ECS 36A, April 11, 2024
Announcements

• Be sure you use this command to run your program in the CSIF before submitting it to Gradescope:
  gcc –ansi –pedantic –Wall filename.c –o filename

• Some compilers allow // to comment out the rest of the line
  • Not part of the C90 standard
Announcements

My office hours:

• Tuesday 12:30pm–1:30pm, 2203 Watershed Sciences
• Wednesday 12:00pm–1:00pm, 2203 Watershed Sciences
• Friday 1:00pm–2:00pm, 2209 Watershed Sciences
  • I’m trying to get 2203 Watershed Sciences and will post an announcement if/when I do
Detail of –53 Being unsigned int 429496724

• Assume we are working on a 32-bit system
• Here is –53 represented as a 32-bit number:
  111111111111111111111111001011
• But if you read it as unsigned, this represents a *positive* number, here 4294967243
• Why? Because it is represented as \(2^{32} - 53\), not 53.
Functions

• Perform some task the program will do repeatedly
• Helpful for organizing programs
• Improves readability
Format

• Here is a function definition:

```c
int add17(int num) {
    int y;        /* used to hold sum */
    y = num + 17;
    return(y);
}
```

• Here is a function call:

```c
. . .
    sum1 = add17(53);
. . .
    sum2 = add17(-12);
. . .
```
In Detail – Function Definition

```c
int funct(int par1, float par2, char par3){ . . .
```

- `type of what function returns; if it doesn’t return anything, use `void` here`
- `name of function`
- `type of first parameter`
- `variable representing first parameter`
- `type of second parameter`
- `variable representing second parameter`
- `type of third parameter`
- `variable representing third parameter`
int x;
float fx;
x = funct(7, fx, ‘a’);

• Arguments are matched with parameters in order
• Here, from previous slide:
  • par1 is 7
  • par2 is the value contained in fx
  • par3 is ‘a’
• Note parameter types matches argument types
Prototypes or Forward Declarations

• Functions must be declared before use
• If defined before use, the function type, name, and parameter list serves as the declaration
• If defined after use, compiler makes assumptions about the types of parameters and function
  • And gcc will give you a warning
• A function prototype looks exactly like the first line of a function definition
  • int funct(int par1, float par2, char par3);
  • Note the “;” at the end!
void swap(int a, int b) {
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
And On The Calling End

\[
x = 13; \\
y = 5; \\
printf(\"x = %d, y = %d\n\", x, y); \\
swap(a, b); \\
printf(\"x = %d, y = %d\n\", x, y);
\]

April 14, 2023
ECS 36A, Spring Quarter 2023
void swap(int a, int b) 
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}

x = 3; y = 13;
swap(x, y);

print("x = %d; y = %d\n", x, y);
Pointers

• A variable containing the address of another variable

• Example:

```c
int x = 0;
int *px;
px = &x;
printf("x = %d, px = %p, *px = %d\n", x, (void *)px, *px);
```

• Operators:
  • `&variable`: address of `variable`
  • `*variable`: what is in the memory location with the address stored in `variable`
In Pictures

```python
v = 234
print("v = %d\n", v);
```

- prints “234” (without the “s, ending in newline)

```python
pv = 0x7fff34293
print("pv = %p\n", (void *)pv);
```

- prints “0x7fff34826” (without the “s, ending in newline)

```python
*pv = 234
print("*pv = %d\n", *pv);
```

- prints “234” (without the “s, ending in newline)
Function Arguments (No Pointers)

```c
void swap(int a, int b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```
Function Arguments (Pointers)

```c
void swap(int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```
And On The Calling End (No Pointers)

\[
x = 13; \\
y = 5; \\
printf(\"x = %d, y = %d\n\", x, y); \\
swap(x, y); \\
printf(\"x = %d, y = %d\n\", x, y); \\
\]

April 17, 2023

ECS 36A, Spring Quarter 2023
And On The Calling End (With Pointers)

\[
x = 13; \\
y = 5; \\
printf("x = \%d, y = \%d\n", x, y); \\
swap(&x, &y); \\
printf("x = \%d, y = \%d\n", x, y);
\]

April 17, 2023
Scope

• When multiple variables have the same name, which one is used?
  • Rule #1: two variables cannot have the same name in a block (e.g., function)
• Use the variable that is “nearest” to the reference
  • That’s the one in scope