KU LEUVEN

Introduction to C

Pointers: basics

Pointers

- Access to values of variables in memory
- Method of passing parameters from/to functions.
  - Simulate call-by-reference
- Strong connection between arrays and pointers
- Create and manipulate dynamic data structures.
  - Creation of dynamic structures (Linked lists)

Pointer

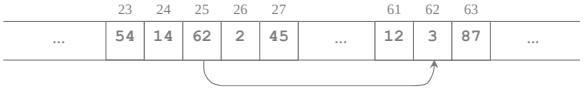
- A pointer is a variable that contains the address of another variable.
- When a variable is defined, it is allocated a portion of memory. The variable has a value and an address.

```
char x =  3;
```

- Assume **x** is stored at address 62.
- A pointer is a variable that holds the address of another variable.

```
char *px = &x;
```

- Assume **px** is stored at address 25. This variable points to **x**.



Pointer Syntax

- Declaration with * means “is a pointer to”.
- The & operator means “address of”.

```
int i;
int *j = &i;
```

- Spacing has no effect.

```
int* i, j, * k;
int val = 45;
k = &val;
```

- (Better?) style:

```
int * i, * k, j;
```

Pointer Syntax

- Deferencing or indirection operator *****

```
int i = 2, x;
int * j = &i;
*j = 5; /* i now equals 5 */
x = *j; /* x now equals i */
```

- Pointer dereferencing not to be confused with pointer declaration syntax. Difference is apparent from context.
- Tip: when dereferencing, use no space


```c
/*
pointer_1.c
http://gribblelab.org/cbootcamp/8_Pointers.html
*/
#include <stdio.h>

int main ()
{
  int age = 30;
  int * p;

  p = &age;

  printf("age=%d\n", age);
  printf("sizeof(age)=%ld\n", sizeof(age));

  printf("p=%p\n", p);
  printf("*p=%d\n", *p);
  printf("sizeof(p)=%ld\n", sizeof(p));

  *p = 40;
  printf("*p=%d\n", *p);
  printf("age=%d\n", age);

  age = 50;
  printf("*p=%d\n", *p);
  printf("age=%d\n", age);


  return 0;
}
```

```
frankvp@CRD-L-08004:.../Pointer$ gcc pointer_1.c -o pointer_1
frankvp@CRD-L-08004:.../Pointer$ ./pointer_1
age=30
sizeof(age)=4
p=0x7fff8203928c
*p=30
sizeof(p)=8
*p=40
age=40
*p=50
age=50
frankvp@CRD-L-08004:.../Pointer$
```

```c
#include <stdio.h>
// pointer_4.c
int main()
{

int m=3, n=100, * p;

p=&m;
printf("m is %d\n",*p);
m++;
printf("m is now %d\n",*p);
p=&n;
printf("n is %d\n",*p);
*p=500;
printf("n is now %d\n", n);

}
```

```
frankvp@CRD-L-08004:.../Pointer$ gcc pointer_4.c -o pointer_4
frankvp@CRD-L-08004:.../Pointer$ ./pointer_4
m is 3
m is now 4
n is 100
n is now 500
frankvp@CRD-L-08004:.../Pointer$
```


Pointer Syntax: NULL

- Bad practice to declare a pointer and not assign it to a valid object.
- Have a pointer that points to “nowhere”, make this explicit by assigning it to **`NULL`**.
  It indicates that it does not point to a meaningful position.

```
double * pval1 = NULL;
double * pval2 = 0;
```

- The integer constants 0 and 0L are valid alternatives to **`NULL`**, but the symbolic constant is more readable.
- NULL is defined in **`stdio.h`**

Pointers and Functions: Pass by Reference (simulation)

```c
/*
swap_by_value.c
Swap two variables - no pointers*/

#include <stdio.h>

void swap(double x, double y);


int main(){

    double a = 5, b = 10;

    printf("a = %f, b = %f\n", a, b);
    swap(a, b);
    printf("a = %f, b = %f\n", a, b);

    return 0;
}


void swap(double x, double y)
{
    double t = x;
    x = y;
    y = t;
}
```

- Variables are *always* passed to functions “by value”: a copy is passed.
- The following example won’t work. **`(swap_by_value.c)`**

```
frankvp@CRD-L-08004:.../Pointer$ gcc swap_by_value.c -o swap_by_value
frankvp@CRD-L-08004:.../Pointer$ ./swap_by_value
a = 5.000000, b = 10.000000
a = 5.000000, b = 10.000000
frankvp@CRD-L-08004:.../Pointer$
```

KU LEUVEN

Pass by Reference

```c
/*
swap.c
Swap two variables */

#include <stdio.h>

void swap(double * px, double * py);


int main (void)
{
    double a = 5, b = 10;

    printf("a = %f, b = %f\n", a, b);
    printf("address a = %p, address b = %p\n", &a, &b);
    swap(&a, &b);
    printf("address a = %p, address b = %p\n", &a, &b);
    printf("a = %f, b = %f\n", a, b);

    return 0;
}

void swap(double * px, double * py)
{
    double t = *px;
    *px = *py;
    *py = t;
}
```

- Desired effect achieved by passing pointers to the variables.
- Example **`swap.c`**
  - Pointers are passed “by value”, but these copies still hold addresses of original variables.

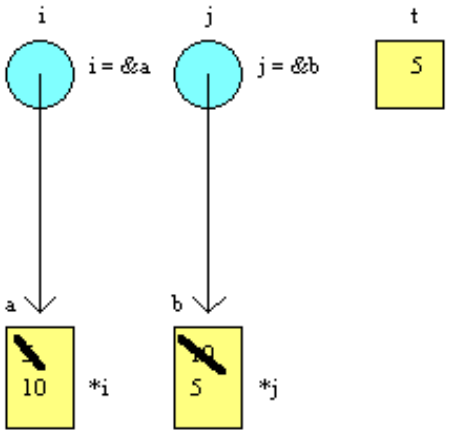
```
frankvp@CRD-L-08004:.../Pointer$ gcc swap.c -o swap
frankvp@CRD-L-08004:.../Pointer$ ./swap
a = 5.000000, b = 10.000000
address a = 0x7ffca03f5338, address b = 0x7ffca03f5340
address a = 0x7ffca03f5338, address b = 0x7ffca03f5340
a = 10.000000, b = 5.000000
frankvp@CRD-L-08004:.../Pointer$
```

KU LEUVEN

pointers and function arguments

```c
#include <stdio.h>

/* Swap two variables */
void swap(double *px, double *py)
{
    double t = *px;
    *px = *py;
    *py = t;
}
int main (void)
{
    double a = 5, b = 10;
    printf("a = %f, b = %f\n", a, b);
    swap(&a, &b);
    printf("a = %f, b = %f\n", a, b);
    return 0;
}
```



Pass by Reference

- Pointers provide *indirect* access to variables. This is why the ***** operator is called the *indirection* operator.
- Useful for:
  - Modifying the values of function arguments. Effectively enabling multiple return values. Example **`scanf.c`**
  - Avoiding copying of large objects (e.g., arrays, structures).

Pointer Syntax: `const` Pointers

- A pointer may be declared **`const`** in two different ways.
- **Constant pointer**
  - `<type of pointer> * const <name of pointer>`
  - This type of pointer is a pointer that cannot change the address it is holding.
- **Pointer to a constant**
  - `const <type of pointer>* <name of pointer>`
  - This type of pointer can change the address it is pointing to but cannot change the value kept at those addresses.

Constant pointer

- `<type of pointer> * const <name of pointer>`
- Can change value of pointed-to object, but pointer must always refer to the same address.

```c
#include<stdio.h>
/*
const_pointer.c
constant pointer: A constant pointer is a pointer that cannot change the address its holding.
Once a constant pointer points to a variable then it cannot point to any other variable.
*/

int main(void)
{
    int var1 = 0, var2 = 0;
    int * const ptr = &var1;

    printf("var1 pointed by ptr = %d\n", *ptr);
    *ptr = 10;
    printf("var1 pointed by ptr = %d\n", *ptr);

//    ptr = &var2;  // try compiling

    return 0;
}
```

```
frankvp@CRD-L-08004:.../Pointer$ gcc const_pointer.c -o const_pointer
frankvp@CRD-L-08004:.../Pointer$ ./const_pointer
var1 pointed by ptr = 0
var1 pointed by ptr = 10
frankvp@CRD-L-08004:.../Pointer$
```

Pointer to a constant

- `const <type of pointer>* <name of pointer>`
- Can change *what* it points to but cannot change the *value* of the object it points to.

```c
#include<stdio.h>
/*
pointer_to_const.c
This type of pointer can change the address it is pointing to
but cannot change the value kept at those addresses.
*/

int main(void)
{
    int var1 = 0;
    int var2 = 100;
    const int *ptr = &var1;

    var1 = 6;
    printf("%d\n", *ptr);

//    *ptr = 1;   // try compiling this line
    var1 = 1;
    printf("%d\n", *ptr);

    ptr = &var2;
    printf("%d\n", *ptr);

    return 0;
}
```

```
frankvp@CRD-L-08004:.../Pointer$ gcc pointer_to_const.c -o pointer_to_const
frankvp@CRD-L-08004:.../Pointer$ ./pointer_to_const
6
1
100
frankvp@CRD-L-08004:.../Pointer$ gcc pointer_to_const.c -o pointer_to_const
pointer_to_const.c: In function 'main':
pointer_to_const.c:17:10: error: assignment of read-only location '*ptr'
   17 |     *ptr = 1;   // try compiling this line
      |          ^
frankvp@CRD-L-08004:.../Pointer$
```

KU LEUVEN

Pointer Arithmetic

- Each variable type has a corresponding pointer type.
  - This allows the compiler to scale pointer-offset expressions appropriately. That is, it automatically calculates byte-offset.
- Pointer arithmetic is uniform for all types, and type-size details are hidden from the programmer.

KU LEUVEN

```c
#include <stdio.h>
/*
pointer_arithmetic_1.c
https://www.geeksforgeeks.org/pointer-arithmetics-in-c-with-examples/
*/
#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores
    // the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("sizeof(int): %ld\n", sizeof(int));
    printf("Pointer ptr1 before Increment: ");
    printf("%p \n", ptr1);
    printf("content of memory ptr1 pointing at: %d \n", *ptr1);
    ptr1++;     // Incrementing pointer ptr1;
    printf("Pointer ptr1 after Increment: ");
    printf("%p \n", ptr1);
    printf("content of memory ptr1 pointing at: %d \n\n", *ptr1);

    printf("Pointer ptr2 before Decrement: ");
    printf("%p \n", ptr2);
    printf("content of memory ptr2 pointing at: %d \n", *ptr2);
    ptr2--; // Decrementing pointer ptr2
    printf("Pointer ptr2 after Decrement: ");
    printf("%p \n", ptr2);
    printf("content of memory ptr2 pointing at: %d \n\n", *ptr2);

    return 0;
}
```

```
frankvp@CRD-L-08004:.../Pointer$ ./pointer_arithmetic_1
sizeof(int): 4
Pointer ptr1 before Increment: 0x7ffc40bdcd44
content of memory ptr1 pointing at: 4
Pointer ptr1 after Increment: 0x7ffc40bdcd48
content of memory ptr1 pointing at: 1086180680

Pointer ptr2 before Decrement: 0x7ffc40bdcd44
content of memory ptr2 pointing at: 4
Pointer ptr2 after Decrement: 0x7ffc40bdcd40
content of memory ptr2 pointing at: 0

frankvp@CRD-L-08004:.../Pointer$
```

KU LEUVEN

Pointer Arithmetic Operations

```c
float fval, array[10];
float *p1, *p2, *p3 = &array[5];
int i=2, j;
p1 = NULL;   /* Assignment to NULL (or to 0 or 0L). */
p2 = &fval;  /* Assignment to an address. */
p1 = p2;     /* Assignment to another pointer - same type).*/
p2 = p3 - 4; /* Integer operation */
p2 += i;     /* Integer operation */
j = p3 - p2; /* Pointer subtraction: number of elements */
i = p2 < p3; /* Relational operations <, >, ==, !=, <=, >= */
```

- At end of this sequence:
  - `p1` points to `fval`
  - `p2` points to `array[3]`
  - `p3` points to `array[5]`
  - `i` equals `1` and `j` equals `2`

KU LEUVEN

Invalid Pointer Operations

- Addition or subtraction by a floating-point value.
- Multiplication or division by value of any type.
- Assignment to a non-pointer type.

Pointer return from function

- Return the memory address of the value is stored instead of the value itself
- Be careful not to return an address to a temporary variable in a function

```c
#include <stdio.h>
#include <stdlib.h>

/*
taken from Computer programming in C for beginners
*/

int * test(int); // the prototype
int main()
{
int var = -20;
int * ptr = NULL;
ptr = test(var);
printf("This is what we got back in main()-pointer: %p \n", ptr);
printf("This is what we got back in main()-value: %d \n", *ptr);
return 0;
}

int * test(int k)
{
int y = abs(k);
int * ptr1 = &y;
printf("The value of y in test() directly is %d \n", y);
printf("The value of y in test() indirectly is %d \n", *ptr1);
return ptr1;
}
```

Pointer return from function

```c
#include <stdio.h>
// return_pointer_max.c
// https://cs.brynmawr.edu/Courses/cs246/spring2014/Slides/10_Pointer_Array.pdf

int * max(int * x, int * y);

int
main ()
 {
int a = 1, b = 2;
int *ptr;

ptr = max(&a, &b);

printf (" pointer returned %p pointing to max value %d \n", ptr, *ptr);

return 0;

}

int * max(int * x, int * y)
 {
if (*x > *y)
   return x;
return y;
}
```

```c
#include <stdio.h>
// return_pointer_max_bis.c
// https://cs.brynmawr.edu/Courses/cs246/spring2014/Slides/10_Pointer_Array.pdf

int * max(int x, int y);

int
main ()
 {
int a = 1, b = 2;
int *ptr;

ptr = max(a, b);

printf (" pointer returned %p pointing to max value %d \n", ptr, *ptr);

return 0;

}


int * max(int x, int y)
 {
if (x > y)
   return &x;
return &y;
}
```

KU LEUVEN

Pointer return from function

- A function may return a pointer value.

```
int* func_returns_pointer(void);
```

- However, this is often a mistake – if points to a local variable in the function
  - *File: return_pointer_danger_2.c*
  - Especially dangerous because it sometimes works

KU LEUVEN

```c
#include<stdio.h>
// return_pointer_danger_2.c


int *abc(); // this function returns a pointer of type int
int fabc(int a);

int main()
{
    int b;
    int *ptr;
    ptr = abc(10);
    printf("pointer pointing to %p - value %d \n", ptr, *ptr);
    b = fabc(23);
    printf("pointer pointing to %p - value %d \n", ptr, *ptr);
    printf("b - value %d \n", b);
    return 0;
}

int *abc(int a)
{
    int x, *p;
    x = a;
    p = &x;
    return p;
}

int fabc(int a)
{
    int x;
    x = a;
    return x + x;
}
```

```
(base) frankvp@CRD-L-08004:/mnt/c/Temp/Develop/CDev/Pointer$ gcc -Wall return_pointer_danger_2.c -o return_pointer_danger_2
(base) frankvp@CRD-L-08004:/mnt/c/Temp/Develop/CDev/Pointer$ ./return_pointer_danger_2
pointer pointing to 0x7ffc542acd4c - value 10
pointer pointing to 0x7ffc542acd4c - value 23
b - value 46
(base) frankvp@CRD-L-08004:/mnt/c/Temp/Develop/CDev/Pointer$
```

KU LEUVEN

Pointer return from function

- Pointer return values are OK if the object it points to remains in existence.
- Static or external variables: static extent.

```
double* geometric_growth(void){
   static double grows = 1.1;
    grows *= grows;
   return &grows;}
```

- Variables with dynamic extent. (Dynamic memory)
- Variables passed as input arguments to the function.

```
char* find_first(char* str, char c)
/* Return pointer to first occurrence of c in str.
    Return NULL if not found. */
{
    while(*str++ != '\0')
        if (*str == c) return str;
    return NULL;
}
```


KU LEUVEN