# COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/

# Pointers

- Pointers denote addresses in memory
- In C types, the "*" represents the use of a pointer
- Pointers always have a base-type: "`int *`" or "`union foo *`"
- The base-type specifies the type of the data at that location in memory
- Pointers are similar to references in Java

# The Address-Of Operator

- You can obtain the address of a variable using the "&" operator
- The resulting value is always a pointer

Example:

```
int i;
int * i_ptr = &i;
```

# The Address-Of Operator

You can **not** take the address of a constant:

```
int * i_ptr = &5;
```

will not compile!

# The Address-Of Operator

Note that "&" is also used for bitwise-and:

```
int one = 3 & 5;  // bitwise-and
int * ptr = &one; // address-of
```

# The Dereferencing Operator

The * operator is the inverse of the "&" operator:

```
int i = 4;
int * i_ptr = &i;
int j = *i_ptr;    // j = 4
(*i_ptr) = 6;      // i = 6, j = 4
```

# Dereferencing `structs` and `unions`

You can use the `->` operator to dereference a `struct` or `union` if you are also selecting a field:

```
struct foo { int a; int b; };
struct foo * fodder;
fodder = malloc(sizeof(struct foo));
fodder->a = 4;
(*fodder).b = 4;
```

# Returning multiple values using pointers

```
void run(int a1, int a2, int * r1, int * r2) {
  *r1 = a1 + a2;
  *r2 = a1 * a2;
}
int main(int argc, char ** argv) {
  int p, m;
  run(3, 4, &p, &m);
  printf("%d %d\n", p, m);
}
```

# Arrays

- Arrays in C are always from $[0 : n - 1]$ (like in Java)
- C does not support bounds checking on array access
- Results of accessing arrays out of bounds are "undefined"
- Arrays are implemented using pointers

$\Rightarrow$ You will only fully understand them after lecture 6

# Array Types

In some sense, arrays are pointers. Depending on the situation, C uses either:

- `int * a;` // works always
- `int a[];` // for parameters
- `int a[SIZE];` // for stack allocation

# Allocating Arrays: on stack

```
int main(int argc, char * argv[]) {
  int myArray[10];
  int i;
  myArray[0] = 1;
  myArray[1] = 1;
  for (i=2;i<10;i++)
    myArray[i] = myArray[i-1] + myArray[i-2];
  printf("%d\n", myArray[9]);
  return 0;
}
```

# Allocating Arrays: on stack

```
static int * fib() {
  int myArray[10];
  int i;
  myArray[0] = 1;
  myArray[1] = 1;
  for (i=2;i<10;i++)
    myArray[i] = myArray[i-1] + myArray[i-2];
  return myArray; // compiler warning! }
int main(int argc, char * argv[]) {
  int * a = fib();
  printf("%d\n", a[9]); // undefined!
  return 0; }
```

# Allocating Arrays: on the heap

```
static int * fib() {
  int * myArray; int i;
  myArray = malloc(sizeof(int) * 10);
  myArray[0] = 1;  myArray[1] = 1;
  for (i=2;i<10;i++)
    myArray[i] = myArray[i-1] + myArray[i-2];
  return myArray;  }
int main(int argc, char * argv[]) {
  int * a = fib();
  printf("%d\n", a[9]);
  free(a);
  return 0;  }
```

# So what should you use?

Use

- `BASETYPE * a;` // almost always
- `BASETYPE a[]` // only (legal) for function arguments if you want to stress a's nature as an array

`BASETYPE a[SIZE];` is the only legal syntax for stack-allocation.

# Multidimensional Arrays

- `int ** a;`
- `int a[][];`
- `int * a[]; // common for argv`

# Heap-Allocation of Multidimensional Arrays

```
static int ** matrix() {
  int ** ret;  int i;
  ret = malloc(sizeof(int*) * 100);
  for (i=0;i<100;i++)
    ret[i] = malloc(sizeof(int) * 100);
  return ret;  }
int main(int argc, char * argv[]) {
  int ** m;  int i;
  m = matrix();
  for (i=0;i<100;i++) free(a[i]);
  free(a);
  return 0;  }
```

# Heap-Allocation of Multidimensional Arrays

```
/* define IA as array of 100 integers */
typedef int IA[100];
static IA * matrix() {
  IA * ret;  int i;
  ret = malloc(sizeof(IA) * 100);
  return ret;  }
int main(int argc, char * argv[]) {
  IA * m;   int i;
  m = matrix();
  free(m);
  return 0;  }
```

# Array Initialization

```
int primes[] = { 2, 3, 5, 7, 11, 13 };
void print_primes() {
  unsigned int offset;

  for (i=0;i<sizeof(primes)/sizeof(int);i++)
    printf("%d\n", primes[i]);
}
```

# Array Initialization

```
int primes[] = { 2, 3, 5, 7, 11, 13, -1 };
void print_primes() {
  unsigned int offset;

  offset = 0;
  while (primes[offset] != -1)
    printf("%d\n", primes[offset++]);
}
```

# const

`const` is a type modifier used to hint that the memory referred to by a pointer should not be modified.

**Example:**

```
void print_array(const int * arr, unsigned int len) {
  unsigned int i;
  for (i=0;i<len;i++)
    printf("%d ", arr[i]); // ok
  arr[0] = 42; // compiler error
}
```

# C is not type-safe

It is possible to cast a `const` variable:

**Example:**

```
void print_array(const int * arr, unsigned int len) {
  unsigned int i;
  for (i=0;i<len;i++)
    printf("%d ", arr[i]); // ok
  ((int*)arr)[0] = 42; // compiler happy
}
```

# Strings

- Strings in C are arrays of characters (char)
- Strings should always be 0-terminated
- Not all `char*` or `char[]` are strings!
- Programmer is responsible for 0-termination!

$\Rightarrow$ Major source of bugs in C code!

# Strings

What happens if you write `"Hello World"` in C?

- C compiler creates 0-terminated `char` array
- Precise type of the value is `const char *`
- String is allocated in data area; do not modify!
- If the same string occurs multiple times, compiler may alias data area.
- `"Hello World"[4] == 'o'`

# Common String Functions

- size_t strlen(const char *)
- char * strstr(const char * haystack, const char * needle);
- int strcmp(const char * s1, const char * s2);
- char * strcpy(char * dest, const char * src);
- char * strcat(char * dest, const char * src);

**Homework:** study the `man` pages for these functions!

# Example: strdup

```
char * strdup(const char * s) {
  char * ret;
  size_t i;

  ret = malloc(strlen(s) + 1);
  for (i=0;i<strlen(s)+1;i++)
    ret[i] = s[i];
  return ret;
}
```

# Array Initialization Revisited

```
struct Student {
  const char * name;
  unsigned int age;
};
struct Student students[] = {
  { "Alice", 19 },
  { "Bob", 21 },
  { "Carol", 27 },
  { "Dave", 31 },
  { NULL, 0 }
};
```

# Variadic Functions

- Variadic functions take a variable number of arguments
- In Java (since 5.0) implemented using arrays
- More complicated story in C...

# Defining Variadic Functions

- Variadic functions must take at least one normal argument
- Use "..." at the end to specify "0 or more additional arguments"

**Example:**

```
int printf(const char * format, ...);
```

# Accessing the arguments

```
int print_ints(unsigned int number_of_args, ...)
  va_list va;
  unsigned int i;
  va_start(va, number_of_args);
  for (i=0;i<number_of_args;i++)
    printf("%d", va_arg(va, int));
  va_end(va);
}
```

# Fun with Variadic Functions (1/3)

```
int print_stuff(const char * format, ...) {
  va_list va;
  unsigned int i;
  va_start(va, format);
  i = 0;
  while (format[i] != '\0') {
    // ... (see next slide)
  }
  va_end(va);
}
```

# Fun with Variadic Functions (2/3)

```
switch (format[i]) {
case 'i':
  printf("%d", va_arg(va, int)); break;
case 's':
  printf("%s", va_arg(va, const char*)); break;
case 'f':
  printf("%f", va_arg(va, float)); break;
case 'x':
  printf("%4s", va_arg(va, struct Student*)->name);
  break;
default: abort();
}
```

# Fun with Variadic Functions (3/3)

```
int main(int argc, char ** argv) {
  struct Student alice;
  alice.name = "Alice";
  print_stuff("isf", 42, "Test", 4.2);
  print_stuff("xsi", &alice, "Test", 1);
  print_stuff("ifs", 1, 2, 3); /* crash! */
  return 0;
}
```

# printf (1/3)

```
int printf(const char * format, ...);
```

Prints format-string, uses arguments to determine values conversion specifications (which start with %).

**Homework:** Read `man 3 printf`!

# printf (2/3)

- "%d" – print `int` (decimal)
- "%u" – print `unsigned int`
- "%x" – print `unsigned int` (hexadecimal)
- "%o" – print `unsigned int` (octal)
- "%f" – print `double`
- "%s" – print `const char *`

# printf (3/3)

- "%08d" – print 8 characters of `int`, padd with zeros
- "%.3f" – print `double` with 3 decimal places
- "%5s" – print first five characters of `const char *`
- "%.*s" – print `unsigned int` characters of `const char *`
- "%p" – print `void *` (any pointer/address)

# Questions

?