Systems@ETH zürich

Tutorial 2: More C

Computer Architecture and Systems Programming (252-0061-00)

Herbstsemester 2012

Pointers

Systems@ETH zürich

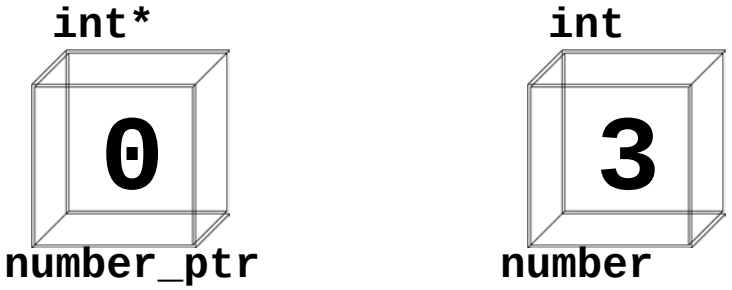
- Every variable in C has:
  - **Name**: what is it called?
  - **Address**: where in memory is it?
  - **Type**: how to interpret the value
  - **Value**: what is stored at the address
- C pointers are variables that contain the *addresses* of other variables.
- Java has no pointers (only object references), C# has them only in “unsafe code”

```
// px is a ptr to an integer
int *px;

// x is an integer
int x;

// px gets the address of x
// or px points to x
px = &x;

// x gets the contents of
// whatever x points to
x = *px
```

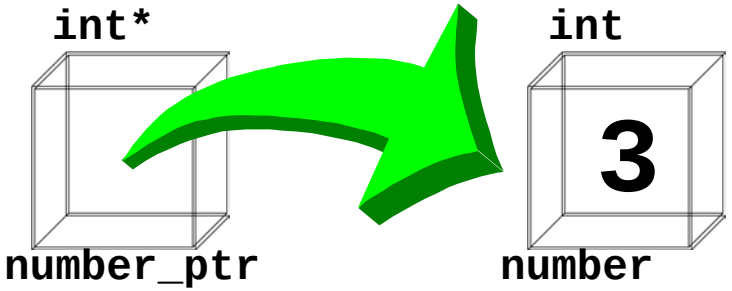
Pointers

Systems@ETH zürich

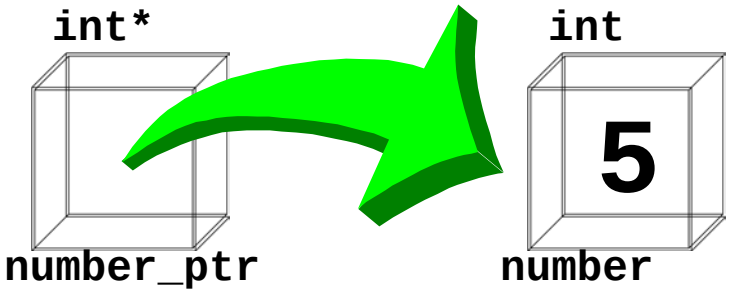
```
int number = 3;
int *number_ptr = NULL;
```




```
number_ptr = &number;
```

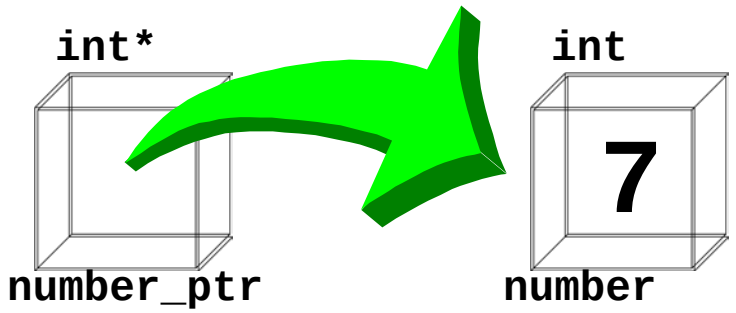



```
number = 5;
```

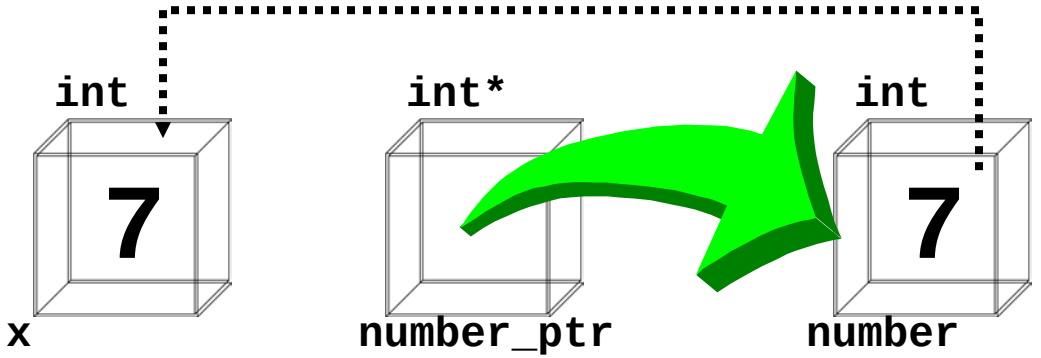


Pointers

```
*number_ptr = 7;
```

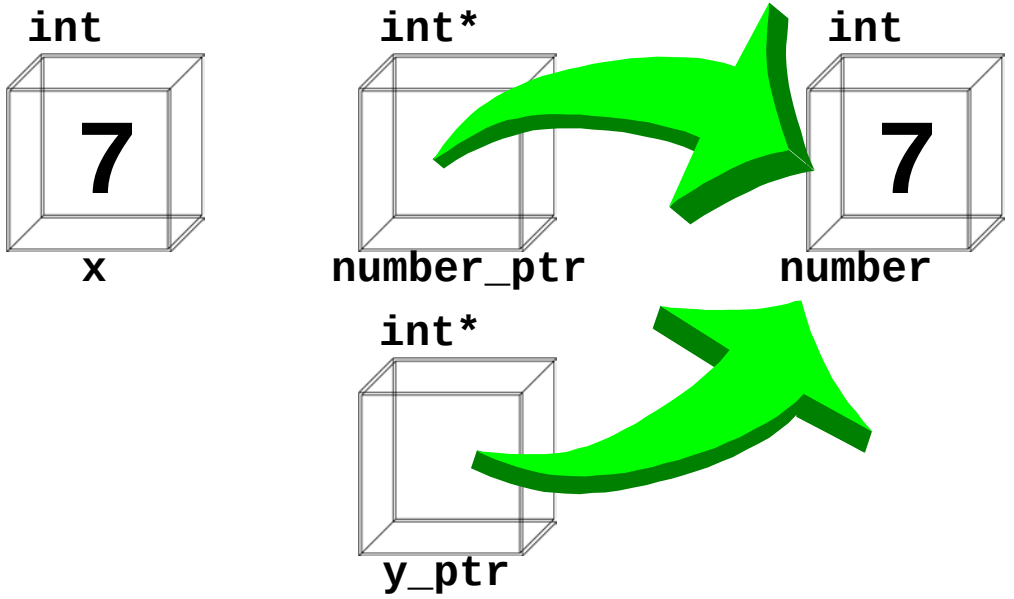



```
int x = *number_ptr;
```



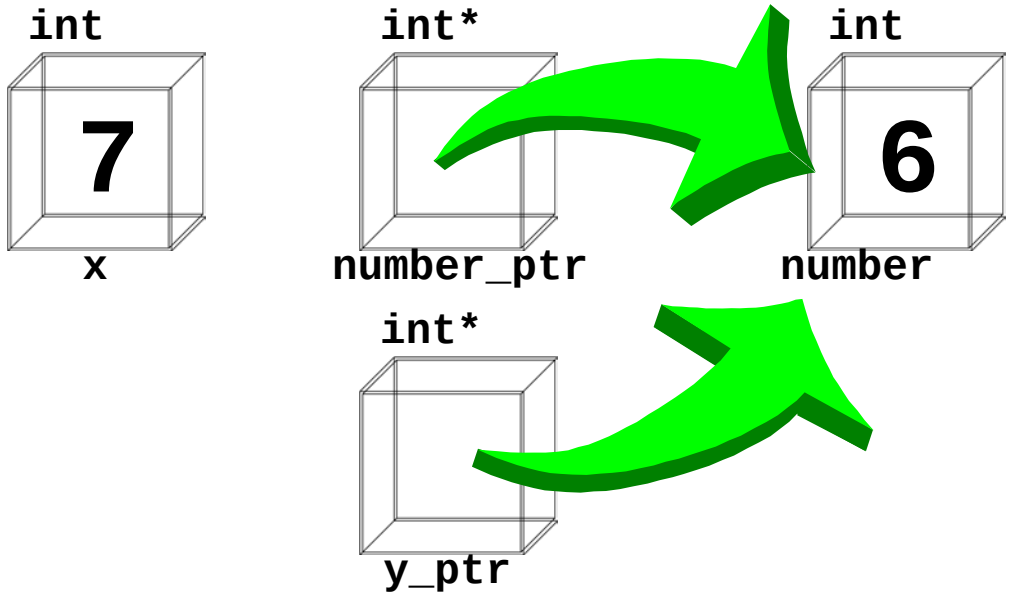
Pointers

```
int *y_ptr = number_ptr;
```



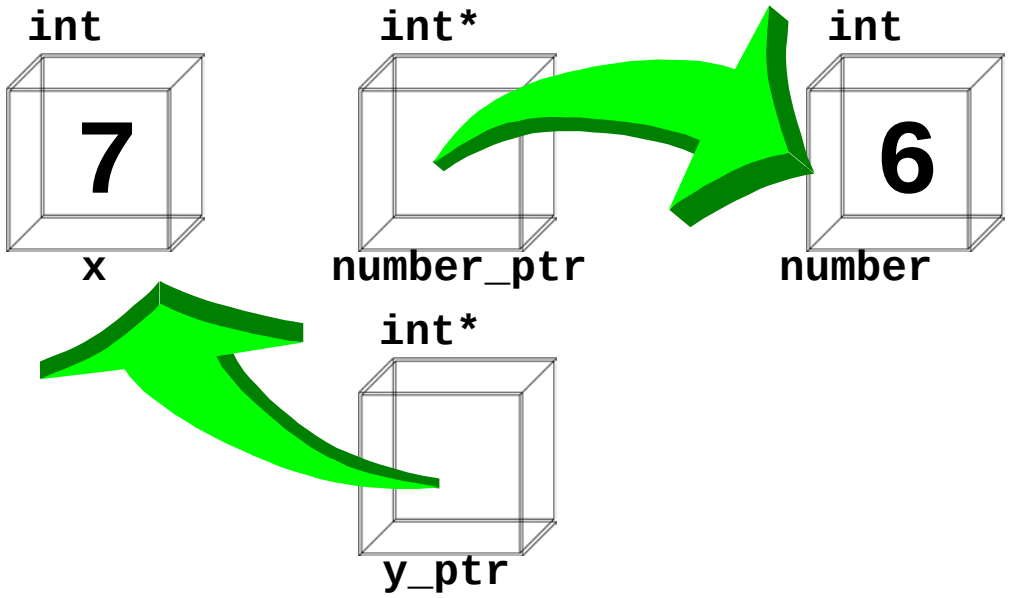
Pointers

```
*y_ptr = 6;
```



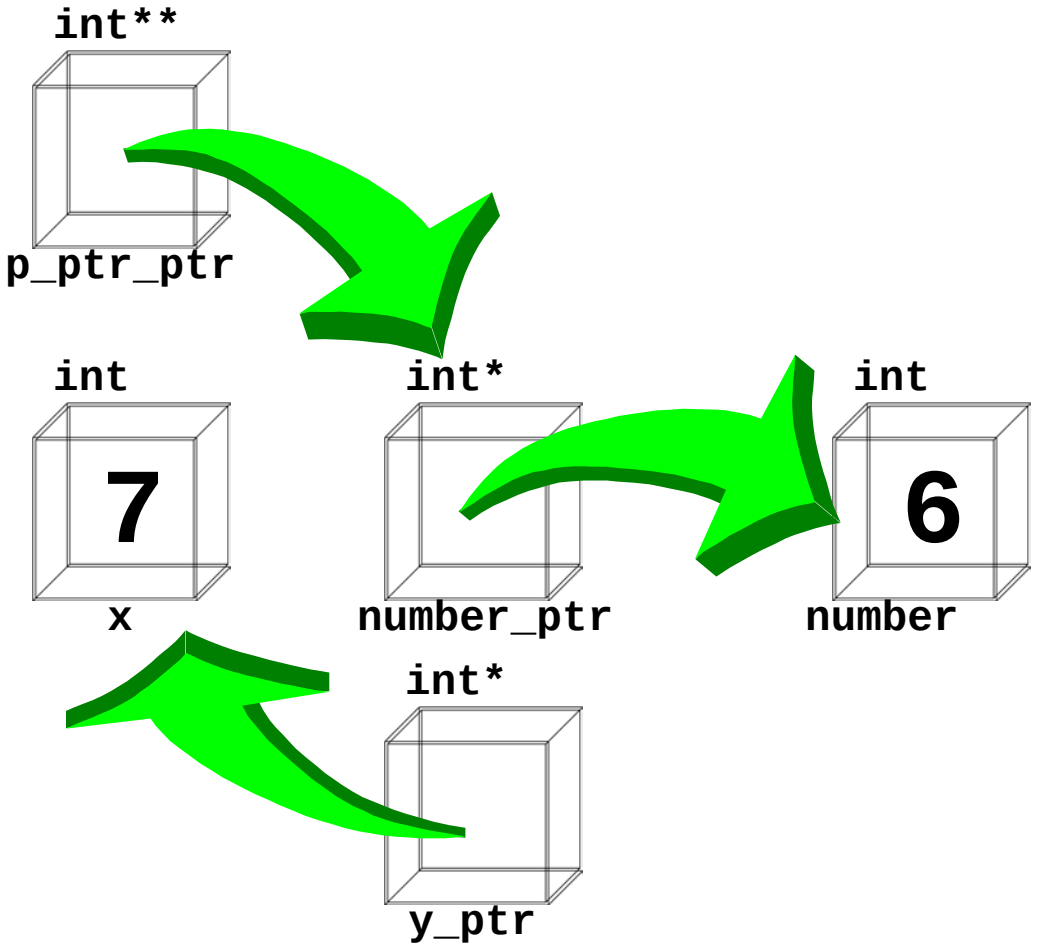
Pointers

```
y_ptr = &x;
```



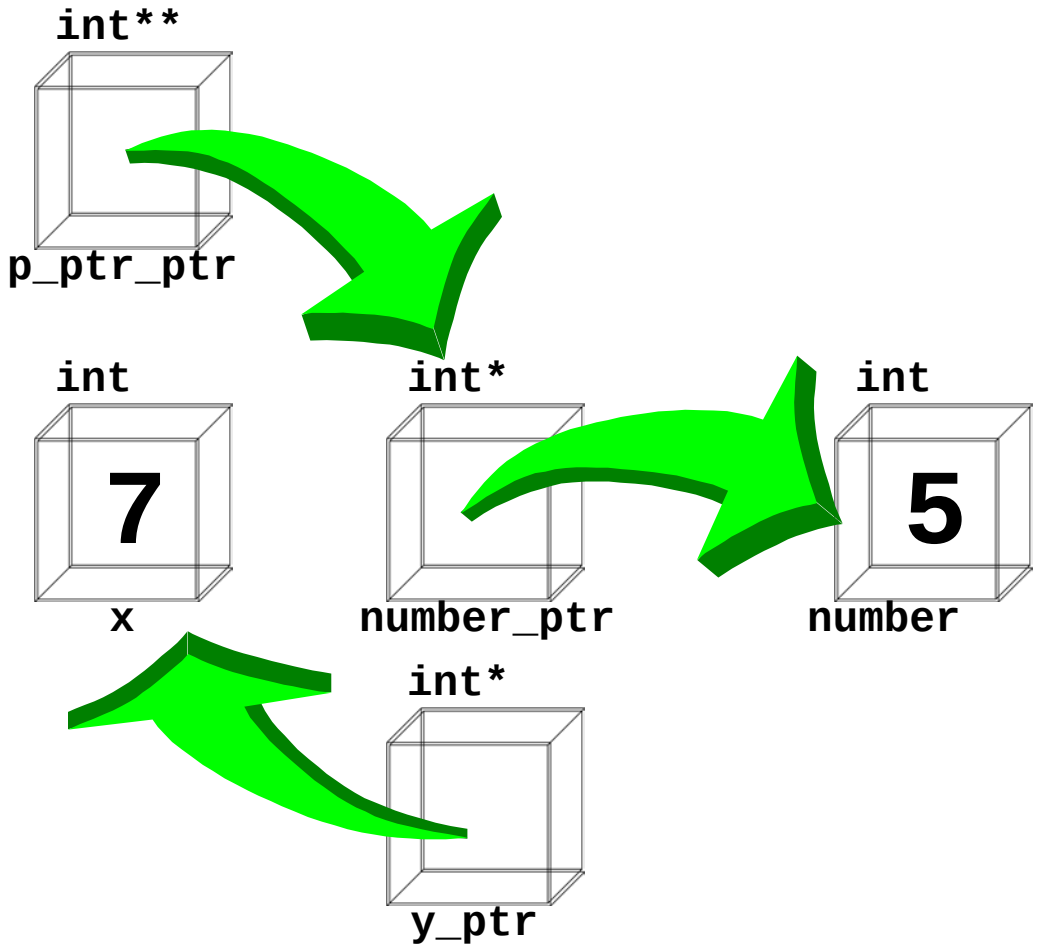
Pointers

```
int **p_ptr_ptr;
p_ptr_ptr = &number_ptr;
```



Pointers

```
*(*p_ptr_ptr) = 5;
```



C Pointer Declarations

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Pointers and arrays

Systems@ETH zürich

- An array is in reality (mostly) a pointer:

```
int a[10], y;
int *px;

px = a;
// px points to a[0]

px++;
// px points to a[1]

px=&a[4];
// px points to a[4]

y = *(px+3)
//y gets the value in a[3]
```

- Pointer arithmetic in C guarantees that if a pointer is incremented or decremented, the pointer will vary according to its type.
- For instance, if **px** points to an array, **px++** will always yield the next element independently of the array type

Pointers and strings

Systems@ETH zürich

- There is no “string type” in C
- Strings are arrays of, and pointers to, chars:

```
char *message;
message = “Some string”;
```

- **`message`** is a pointer that now points to the first character in the string “**`Some string`**”
- This is very useful to know, but always use **`string.h`** functions if you can
  - Avoid many errors
  - Easier to read the code

Example string library

Systems@ETH zürich

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

int main(int argc, char *argv[]) {
  char name1[12], name2[12], mixed[25];
  char title[20];

  strcpy(name1, "Rosalinda");
  strcpy(name2, "Zeke");
  strcpy(title, "This is the title.");

  printf("    %s\n\n", title);
  printf("Name 1 is %s\n", name1);
  printf("Name 2 is %s\n", name2);

  if(strcmp(name1, name2) > 0)
              /* returns 1 if name1 > name2 */
    strcpy(mixed, name1);
  else
    strcpy(mixed, name2);

  printf("The biggest name alphabetically is %s\n",
      mixed);

  strcpy(mixed, name1);
  strcat(mixed, "  ");
  strcat(mixed, name2);
  printf("Both names are %s\n", mixed);
  return 0;
}
```

```
This is the title.

Name1 is Rosalinda
Name2 is Zeke
The biggest name alphabetically is Zeke
Both names are Rosalinda  Zeke
```

Trouble with pointers

Systems@ETH zürich

What is printed by the following code?

```
#include <stdio.h>
void f(int *aa, int *bb) {
        *bb = 8;
        aa[1] = bb[2];
        aa = bb;
}
int main(int argc, char *argv[]) {
        int a[5] = { 1, 2, 3, 4, 5 }, *b;
        b = a + 2;
        f(a,b);
        printf("%d %d %d %d %d\n",
                        a[0], a[1], a[2], a[3], a[4]);
        return 0;
}
```

What is printed by the following code?

```
#include <stdio.h>
void g(int *aa, int *bb) {
        bb[2] = aa[-2];
        *aa++ = 17;
        *++aa = 10;
}
 int main(int argc, char *argv[]) {
        int blap[7] = { 1, 2, 3, 4, 5, 6, 7 };
        int *c = blap + 3;
        g(c,blap);
        printf("%d %d %d %d %d %d %d\n",
        blap[0], blap[1], blap[2], blap[3],
             blap[4], blap[5], blap[6]);
        return 0;
}
```

Structures

- Like Java or C# classes, but:
  - No methods or static members
  - No inheritance
  - Everything is implicitly public

```
struct Id_card {
        char name[100];  // Name
        char adresse[100]; // Address
        short int birthyear; // Birth year
        int telefon;   // Telefonnummer
        short int semester; // Semester
} ethz, uniz;

struct Id_card erasmus;
```

- Can be copied using '='
- Should not be compared using '=='.

- Access to members as in Java or C#:

```
ethz.name = "Gustavo";
ethz.telefon = 1234567;
```

- Often referred to by pointers:

```
struct Id_card *pid;
pid = &ethz_student;
(*pid).name = "Gustavo";
(*pid).telefon = 1234567;
```

- Better to use '->' for the latter:

```
pid->name = "Gustavo";
pid->telefon = 1234567;
```

Example structures

Systems@ETH zürich

```
#define MAX_BOX 10;
int main (int argc, char *argv[]) {
        struct Typ_box {
                char content[50];      /* what is in the box */
                int number;            /* how many */
                float price;           /* how much one is */
        };

        float value;
        struct Typ_box box_list[MAX_BOX];

        /* Initialization … */

        /* Total Value */
        for (int i = 0; i < MAX_BOX; i++)
          value += box_list[i].number * box_list[i].price;
        …
        }
```

Unions

Systems@ETH zürich

- Like a struct, but holds only one of a set of alternative values:

```
union u {
   int ival;
   float fval;
   char *sval;
  } my_uval;
```

- Accessed like a struct
- No checking on which value is correct!

Sizes

Systems@ETH zürich

- How much memory does a value take up?
- Depends on machine and compiler!
- Use:

  `sizeof(`type**`)`** or **`sizeof(`**value**`)`**

- Evaluates at compile time to size in bytes
- e.g.

```
struct se *s_ptr =
  (struct se *)malloc(sizeof(struct se));
```

typedef

- Introduces a new type definition
  - New name for a type
- Examples:

```
typedef unsigned uint32_t;
uint32_t ui;
…
typedef int **myptr;
int *p;
myptr mp = &p;
…
typedef struct skbuf skbuf_t;
skbuf_t *sptr;
```

Avoiding Complex Declarations

Systems@ETH zürich

- Use **`typedef`** to build up the declaration

- Instead of **`int (*(*x[3])())[5]`**:

```
typedef int fiveints[5];
typedef fiveints* p5i;
typedef p5i (*f_of_p5is)();
f_of_p5is x[3];
```

- **`x`** is an array of 3 elements, each of which is a pointer to a function returning an array of 5 ints

void

Systems@ETH zürich

- There is a type called **`void`**.
- It has no value.
- Used for:
  - Untyped pointers (to raw memory): "**`void *`**"
  - Declaring functions with no return value (procedures)

- **`sizeof(void)`** shouldn't work
  - Why?
  - (Non-standard) GCC allows **`sizeof(void)==1`**
  - Why?

Examples

Systems@ETH zürich

```
/*  SWAP.C     exchange values */

#include  <stdio.h>
void swap(float *x, float *y);       // prototype

int main(int argc, char *argv[]) {
      float   x, y;
      printf("Please input 1st value: ");
      scanf("%f", &x);
      printf("Please input 2nd value: ");
      scanf("%f", &y);
      printf("Values BEFORE 'swap' %f, %f\n", x, y);
      swap(&x, &y);   /*      address of x, y */
      printf("Values AFTER  'swap' %f, %f\n", x, y);
      return 0;
}

/* exchange values within function */
void swap(float *x, float *y) {
      float   t;
      t = *x;      // *x is value pointed to by x
      *x = *y;
      *y = t;
     printf("Values WITHIN 'swap' %f, %f\n", *x, *y);
}
```

```
/* Compute factorial function */
/* fact(n) = n * (n-1) * … * 2 * 1 */

#include <stdio.h>

int fact(int n)
{
      if (n == 0) {
            return(1);
      } else {
            return(n * fact(n-1));
      }
}

int main(int argc, char *argv[])
{
      int n, m;

      printf("Enter a number: ");
      scanf("%d", &n);
      m = fact(n);
      printf("Factorial of %d is %d.\n", n, m);
      return 0;
}
```

Dynamic memory allocation

- Unlike Java or C#, C allows the programmer to allocate and deallocate memory dynamically
- Functions are in stdlib.h
  malloc
  calloc
  realloc
  free
- Programmer must ensure that memory allocated with **malloc** is freed with **free**!
-

```
typedef struct node {
  int x,z;
  struct node *next;
} NODE;

NODE *nptr;
nptr =(NODE *)malloc(sizeof(NODE))

if (nptr == NULL) {
   printf("No memory - bye bye");
   exit(1);
}
```

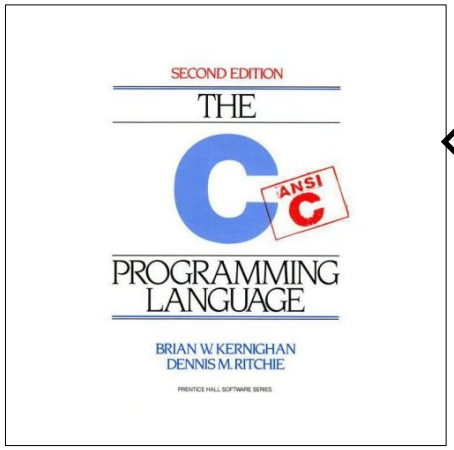
- **malloc** returns a pointer to the allocated memory.
- Return type from malloc is **void *** a
- Good style to cast to appropriate type.
- Memory must be freed with **free(nptr);**

Example dynamic array

```
/* This program simply reads integers
   into a dynamic array until eof. The
   array is expanded as needed */

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

// Initial array size
#define INIT_SIZE 8

int main(int argc, char *argv[])
{
    int num;        // Num. of integers
    int *arr;       // Array of ints.
    size_t arrsize;// Array size
    int m;          // Index
    int in;         // Input number

    // Allocate the initial space.
    arrsize = INIT_SIZE;
    arr =
     (int*)malloc(arrsize*sizeof(int));

  // Read in the numbers.
  num = 0;
  while(scanf("%d", &in) == 1) {
     // See if there's room.
    if(num >= arrsize) {
       // There's not.  Get more.
      arrsize *= 2;
      arr = (int*)realloc(arr,
                  arrsize*sizeof(int));
      if(arr == NULL){
        fprintf(stderr,
            "Allocation failed.\n");
        exit(1);
      }
    }
    // Store the number.
    arr[num++] = in;
  }
  // Print out the numbers
  for(m = 0; m < num; ++m) {
    printf("%d\n", arr[m]);
  }
  free(arr); // Always good to free
  return 0;
}
```

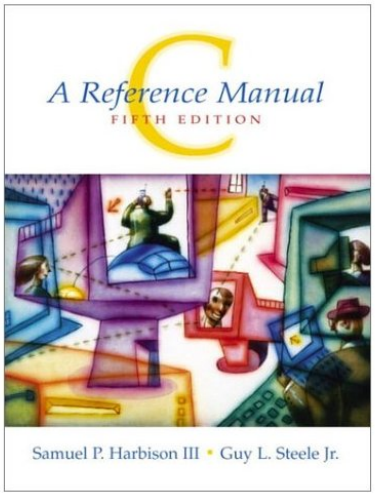
Further reading

Online: http://www.iu.hio.no/~mark/CTutorial/CTutorial.html



Old, but a great tutorial (how I learned C)

Very advanced: all the stuff you *never* wanted to know about C ☺



Definitive.