported

This is vastly different than C++.

Not really an issue for 2655 since I/O that mimics Assembly Language will be provided.

However, for future reference

# I/O Overview

- I/O functions have two versions: those that use stdin (kbd) and stdout (screen) and those that use a user specified file. Will focus on former initially
- Two variants of I/O:
  1. Type specified by the operation (character, string, etc.)
  2. Formatted I/O

# Character I/O

```
int getchar();

int putchar(int ch);
```

- These read and write from stdin and stdout
- getchar reads all valid input characters including whitespace
- putchar writes the specified character to stdout

# Character I/O

Functions `getchar` and `putchar` are virtually identical to assembly character I/O.

Notes:

1. Implicit type casting in putchar is fine.
2. **MUST** cast the return value from getchar.

Example:

```
char ch;
ch = (char) (getchar());
```

# String I/O

```
char *gets (char *s);
int puts(const char *s);
```

- `puts` writes s to stdout and appends a newline to the end of the string. If an error occurs it returns EOF; otherwise it returns a non-negative value.
- `gets` stores the input in s and terminates reading on a newline or EOF. The newline is NOT stored in s, but a null terminator (‘\0’) is added. The function returns either s or null pointer if nothing is read.
- The use of gets is NOT recommended since this function will allow the input length to exceed the length of the supplied storage space. An alternative string input function exists that is better.

# Formatted Input

- Printf – this function converts any type of variable into character strings and outputs then to stdout

  ```
  int printf (control_string, arguments);
  ```

- The control_string is require. It is a string which can either be a string constant or a string variable. It consists of normal characters and conversion specifiers
- Normal characters are directly output, ex. “hello world\n”, control characters are prefixed with the slash

# Formatted Input

- Conversion specifiers start with a % sign, followed by zero or more flags and terminated by a conversion character
- Flags allow variations on the various conversions
- Arguments:
  - One argument for each conversion specifier in the control_string
  - An argument can be either a constant or a variable
  - The order of arguments should correspond to the order of the specifiers

# Formatted Input

- Arguments:
  - If the argument does not correspond to the conversion specifier the function will attempt to convert the argument to the specified type
  - Conflicts can be dangerous
    - ex. Specifying %s and supplying an int will cause a crash.
  - Too few arguments – function ignores extra conversion specifiers.
  - Too many arguments – function ignores extra args.
  - Returns EOF if an output error occurs; otherwise returns # of characters output

# Control String Values for printf

**Table 15–7 Output conversion specifications**

| Conversion | Defined flags - + # 0 *space* | Size modifier | Argument type | Default precision[a] | Output |
|---|---|---|---|---|---|
| `d, i`[b] | - + 0 *space* | *none* | `int` | 1 | dd…d |
| | | `h` | `short` | | -dd…d |
| | | `l` | `long` | | +dd…d |
| `u` | - + 0 *space* | *none* | `unsigned int` | 1 | dd…d |
| | | `h` | `unsigned short` | | |
| | | `l` | `unsigned long` | | |
| `o` | - + # 0 *space* | *none* | `unsigned int` | 1 | oo…o |
| | | `h` | `unsigned short` | | 0oo…o |
| | | `l` | `unsigned long` | | |
| `x, X` | - + # 0 *space* | *none* | `unsigned int` | 1 | hh…h |
| | | `h` | `unsigned short` | | 0xhh…h |
| | | `l` | `unsigned long` | | 0Xhh…h |
| `f` | - + # 0 *space* | *none* | `double` | 6 | d…d.d…d |
| | | `L` | `long double` | | -d…d.d…d |
| | | | | | +d…d.d…d |
| `e, E` | - + # 0 *space* | *none* | `double` | 6 | d.d…de+dd |
| | | `L` | `long double` | | -d.d…dE-dd |
| `g, G` | - + # 0 *space* | *none* | `double` | 6 | *like* `e, E`, |
| | | `L` | `long double` | | *or* `f` |
| `c` | - | *none* | `int` | 1 | c |
| | | `l`[c] | `wint_t` | | |
| `s` | - | *none* | `char *` | ∞ | cc…c |
| | | `l`[c] | `wchar_t *` | | |
| `p`[b] | *impl. defined* | *none* | `void *` | 1 | *impl. defined* |
| `n`[b] | | *none* | `int *` | *n/a* | *none* |
| | | `h` | `short *` | | |
| | | `l` | `long *` | | |
| `%` | | *none* | *none* | *n/a* | % |

[a] Default precision, if none is specified.
[b] Available in ISO C; may be rare elsewhere. The conversions `i` and `d` are equivalent on output.

# Control String Details for printf

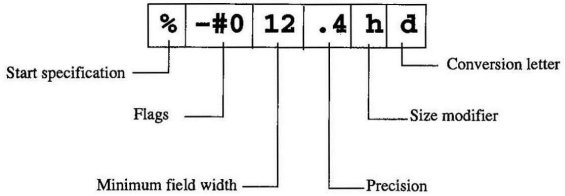
### 15.11.2 Conversion Specifications

In what follows, the terms "characters," "letters," etc. are to be understood as normal characters or letters (bytes) in the case of the **`printf`** functions, and wide characters or letters in the case of the **`wprintf`** functions. For example, in **`wprintf`**, conversion specifications begin with the wide-character percent sign, **`%`**.

A conversion specification begins with a percent sign character, **`%`**, and has the following elements in order:

1. Zero or more *flag characters* (**`-`**, **`+`**, **`0`**, **`#`**, or space), which modify the meaning of the conversion operation.
2. An optional *minimum field width*, expressed as a decimal integer constant.
3. An optional *precision specification*, expressed as a period optionally followed by a decimal integer.
4. An optional *size specification*, expressed as one of the letters **`l`**, **`L`**, or **`h`**.
5. The *conversion operation*, a single character from the set **`c`**, **`d`**, **`e`**, **`E`**, **`f`**, **`g`**, **`G`**, **`i`**, **`n`**, **`o`**, **`p`**, **`s`**, **`u`**, **`x`**, **`X`**, and **`%`**.

The conversion letter terminates the specification. The size specification letters **`L`** and **`h`**, and the conversion operations **`i`**, **`p`**, and **`n`**, are available in ISO C only. The conversion specification "**`%-#012.4hd`**" is shown below broken into its constituent elements:




The optional flag characters modify the meaning of the main conversion operation:

| | |
|---|---|
| **`-`** | Left-justify the value within the field width. |
| **`0`** | Use 0 for the pad character rather than space. |
| **`+`** | Always produce a sign, either **`+`** or **`-`**. |
| *space* | Always produce either the sign – or a space. |
| **`#`** | Use a variant of the main conversion operation. |

# Formatted Output

- Scanf – this function reads character input from stdin and converts it into typed items
    - Important unlike printf, in the event of an input error scanf immediately terminates.

```
int scanf (control_string, arguments);
```

- Control_string – identical in nature to printf, however, if normal characters occur in the control_string an identical match must occur in the input stream or this is an error
- Normal characters are simply matched and ignored

# Formatted Output

- The inclusion of whitespace in the control_string will cause any and all sequential whitespace to be skipped
    - Most conversion specifiers already do this so this is redundant
    - Useful for the character specifier, but excludes skipping of newlines
- Specifiers are virtually identical to scanf, except %n, which results in an integer number of characters read so far (useful when reading string input)

# Formatted Output

- The flags are very different:
- The * - suppression flag – cause the type of item to be read, but the result is not stored.
    - This is useful when reading formatted data, but only want limited subset of the data.
- A number – maximum field width – this causes the conversion to stop if the field size is reached.
    - The input is NOT flushed to the next newline.

# Formatted Output

- Arguments – these MUST be variables, and in fact must pass the address of the variable since the function must store the result in a location. Omitting the address-of operator on non-string arguments is the most common I/O error.
    - This will typically result in a BUS error.
- Scanf returns EOF if the stream is initially empty or an input failure; otherwise the # of successful conversions (when matching or a matching failure)

# Control String Values for scanf

**Table 15–4 Input conversions (scanf, fscanf, sscanf)**

| Conversion letter | Size specifier | Argument type | Input format |
|---|---|---|---|
| `d` | none | `int *` | [−\|+]dd…d |
| | `h` | `short *` | |
| | `l` | `long *` | |
| `i`[a] | none | `int *` | [−\|+][`0`[`x`]]dd…d[b] |
| | `h` | `short *` | |
| | `l` | `long *` | |
| `u` | none | `unsigned *` | [−\|+]dd…d |
| | `h` | `unsigned short *` | |
| | `l` | `unsigned long *` | |
| `o` | none | `unsigned *` | [−\|+]dd…d[c] |
| | `h` | `unsigned short *` | |
| | `l` | `unsigned long *` | |
| `x` | none | `unsigned *` | [−\|+][`0x`]dd…d[d] |
| | `h` | `unsigned short *` | |
| | `l` | `unsigned long *` | |
| `c` | none | `char *` | a fixed-width sequence of characters; must be multibytes if `l` is used |
| | `l`[e] | `wchar_t *` | |
| `s` | none | `char *` | a sequence of non-whitespace characters; must be multibytes if `l` is used |
| | `l`[e] | `wchar_t *` | |
| `p`[a] | none | `void **` | a sequence of characters such as output with `%p` in `fprintf`. |
| `n`[a] | none | `int *` | none; the number of characters read is stored in the argument |
| | `h` | `short *` | |
| | `l` | `long *` | |
| `f`, `e`, `g` | none | `float *` | any floating-point constant or decimal integer constant, optionally preceded by – or + |
| | `l` | `double *` | |
| | `L`[a] | `long double *` | |
| `[` | none | `char *` | a sequence of characters from a scanning set; must be multibytes if `l` is used |
| | `l`[e] | `wchar_t *` | |

[a] ISO C addition.
[b] The base of the number is determined by the first digits in the same way as for C constants.
[c] The number is assumed to be octal.
[d] The number is assumed to be hexadecimal regardless of the presence of `0x`.
[e] ISO C Amendment 1 addition.

# References

The three tables are from:

C A Reference Manual, 4th Ed.

Samuel P. Harbison & Guy L. Steele Jr.

Prentice Hall, ISBN 0-13-326224-3

- printf p 372
- details p 368
- scanf p 360