CS 33

Introduction to C
Part 6
Numeric Conversions

short a;
int b;
float c;

b = a;    /* always works */
a = b;    /* sometimes works */
c = b;    /* sort of works */
b = c;    /* sometimes works */
Implicit Conversions (1)

```c
float x, y=2.0;
int i=1, j=2;

x = i/j + y;
/* what's the value of x? */
```
Implicit Conversions (2)

```c
float x, y = 2.0;
int i = 1, j = 2;
float a, b;

a = i;
b = j;
x = a / b + y;
/* now what's the value of x? */
```
Explicit Conversions: Casts

float x, y = 2.0;
int i = 1, j = 2;

x = (float)i/(float)j + y;
/* and now what's the value of x? */
Purposes of Casts

• Coercion

```c
int i, j;
float a; // sizeof(float) == 4
a = (float)i/(float)j;
```

• Intimidation

```c
float x, y;
swap((int*)&x, (int*)&y);
```
Quiz 1

• Will this work?

```c
double x, y; //sizeof(double) == 8

... 

swap((int *)&x, (int *)&y);
```

a) yes  
b) no
Nothing, and More ...

• *void* means, literally, nothing:

```c
void NotMuch(void) {
    printf("I return nothing\n");
}
```

• What does *void* *const* mean?
  – it’s a pointer to anything you feel like
    » a generic pointer
Rules

• Use with other pointers

```c
int *x;
void *y;
x = y; /* legal */
y = x; /* legal */
```

• Dereferencing

```c
void *z;
func(*z); /* illegal!*/
func(*((int *)z)); /* legal */
```
Fun with Functions (1)

```c
void ArrayDouble(int A[], int len) {
    int i;
    for (i=0; i<len; i++)
        A[i] = 2*A[i];
}
```
Fun with Functions (2)

```c
void ArrayBop(int A[],
    int len,
    int (*func)(int)) {
    int i;
    for (i=0; i<len; i++)
}
```
Fun with Functions (3)

```c
int triple(int arg) {
    return 3*arg;
}

int main() {
    int A[20];
    /* initialize A */
    ArrayBop(A, 20, triple);
    return 0;
}
```
Swap, Revisited

```c
void swap(int *i, int *j) {
    int *tmp;
    tmp = *j; *j = *i; *i = tmp;
}
/* can we make this generic? */
```
An Application: Generic Swap

```c
void gswap (void *p1, void *p2, int size) {
    int i;
    for (i=0; i < size; i++) {
        char tmp;
        tmp = (*((char *)p1))[i];
        (*((char *)p1))[i] = (*((char *)p2))[i];
        (*((char *)p2))[i] = tmp;
    }
}
```
Using Generic Swap

short a=1, b=2;
gswap(&a, &b, sizeof(short));

int x=6, y=7;
gswap(&x, &y, sizeof(int));

int A[] = {1, 2, 3}, B[] = {7, 8, 9};
gswap(A, B, sizeof(A));
For Our Next Trick …

```c
int *f0(int *a) {
    ...
}

double *f1(double *a) {
    ...
}

char *f2(char *a) {
    ...
}
```

What’s my type?
Working Our Way There …

• An array of 3 ints
  \[-\text{int A}[3];\]

• An array of 3 int *s
  \[-\text{int *A}[3];\]

• A func returning an int *, taking an int *
  \[-\text{int *f(int *)};\]

• A pointer to such a func
  \[-\text{int *(*pf)(int *)};\]
There …

• An array of func pointers
  \[\text{int }(*(*\text{pf}[3]))(\text{int }*);\]

• An array of generic func pointers
  \[\text{void }(*(*\text{pf}[3]))(\text{void }*);\]
Using It

```c
int *f0(int *a) { *a += 1; return a; }
double *f1(double *a) { *a += 1; return a; }
char *f2(char *a) { *a += 1; return a; }

int main() {
    int x = 1;
    int *p;
    void **pf[3];
    pf[0] = (void **(*)(void *))f0;
    pf[1] = (void **(*)(void *))f1;
    pf[2] = (void **(*)(void *))f2;
    p = pf[0](&x);
    printf("%d\n", *p);
    return 0;
}
```

$ ./funcptr
2
$
int *f0(int *a) { *a += 1; return a; }
double *f1(double *a) { *a += 1; return a; }
char *f2(char *a) { *a += 1; return a; }
int main() {
    int x = 1;
    int *p;
    void *(*pf[3])(void *);
    pf[0] = (void *(*(*)(void *))f0);
    pf[1] = (void *(*(*)(void *))f1);
    pf[2] = (void *(*(*)(void *))f2);
    p = pf[1](&x); // was pf[0]
    printf("%d\n", *p);
    return 0;
}

What is printed?

a) 2
b) 2.5
c) something different from the above
d) nothing: syntax error
Casts, Yet Again

• They tell the C compiler: “Shut up, I know what I’m doing!”

• Sometimes true
  \[
  \text{pf}[0] = (\text{void }(*)(\text{void }*))f0;
  \]

• Sometimes false
  \[
  \text{long } f = 7;
  (\text{void }(*)(\text{int}))f(2);
  \]
Laziness …

• Why type the declaration
  \[
  \text{void \:*(*f)(void \,*\,,\, void \,*);}\\
  \]
• You could, instead, type
  \[
  \text{MyType \,*\, f;}\\
  \]
• (If, of course, you can somehow define \text{MyType} to mean the right thing)
typedef

- Allows one to create new names for existing types

```c
typedef int *IntP_t;
```

```c
IntP_t x;
``` – means the same as

```c
int *x;
```
typedef struct complex {
    float real;
    float imag;
} complex_t;

complex_t i, *ip;
And ...

typedef void *(*MyFunc_t)(void *, void *);

MyFunc_t f;

// you must do its definition the long way

void *f(void *a1, void *a2) {
    ...
}

Quiz 3

• What’s A?

typedef double X_t [M];
X_t A[N];

a) an array of N doubles
b) an MxN array of doubles
c) an NxM array of doubles
d) a syntax error
CS 33

Data Representation

Part 1
Unsigned Integers

\[ \text{value} = \sum_{i=0}^{w-1} b_i \cdot 2^i \]
Signed Integers

- Sign-magnitude

\[
\begin{array}{cccccccc}
\text{sign} & b_{w-1} & b_{w-2} & b_{w-3} & \ldots & b_2 & b_1 & b_0 \\
\text{magnitude} & & & & & & & \\
\end{array}
\]

\[
\text{value} = (-1)^{b_{w-1}} \cdot \sum_{i=0}^{w-2} b_i \cdot 2^i
\]

- two representations of zero!
  - computer must have two sets of instructions
  - one for signed arithmetic, one for unsigned
Signed Integers

• Ones’ complement
  – negate a number by forming its bit-wise complement
    » e.g., (-1)·01101011 = 10010100

\[ b_{w-1} = 0 \Rightarrow \text{non-negative number} \]

\[ \text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ b_{w-1} = 1 \Rightarrow \text{negative number} \]

\[ \text{value} = \sum_{i=0}^{w-2} (b_i-1) \cdot 2^i \]
Signed Integers

- Two’s complement
  \[ b_{w-1} = 0 \Rightarrow \text{non-negative number} \]
  \[ \text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i \]
  \[ b_{w-1} = 1 \Rightarrow \text{negative number} \]
  \[ \text{value} = (-1) \cdot 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]
  one zero!