Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CS Basics
12) Pointers

E. Benoist & C. Grothoff

Fall Term 2018-19

Computer Science Division

Pointers

- Pointers
- Dynamic Memory Allocation
- Pointer Arithmetic
- Multidimensional Arrays
- Pointers on Functions
- Conclusion
- Command line parameters

Pointers

Pointer

- **Contains an address in memory**
  - It is the location where the value is stored (and not the value itself)
- **Pointers are used frequently in C**
  - To pass information back and forth between to and from a function
  - Pointers give a way to return multiple data items from a function via function arguments (example: `scanf()`)
  - Pointers to functions allow to pass functions as arguments to functions

Fundamentals

- **Variables in Memory**
  - A variable represents information in memory
  - `int i;` the value of i is an integer
  - Variables can be automatic or static
- **Automatic Variables**
  - In functions or statements
  - Created dynamically
  - Accessible only in the function (or statement)
  - Memory is released when the variable is not needed anymore
- **Static and Global Variables**
  - Static variables are defined within a function
  - Global variables are independant of any function
  - Both remain in memory when the functions terminate
  - Similar to BSS memory section in Assembly

Address Operator & and Dereference Operator *

- **Suppose `v` is a variable**
  - Compiler assigns memory cell for this data item
  - The address of this cell is `&v`

```
pv = &v;
```

- **Pointer**
  - The new variable `pv` is a pointer to `v`.
  - The data item can be accessed using the star operator `*`
  - `*pv` is equivalent to `v`
  - `*` is called the *indirection* or *dereference operator*

Example of pointers

```
#include <stdio.h>
main(){
   int u = 3;
   int v;
   int *pu; /* pointer to an integer */
   int *pv; /* pointer to an integer */
   pu = &u;  /* assign address of u to pu */
   v = *pu; /* assign value of u to v */
   pv = &v; /* assign addresse of v to pv */
   printf("u␣=␣%d␣␣␣&u=%x␣␣pu=%x␣␣␣*pu=%d\n",u,&↘
   →u,pu,*pu);
   printf("v␣=␣%d␣␣␣&v=%x␣␣pv=%x␣␣␣*pv=%d\n",v,&↘
   →v,pv,*pv);
   return 0;
}
```

Another example

- **pointers2.c**

```
#include <stdio.h>
main(){
   int u1, u2;
   int v = 3;
   int *pv; /* pv is a pointer to v */
   u1 = 2 * (v + 5);
   pv = &v;
   u2 = 2 *(*pv + 5);
   printf("u1 = %d;  u2 = %d \n",u1,u2);
   /* Output:
      u1 = 16;  u2 = 16
    */
}
```

Pointer Declaration

- **Pointer variables are declared using an asterisk (*)**

```
data-type  *ptvar
```

- **Examples**

```
float u, v; /* are float variables */
float *pv; /* is a pointer */
pv = &v;
float *pu = &u;
```

Passing Pointers to a function

- **Passing arguments by "reference" (or by address)**
  - Normal arguments are passed by value: value is copied
  - Pointers arguments: the element is not copied,
  - Only the address is transfered
- **Syntax**

```
return-type funct2(data-type *arg1, data↘
→-type2 *arg2){
   ...
}
```

- **Example**

```
void funct2(int *pu, int *pv){
   ....
}
```

Example

- **pointers4.c**

```
#include <stdio.h>
void funct1(int u, int v);
void funct2(int *pu, int *pv);

main(){
  int u = 1;
  int v = 3;
  puts("before calling functions");
  printf("u = %d; v = %d \n",u,v);
  funct1(u,v);
  puts("After funct1(u,v);");
  printf("u = %d; v = %d \n",u,v);
  funct2(&u,&v);
    puts("After funct2(&u,&v);");
  printf("u = %d; v = %d \n",u,v);
}
void funct1(int u, int v){
  u = 0;
  v = 0;
  return;
}
void funct2(int *pu, int *pv){
  *pu = 0;
  *pv = 0;
  return;

}
```

Passing arguments by reference

- **Arguments passed by value (i.e. without pointers)**
  - Value is copied into an automatic variable in the function
  - The new variable can be modified: no change on the initial value
  - Two different positions in memory!
- **Arguments passed by reference (i.e. with pointers)**
  - The function receives the address of the variable
  - The function can read the value (using *)
  - The function can change the value
  - The size of the information in argument is very small (just the address) Almost nothing is copied

Example: Arguments values can be changed

- **pointers5.c**

```
#include <stdio.h>

void multiply(int* pi1, int pi2);
int main(){
  int i1;
  int i2;
  puts("Enter two numbers ");
  scanf("%d %d", &i1, &i2);
  multiply(&i1, i2);
  printf("i1=%d\n",i1);
}

void multiply(int* pi1, int pi2){
  *pi1 *= pi2;
}
```

Pointers and one-dimensional arrays

- **An array name IS a pointer to the first element in the array**
  - If x is an array: the address of the first element is `&x[0]` or simply `x`
  - The address of the second element is `&x[1]` or simply `x+1`
    notice: $+1$ depends on the size of the elements!
  - And more generally `&x[i]` is `x+i`
- **Content of an array**
  - Both expressions are equivalent `x[i]` and `*(x+i)`

Example: array

- **pointers6.c**

```
#include <stdio.h>
main(){
   static int x[10] = ↘
   →{10,11,12,13,14,15,16,17,18,19};
   for(int i = 0; i <= 9; ++i){
      printf("i␣=␣%d;␣x[i]␣=␣%d;␣*(x+i)=%d␣\n"↘
      →,i,x[i],*(x+i));
      printf("␣␣␣␣␣␣␣␣&x[i]␣=␣%X;␣x+i=%X␣\n",&x↘
      →[i],x+i);
   }
}
   /* Output:
        i = 0; x[i] = 10; *(x+i)=10
               &x[i] = 601040; x+i=601040
        i = 1; x[i] = 11; *(x+i)=11
               &x[i] = 601044; x+i=601044
        i = 2; x[i] = 12; *(x+i)=12
               &x[i] = 601048; x+i=601048
```

Access elements of an array

- **All the following expressions are equivalent**
  - `copy[i]=line[i];`
  - `copy[i]=*(line+i);`
  - `*(copy + i)=*(line + i);`
  - `*(copy + i)=line[i];`
  - Copy of an element from line into copy
- **Pointer pointing on the same array**
  - But starting from another start point

```
/* substring, without copy, same memory ↘
→part */
char* p1 = &line[2];
char* p2 = line+2;
```

Dynamic Memory Allocation

Dynamic Memory Allocation

- **Arrays can be:**
  - Automatic: stored temporary, destroyed at the end of a statement
  - Static / global: stored for the entire program execution
- **Often one cannot reserve memory at compile time**
  - Size is not known as it depends on an input
  - Size may change dynamically
  - Array may be too big for the stack
  - can not be foreseen at the compilation (similar to .BSS section in assembly or to stack information for automatic variables).
- **Solution: allocate memory dynamically**
  - Reserve memory in the free memory
  - Called the "heap"
  - Information remains until it is errased
  - Allocation duration does not depend on any scope
  - Can be used from anywhere in the program
  - Need to know the address (i.e. the pointer)

Example: dynamic1.c

```c
#include <stdio.h>
#include <stdlib.h>
double avrge(double *marks, int size);
main(){
   int i, n;
   double* marks;
   /* read in a value for n */
   puts("Number of marks:");
   scanf("%d", &n);
   if(n>0){
      marks = (double *) malloc(n*sizeof(double));
      for(i=0;i<n;i++){
         printf("%d) Enter Mark:",(i+1));
         scanf(" %lf",marks+i); // or: scanf(" %lf",&mark↘
         →[i]);
      }
      double average = avrge(marks, n);
      printf("Average is: %lf\n", average);
  } else printf("Need at least one number to calculate ↘
  →mean!\n");
}
```

Example: dynamic1.c (Cont.)

```
double
avrge (const double *marks, int size){
  double sum = 0.0;

  for (int i=0;i<size;i++)
    sum += marks[i]; // equivalent to *(marks+i↘
    →)

  return sum / size;
}
```

Dynamic Memory Allocation

- `malloc()` **function**
  - Belongs to the standard C library `stdlib.h`
  - Reserves the cells in memory
  - Creates a dynamic array in memory
- **Syntax**
  - `type* val = (type*) malloc(number * sizeof(type));`
- **Example**

```
double* marks;
marks = (double *) malloc(n*sizeof(↘
→double));
```

Proper Array Allocation

- Why write so much text?
- C has a preprocessor to take care of that!

Example

```
/* do this once, ideally in a header; ↘
→existing C code
   likely has something like this already ↘
   →*/
#define new_array(n,t) (t*) malloc(n * ↘
→sizeof(t))

double *marks;
marks = new_array (n, double);
if (NULL == marks) fail();
```

Free Memory

- `free()` **function**
  - Belongs to the standard C library `stdlib.h`
  - Frees the memory allocated dynamically
  - Should be called with the value returned by `malloc()`
- **Syntax**
  - `free(pointer)`
- **Example**

```
free(marks);
```

Pointer Arithmetic

Operations on Pointers

- **An integer value can be added to a pointer**
  - To access any individual array element.
  - `++px` array starts one element further
  - `--px` array starts one element before
  - `(px + 3)` the address of the fourth element
  - `(px + i)` the address of the (i+1)th element.
  - `(px - i)` the address of an element before the array px.

Operation on Pointers (Example)

```
#include <stdio.h>

int main(){
   int* px;
   int i = 1;
   float f = 0.3;
   double d = 0.005;
   char c = '*';

   px = &i;
   printf("Values: i=%i f=%f d=%f c=%c\n", i,f,d,c);
   printf("Addresses: i=%p f=%p d=%p c=%p\n", &i,&f,&d↘
   →,&c);
   printf("Pointer Values (size of int =%ld):\n px=%p;↘
   → px+1=%p;  px+2=%p;  px+3=%p\n", sizeof(int),px,px↘
   →+1,px+2,px+3);

   printf("Dereference: at px+1=%p, value is:%f \n", px↘
   →+1,*(px+1));
}
```

Operation on pointers

- **One can subtract two pointers:**
  - Returns the size of some information for instance

```
#include <stdio.h>
int main(){
   int * px, * py;
   static int a[6] = {1,2,3,4,5,6};
   px = &a[0];
   py = &a[5];
   printf("px=%p;  py=%p  \n",px,py);
   printf("py - px = %ld \n", py-px);
   /* Output
      px=0x601020;  py=0x601034
      py - px = 5
   */
}
```

Loop using pointers

- **Increment one pointer:**

```
#include <stdio.h>

main(){
   static int x[10] = ↘
   →{10,11,12,13,14,15,16,17,18,19};
   int* b;

   for (int i=0,b=x; i <= 9; ++i,b++){
        printf("b␣=␣%p;␣*b␣=␣%d\n",b,*b);
   }
   /* Output:
b = 0x601060; *b = 10
b = 0x601064; *b = 11
b = 0x601068; *b = 12
....
     */
}
```

Multidimensional Arrays

Multidimensional Arrays

- **One dimensional arrays can be represented with one pointer and one offset**
  - `*(arr + i)`
  - Represents the element with index `i` in the array `arr`
- **Multidimentional arrays can be represented with an equivalent pointer notation**
  - A two dimentional array (for example) is a collection of one dimensional arrays
  - We can define as: a pointer to a group of contiguous one-dimensional arrays.
  - Syntax

```
dat-type (*ptvar)[expression 2]
```

Example

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

#define MAXROWS 4

int main(){
  int cols, rows;
  int *a[MAXROWS];

  puts("Number of cols:");
  scanf(" %d",&cols);
  puts("Number of rows:");
  scanf(" %d", &rows);

  for(int row = 0; row < rows ; row++){
    a[row]= (int *) malloc (cols* sizeof(int));
  }
```

Example (Cont.)

```
printf("Enter table");
for(int r=0;r<rows;r++){
  for(int c=0;c<cols;c++){
    printf("val (%d,%d)=",r,c);
    scanf(" %d",(*(a+r)+c));
  }
}
for(int r=0;r<rows;r++){
  for(int c=0;c<cols;c++){
    printf(" val a[%d][%d]=%d",r,c,a[r][c]);
  }
  putchar('\n');
}
puts("other view of the same array");
for(int r=0;r<rows;r++){
  for(int c=0;c<cols;c++){
    printf(" val *(*(a+%d)+%d)=%d",r,c,*(*(a+r)+c));
  }
  putchar('\n');
}
}
```

Dynamically read a multidimentional array

- **We want to have an array of strings**
  - It is an array containing elements with a fixed length
  - Each string must have the same length (MAXSIZE in our example)
  - The new strings are allocated when needed

```
char *arrayOfStrings[10];
...
arrayOfStrings[n]=(char*) malloc(MAXSIZE * ↘
→sizeof(char));
```

- **Each string is read from the stdin**
  - `scanf("%s",arrayOfStrings[n]);`

Example

```
int main(){
   int n=0;
   char *arrayOfStrings[10];
   puts("Enter each string on a separate line");
   puts("Type control D to finished");
   int max=10;
   int size;
   do{
      arrayOfStrings[n]=(char*) malloc(MAXSIZE * sizeof(↘
      →char));
      scanf("%s",arrayOfStrings[n]);
      size = strlen(arrayOfStrings[n]);
      n++;
   }while(size > 0);
   sort(--n,arrayOfStrings);
   printf("List of strings (after sort)\n");
   for(int i=0;i<n;i++){
      printf("%d) %s\n",(i+1),arrayOfStrings[i]);
   }
}
```

Example (Cont.)

- **sort should be implemented as an exercise**

```
/* Should sort, function to be written in ↘
→exercise */
void sort(int size, char* x[]){
  return;
}
```

Pointers on Functions

Passing functions to other functions

- **A pointer to a function can be passed to another function as an argument**
  - One function can be transfered to one another
- **Two functions: the guest (the one being passed as a parameter) and the `host` (the one receiving this parameter)**
  - The guest is passed to the host where it can be accessed
  - Successive calls to the host function can pass different pointers (i.e. different functions) the the host
- **Syntax:** the parameter of the function must be a function with its arguments

```
data-type (*function-name)() /* no arguments↘
→ */
data-type (*function-name)(type1, type2, ↘
→...) /* just types */
data-type (*function-name)(type1 arg1, type2↘
→ arg2, ...) /* types and args */
```

Example

- **The host function `printSquare()` can receive one of two functions as argument: `printA` or `printB`**
  - If the number is even: printA is the guest function
  - if it is odd: printB is the guest function
- **Syntax**

```
void printSquare(int i, void (*pf)(int a)){
  int square = i*i;
  (*pf)(square);
}
```

Defining a function type

- typedef can be used to introduce new C types
- **Syntax**

```
typedef TYPE (*TypeName)(ARGTYPES);
```

- **Example**

```
typedef void (*PrintFunction)(int value);

void printSquare(int i, PrintFunction pf){
  int square = i*i;
  pf (square); // or equivalent: (*pf)(↘
  →square)
}
```

Example: Pointer on a function

```c
#include <stdio.h>
typedef void (*PrintFunction)(int value);

void printA (int i) { printf("AAAA %i\n",i); }
void printB (int i) { printf("BBBB %i\n",i); }

void printSquare(int i, PrintFunction pf) {
  pf (i * i);
}
int main(){
  int val;
  puts("type a number");
  scanf("%d",&val);
  if ((val%2)==0){
    printSquare(val, printA);
  } else {
    printSquare(val, &printB); // or just ``printB''
  }
}
```

Command line parameters

Command Line Parameters

- **Command Line Parameters are passed to the program**
  - Like for Assembly language
  - On the stack
- **Read parameters in C**
  - two arguments for `int main()`
  - `int argc` the number of arguments
  - `char* argv[]` the list of arguments (each argument is a string)
- **Syntax**

```
int main(int argc, char* argv[]){
   ...
}
```

Command Line Parameters

- **Example**
  - Read the parameters and print them out

```
int main(int argc,char* argv[]){
   printf("Print command line parameters\n");
   for(int i=0; i<argc; i++){
      printf("argv[%d] %s\n",i,argv[i]);
   }
}
/* Output:
bie1@machine $ ./commandline 1 2 3 4 5 fff
Print command line parameters
1) ./commandline
2) 1
3) 2
4) 3
5) 4
6) 5
7) fff
 */
```

Numerical Arguments

- **Parse numerical arguments**
  - We use `sscanf()` to parse numerical arguments

```
#include <stdio.h>
int main(int argc,char* argv[]){
  if(argc!=3)
    goto error;
  int val1,val2;
  int res1=sscanf(argv[1],"%d",&val1);
  int res2=sscanf(argv[2],"%d",&val2);
  if(res1!=1 || res2!=1)
    goto error;
  printf("%d␣x␣%d␣=␣%d\n",val1,val2,val1*↘
  →val2);
  return 0;
 error:
  printf("Usage:\n␣commandlinemultiply␣<num1↘
  →>␣<num2>\n");
  return 1;
}
```

Conclusion

Conclusion

- **Pointers**
  - Pointers contain addresses
  - They can be used as function parameters
  - or as function return values
- **Arguments by address**
  - Function can modify the content of a variable
  - The content is not copied (save time / space)
  - Variable may be modified unexpectedely
- **Pointer = array**
  - An array is a pointer
  - Manipulation of pointers for accessing an array
- **Pointers to functions**
  - Variables can point to function
  - `typedef` can be used to give new types a name

Bibliography

- This course corresponds to chapters *10* of the course book: **Schaum's OuTlines, Programming with C** (second edition), *Byron Gottfied*, Mc Graw-Hill, 1996