CS113: Lecture 5

Topics:

- Pointers
- Pointers and Activation Records

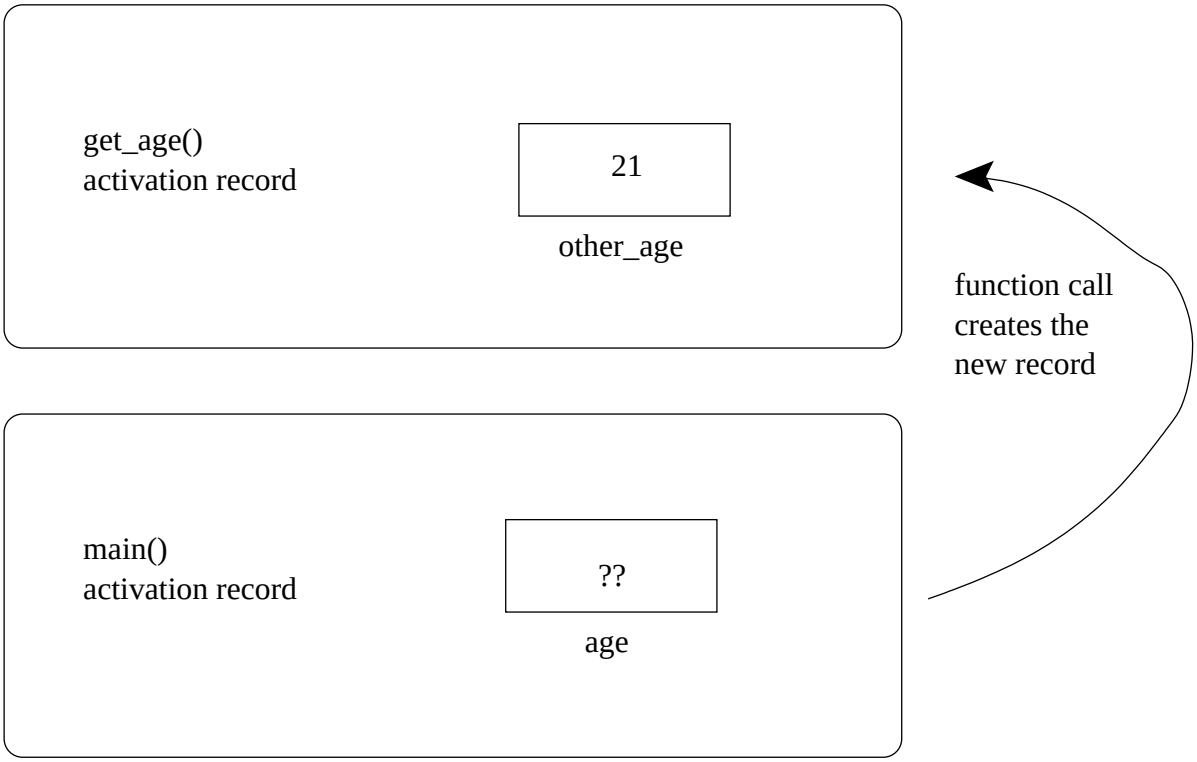
From Last Time: A Useless Function

```
#include <stdio.h>

void get_age( int age );

void main() {
   int age;
   get_age( age );
   printf( "Your age is: %d\n", age );
}

void get_age( int other_age ) {
   printf( "Please enter your age.\n" );
   scanf( "%d", &other_age );
}
```

- This is a contrived example, because we could just have get_age return the age as an integer.

- But what if we *want* a function to modify the contents of a variable we pass to it?
  - Suppose you want a function to sort an array of numbers, or swap the contents of two memory locations?
  - And how does "scanf" work? (Remember the & that comes before the variable you pass to scanf?)

From last time: the execution stack

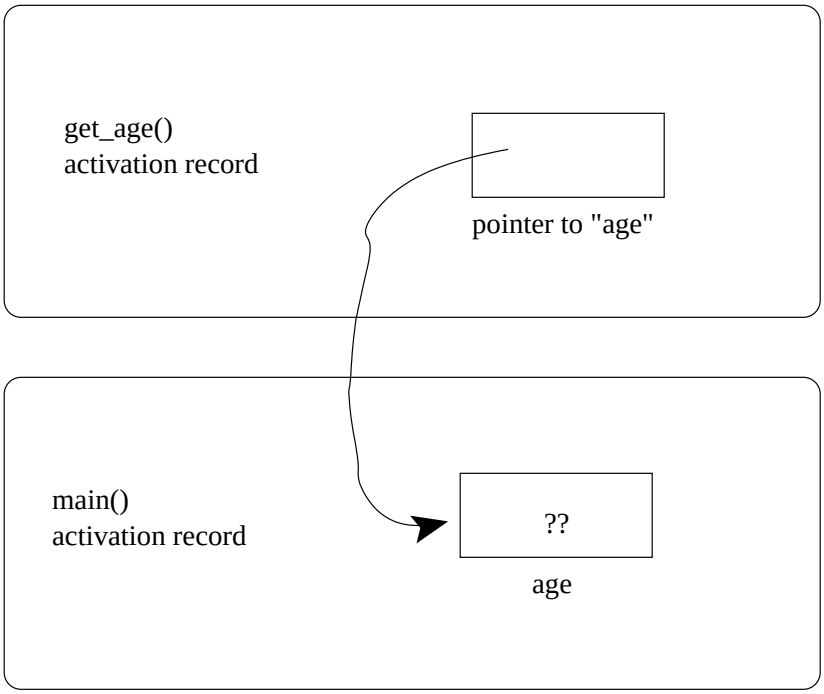


Remember that when `get_age()` finishes, its activation record is destroyed, so the value stored in `other_age` is lost.

Can we do any better?

What if?

Suppose that when main() calls get_age(), we passed a "pointer" to the age variable in main(), so that get_age() knew about that variable.



Then, perhaps, we could tell get_age() to modify the variable that it points to.

In fact, C lets us do exactly this, using *pointers*.

Introduction to Pointers

- A variable in a program is stored in a certain number of bytes at a particular memory location, or address, in the machine.

- Pointers allow us to manipulate these addresses explicitly.

- To declare a pointer variable: add a star to the type you want to point to. Example:

  ```
  int *a;
  ```

  declares a variable `a` of type `int *`, which can be used to hold the address of (or a "pointer to") an `int`.

- Two unary operators ("inverses"):

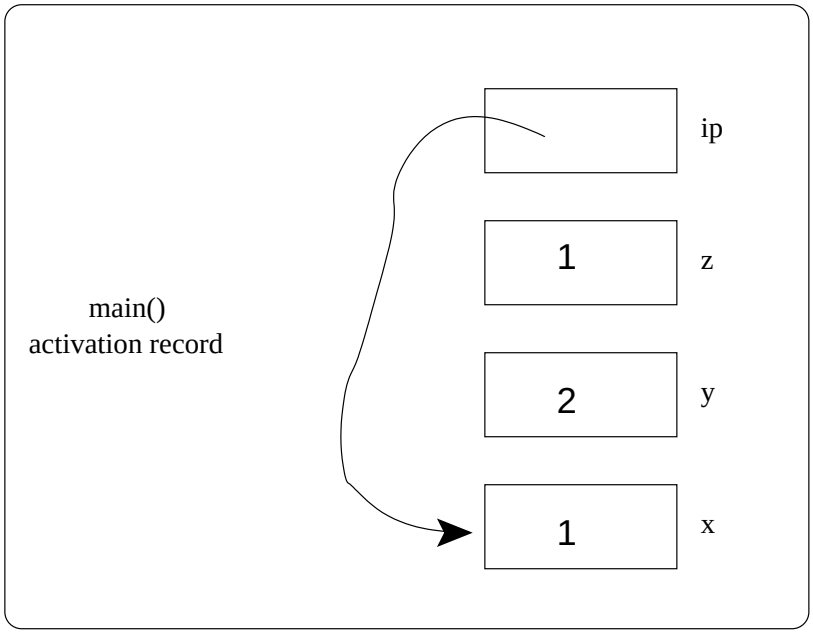
  - `&` operator - "address of" operator. Can be applied to any variable. Type of resulting expression has "one more star" than original expression.

  - `*` operator - "dereference" operator. Can be applied only to expressions that represent memory locations. Accesses the object that the pointer points to. Type of resulting expression has "one less star" than original expression.

Pointers: Example 1

```
void main() {
  int x = 1, y = 2, z = 3;
  int *ip;

  ip = &x;
  z = *ip;
}
```

Here's what it looks like after the last statement:



Don't Get Confused!

Pointer notation is pretty confusing:

- When we declare `int *a`, we're saying that the type of `a` is pointer to variables of type `int`. It might be less confusing if we wrote `int* a`, to emphasize that the *type* of `a` is `int*`, but this isn't usually how C programmers write the declaration. (It works, though.)

- The `*` operator dereferences the pointer to get at the variable we're pointing to. It makes the pointer "less of a pointer", whereas the `*` in the declaration makes the type "more of a pointer".

- In short: don't confuse the `*` *operator* with the `*` in the *declaration* of a pointer variable (or with multiplication)!

Pointers: Example 2

```
int x = 1, y = 2;
int *ip;
char c;
char *cp;

ip = &x;    /* ip now points to x */
printf( "%d\n", *ip );     /* prints 1 */
printf( "%d\n", *ip + 2 ); /* prints 3 */
y = *ip;    /* y is now 1 */
*ip = 0;    /* x is now 0 */

printf( "%d\n", x );       /* prints 0 */

cp = &x;    /* doesn't work; types don't match */
*cp = 'z';  /* what happens? */
cp = &c;
*cp = 'z';
printf( "%c\n", c );       /* prints z */
```

printf vs. scanf

```
void main() {
    int k;
    printf( "Enter an integer: " );
    scanf( "%d", &k );
    printf( "%d", k );
}
```

This also works – scanf is happy as long as it gets a pointer:

```
void main() {
    int k, *pk;
    pk = &k;
    printf( "Enter an integer: " );
    scanf( "%d", pk );
    printf( "%d", k );
}
```

Who wants what information?

- `printf( "%d", ... );` expects an `int`, since it needs to know *what* to print out
- `scanf( "%d", ... );` expects the *address* of an `int`, since it needs to know *where* to place the int typed in
  - `scanf` doesn't care about the actual value of the `int` that it should write to

More practice

```
void main() {
  int a = 3, b = 3;
  int *pa, *pb;

  pa = &a;
  pb = &b;

  if( pa == pb )
    printf( "pa and pb are equal.\n" );
  if( *pa == *pb )
    printf( "*pa and *pb are equal.\n" );

  (*pa)++; /* careful: different from *pa++ */
  *pb += *pa;
  printf( "a: %d, b: %d\n", a, b );

  pb = pa;
  *pa += *pb;
  printf( "a: %d, b: %d\n", a, b );
  if( pa == pb )
    printf( "pa and pb are equal.\n" );
  if( *pa == *pb )
    printf( "*pa and *pb are equal.\n" );

  /* super tricky */
  *((0 > 1) ? &a : &b) = 5;
}
```

How to swap two values?

What's wrong with this?

```
void swap( int x, int y ) {
   int temp;

   temp = x;
   x = y;
   y = temp;
}

void main() {
   int a = 3, b = 5;
   swap( a, b );
   printf( "a is %d, b is %d\n", a, b );
}
```

A correct swap

```
void swap( int *px, int *py ) {
   int temp;

   temp = *px;
   *px = *py;
   *py = temp;
}

void main() {
   int a = 3, b = 5;
   swap( &a, &b );
   printf( "a is %d, b is %d\n", a, b );
}
```

Be careful with your new toys.

When you're using pointers, *always* think of the activation records that will be generated by the program!

- Do not point at expressions that are not variables.

```
int k = 1, *ptr;
ptr = &3;  /* illegal */
ptr = &(k + 99);  /* illegal */
```

- Do not try to dereference non-pointer variables.

```
int k;
printf( "%d", *k ); /* illegal */
```

- What's wrong with this?

```
int *function_3() {
   int b;
   b = 3;
   return &b;
}

void main() {
   int *a;
   a = function_3();
   printf( "a is equal to %d\n", *a );
}
```

  - When a function returns, its activation record (along with the data it contains) gets destroyed!

An example

```
void main() {
   int a, b;
   int *pc, *pd;
   int **ppe, **ppf;

   a = 3;
   b = 5;
   pc = &a;
   pd = &b;
   (*pd)++;
   printf( "a: %d  b: %d\n", a, b );
   *pc += *pd;
   printf( "a: %d  b: %d\n", a, b );

   ppe = &pc;
   ppf = &pd;
   *ppf = pc;
   *pd = 12;
   printf( "a: %d  b: %d\n", a, b );

   **ppe = 50;
   **ppf = 15;
   printf( "a: %d  b: %d\n", a, b );
}
```