COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/
Functions

Similar to (static) methods in Java without the class:

```c
int f(int a, int b) {
    return a + b;
}
```

C always uses **call-by-value**. The C standard allows the compiler to evaluate arguments in any order.
Any C program has one and only one `main` method:

```c
#include <stdio.h>
int main(int argc, char ** argv)
{
    printf("Hello World!\n");
    return 0;
}
```
Types

- Primitive Types
- Compound Types
- Pointers and arrays (Lectures 4-6)
- Function Types (Lecture 10)
Pointers

- Syntax: BASE-TYPE *  
- Similar to references in Java
Arrays

• Syntax: BASE-TYPE []

• Simple example: int b[5]; b[3] = 42;
Primitive Types

- void
- char
- short int
- int
- long
- long long
- float
- double
void

- Used to indicate “no return value” or “no argument”
- Also used for pointers to data of “unknown” or “arbitrary” type using `void *`
char

- Used to represent characters
- In Java, 16 bits, in C always 8 bits!
- Character-set is not defined!
- Applications most often use either UTF-8 or “system default”
- `char` is also a number, so you can do `'a' + 5`
signed char

- Number between -128 and 127
- Could be identical to char, depending on platform!
- Use if you need a char that is signed (rare) or a signed 7-bit number
**unsigned char**

- Number between 0 and 255
- Could be identical to `char`, depending on platform!
- Use if you need to do arithmetic that relies on `char` being unsigned or unsigned 8-bit number
int

• signed number

• On all modern platforms 32 bit (but used to be 16 for some 80286 implementations)

• Range from $-2^{31}$ to $2^{31} - 1$ (2-complement)

• Always identical to signed int

• unsigned int also exists (very useful!)
**short int**

- signed number, 16 bits
- Range from $-2^{15}$ to $2^{15} - 1$ (2-complement)
- Always identical to signed short and signed short int and short
- unsigned short int also exists (useful)
long

• signed number, number of bits corresponds to address bus width of the platform

• So 32 bit on IA32, 64 bit on IA64 and AMD64

• Useful if you need something of the size of your address space (maximum array length, storing pointers as integers, etc.)

• unsigned long also exists (can be good for array indices in scientific computing)
long long

- signed number, 64 bits
- Range from $-2^{63}$ to $2^{63} - 1$ (2-complement)
- unsigned long long also exists (64-bit, unsigned)
float

• 32-bit floating point number

• Similar to Java’s float

• However, unlike Java, floating point operations are platform-dependent

• On most modern platforms for almost all operations the result is exactly the same as in Java
double

• 64-bit floating point number
User-Defined and Compound Types

- enum
- struct
- union
enum

enum DayOfWeek { MO, TU, WE, TH, FR, SA, SU };

- Introduces both new type enum DayOfWeek and constants MO, TU, WE, TH, FR, SA, SU

- Can be used in switch statement:

```c
switch (dow) {
  case SA: case SU:
    printf("Freedom");
    break;
  default:
    printf("Work");
}
```
enums are ints

enum Coin {
    CENT = 1, NICKEL = 5, DIME = 10,
    QUARTER = 25, DOLLAR = 100,
};
int main(int main, char ** argv) {
    enum Coin c = DOLLAR + QUARTER + CENT;
    printf("%u\n", c);
}
struct

Structs are like Java classes without constructors or methods:

```c
#define TUITION 10000
struct Student {
    unsigned char age;
    int assets;
    unsigned short credit_hours;
};
struct Student alice;
void do_register() {
    alice.assets -= TUITION;
    alice.credit_hours += 4;
}
```
Initializing structs

```c
struct Student {
    unsigned char age;
    int assets;
    unsigned short credit_hours;
};

int main() {
    struct Student alice = { 20, 10000, 48 }; 
    printf("Alice owns $%d\n", alice.assets);
    return 0;
}
```
Call-by-value & struct

```c
void do_register(struct Student s) {
    s.asset_value -= TUITION;
    s.credit_hours += 4;
}

int main(int main, char ** argv) {
    struct Student alice = { 20, 10000, 48 };  
    do_register(alice);
    printf("Alice owns $%d\n", alice.asset_value);
    return 0;
}
```
union

- Syntactically like struct
- Memory is shared between all members

```c
union Fun {
    unsigned int data; char name[8]; }
int main(int argc, char** argv) {
    union Fun f;
    f.data = 0x46554E00;
    printf("Unions are %s?\n", 
        f.name);
    return 0;  }
```
Fun!

```c
struct IntOrFloat {
    enum { IOF_INT, IOF_FLOAT } type;
    union {
        int i;
        float f;
    } value;
};

int f() {
    struct IntOrFloat val;
    val.type = IOF_INT;
    val.value.f = 3.14;
    printf("%d\n", val.value.i);
}
```
Bitfields

Bitfields can be defined as members of a struct:

```c
struct Datum {
    unsigned int day:5, month:4, year:7;
}
```
**typedef**

typedef is a way to introduce a new name for a type.

Examples:

typedef unsigned int uint32;
typedef float real;
typedef struct complex { real re; real im; } complex;
typedef struct { real re; real im; } complex;
typedef enum { YES, NO } decision;
Operators

• Very similar to Java.

• Make sure you know: $5 \& 3$ vs. $1 \& \& 5$ and $1 | 2$ vs. $1 || 2$, $2 << 4$ vs. $4 >> 2$

• New operator `sizeof` returns size of a value, variable or type in bytes.

• C does not have `new` as a keyword.

C has additional operators for pointers (*p, a->f, &v) which we will discuss in lecture 6.
sizeof

union Fun {
    unsigned int data; char[8] name;
}

int main(int argc, char ** argv) {
    union Fun f;
    printf("sizeof(union Fun)=%u sizeof(unsigned int)=%u"
" sizeof(f)=%u\n",
    sizeof(union Fun), sizeof(unsigned int),
    sizeof(f));
    return 0;
}
IF

if (expression)
    s1;
else
    s1;

• C has no booleans!
• expression can be a pointer: true \equiv \text{not NULL}
• expression can be a number: true \equiv \text{not zero}
The Ternary Operator “?:”

• Same as in Java, except C has no booleans

```c
int main() {
    int a = 4; int b = 5;
    printf((a > b) ? "YES" : "NO");
    return 0;
}
```
• Just like in Java

• Use `gcc -Wall` to get a warning for switches on enums if some case is missing!

```c
switch (dow) {
    case SA: case SU:
        printf("Freedom");
        break;
    case MO, TU, TH, FR:
        printf("Work");
}
```
int main(int argc, char ** argv) {
    while (1) {
        printf("1");
        while (1) {
            printf("2");
            if (0) continue; // NO labels allowed!
            if (1) break;     // NO labels allowed!
        }
        if (1) break;     // NO labels allowed!
        printf("3");
    }
    printf("4");
    return 0;
}
DO-WHILE

• Just like in Java
• Commonly used in macros (lecture 3)

```c
int main(int argc, char** argv) {
    do {
        printf("1");
    } while (0);
}
```
FOR

Syntax: for (INIT;COND;LOOP) BODY

Examples:

unsigned int i;
for (i=0; i<10; i++) { ... }
for (i=9; i>=0; i--) { ... }
i = 10;
for (; i != 0;) { ...; i--; }
for is not just for integers:

```c
struct List {
    struct List * next;
};
struct List * head = ...;
struct List * pos;
for (pos=head; pos!=NULL; pos=pos->next) { ... }
for (pos=head; pos;) { ... ; pos=pos->next; }
```
FOR and WHILE

for and while are equivalent:

```c
struct List {
    struct List * next;
};
struct List * head = ...;
struct List * pos;
pos = head;
while (pos != NULL) {
    ...
pos = pos->next;
}
```
FOR

• “for(;;) {...}” is the same as “while(1) {...}”
• It is better style to use while(1)
• Use for for iterations over arrays (forwards, backwards)
• Use while for almost everything else
int main(int argc, char** argv) {
    int fd;
    char buf[100];
    int have;
    fd = open("/etc/passwd");
    if (-1 == (have = read(fd, buf, sizeof(buf))))
        goto DONE;
    process(buf, have);
DONE:
    close(fd);
    return 0;
}
GOTO

• Some programmers consider `goto` harmful — to structured programming\(^1\)

• Java reserves the `goto` keyword, but does not allow its use

• Use `goto` only for clean up code in C or to break out of nested loops

\(^1\)E. Dijkstra: “Go To Statement Considered Harmful”
In Java

```java
int f() {
    OUTER:
        while (true) {
            printf("1");
            while (true) {
                printf("2");
                if (true) break OUTER;
            }
            printf("3");
        }
    return 0;
}
```
In C

```c
int f() {
    while (1) {
        printf("1");
        while (1) {
            printf("2");
            if (1) goto EXIT;
        }
        printf("3");
    }
    EXIT:
    return 0;
}
```
Questions

?