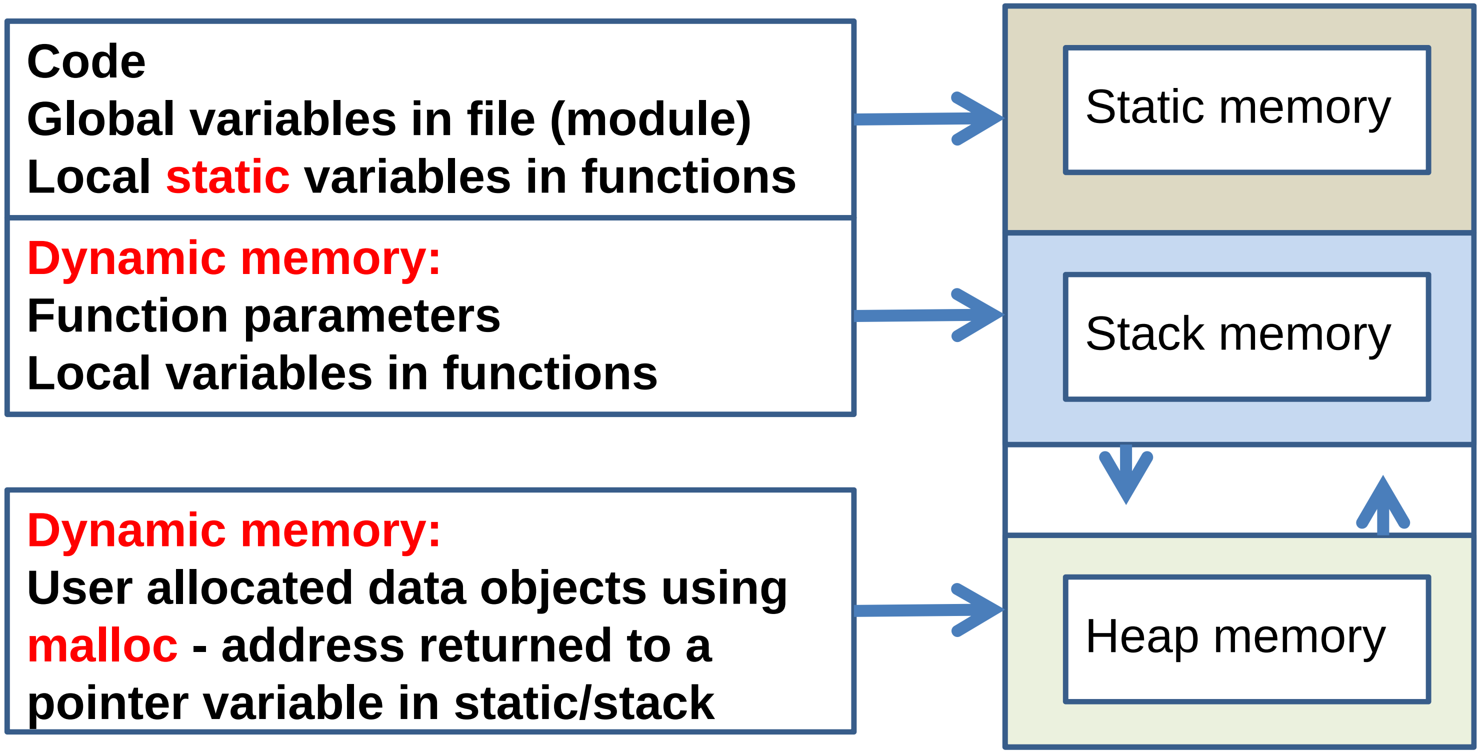
C-types: basic & constructed

C basic types:	**int, char, float, …**
C constructed types:	**pointer, array, struct**

Memory Management



Code
Global variables in file (module)
Local static variables in functions

Dynamic memory:
Function parameters
Local variables in functions

Dynamic memory:
User allocated data objects using malloc - address returned to a pointer variable in static/stack

References

- **Reference** to a variable/function: **name**
- **Reference** to an array element: **index**
- **Reference** to a memory object: **address**
- The value of a **pointer** is an address
  - **Pointers** may refer to
    - Variables, functions:
      - **ptr = &A** **(& = address operator)**
    - Heap allocated data objects
      - **ptr = malloc(sizeof(type))**

Code & Data Objects

- Code Object: a program / function
- Data Object: not always a variable!!!

values

○ A literal integer	e.g.	2
○ A literal char	e.g.	'A'
○ A literal string	e.g.	"ABC"
○ A constant	e.g.	const int k
○ A variable	e.g.	int j
○ A heap object	e.g.	malloc(sizeof(int))

- NB: constants & variables have names (reference)

C Data Types

- **Basic (atomic) data types**
  - char, int, real: float, double
  - integers: short, long, long long, signed, unsigned
  - enum (enumeration) constants
- **Constructed data type instances (variables)**
  - pointer *typename* * ptr;
  - array collection of same type elements

    typename X[size];
  - struct collection of different type elements

    struct tag { members… } X;

    members are: type1 name1; type2 name2; …

C Data Types: Instances

- Variables – type instances – basic types
  - int **j**; char **c**; float **x**;

- Variables – type instances – constructed types

```
int      *        pint;           /* pointer to an int    */
int               array[size];    /* index: 0..(size-1)   */
struct listelem   record;         /* struct variable      */
```

- The general form is <type_name> <variable_name>;
  or <type_definition> <variable_name>;

C Data Types: struct - use

- **Variables – struct definition – example**

```
struct   listelem  {                          /* 2 fields              */
         int                  value;    /* integer field         */
         struct listelem *  next;     /* struct pointer field */
         };
```

```
struct   listelem  listnode;              /* struct variable       */

listnode.value = 3;                         /* field assignment    */
listnode.next   = NULL;
if ( listnode.value == 3 ) { … }         /* using a field          */
```

C Data Types: typedef

- A **typedef** creates an **alias** for another type name
  - ○ E.g. `typedef int listref; /* listref is alias for int */`

- **For constructed types:**

```
typedef int   intarray[size];           /* intarray is the type   */
typedef struct   listelem * listref;    /* pointer to a struct    */
typedef struct   listelem  {            /* listelem is a tag      */
                       int       value;
                       listref   next;  /* not: "struct listelem *" */
                       } listelem;      /* listelem is the type   */
```

C Data Types: typedef struct

```
typedef struct listelem * listref;      /* pointer to a struct   */
typedef     struct     listelem  {      /* listelem is a tag      */
            int       value;
            listref   next;
            } listelem;                 /* listelem is the type */
```

```
listelem   listnode;                    /* struct variable       */
listnode.value = 3;                     /* field assignment      */
listnode.next   = NULL;
if ( listnode.value == 3 ) { … }        /* using a field         */
```

typedef / struct: difference NB!

```
typedef struct listelem * listref;      /* pointer to a struct   */
```

```
typedef    struct    listelem  {      /* listelem is a tag      */
           int      value;
           listref  next;
           } listelem;                /* listelem is the type   */
```

```
           struct    listelem  {      /* listelem is a tag      */
           int      value;
           listref  next;
           } listnode;                /* listnode – variable    */
```

C Data Types: struct pointer

```
typedef struct listelem * listref;      /* pointer to a struct    */
typedef    struct    listelem  {        /* listelem is a tag      */
           int       value;
           listref   next;
           } listelem;                  /* listelem is the type   */
```

```
listref   plistelem;                    /* struct pointer           */
plistelem = malloc(sizeof(listelem));   /* struct instance – heap */
plistelem→value = 3;                    /* field assignment         */
plistelem→next   = NULL;
if (plistelem→value == 3 ) { … }        /* using a field            */
```

C Data Types: linked list

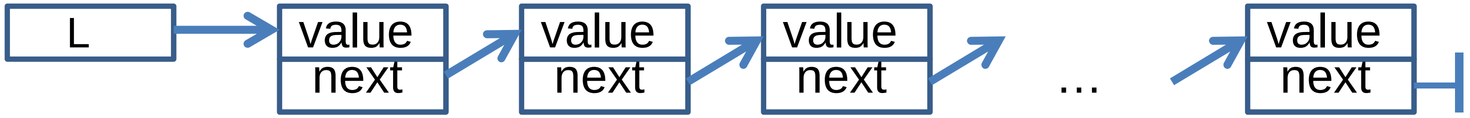
```
typedef struct listelem * listref;      /* pointer to a struct    */
typedef     struct     listelem  {      /* listelem is a tag      */
            int        value;
            listref    next;
            } listelem;                 /* listelem is the type   */
```

```
listref   pnew;                         /* struct pointer          */
pnew = malloc(sizeof(listelem));        /* struct instance – heap */
pnew→value = 3;                         /* field assignment        */
pnew→next   = NULL;
linkin(pnew);                           /* add to linked list      */
```

C Data Types: linked list

```
typedef struct listelem * listref;      /* pointer to a struct    */
typedef     struct     listelem  {      /* listelem is a tag      */
            int       value;
            listref   next;
            } listelem;                 /* listelem is the type   */
```

```
listref   L;                            /* struct pointer         */
```



Use **cons(H,T)** to construct new lists (recursive version)

C Data Types: linked list

- **Comments – recursive sequence code**

listref L; /* reference to the list (ptr) */

create_e(value) returns a pointer of type listref to a new list element created with malloc in the heap

the next field in the list element is of type listref (pointer)

The value of a pointer is an address (remember this!)

List elements are always added and removed at the HEAD (local head!) of the list in the recursive version. cons(H,T)

C Data Types: struct summary

```
typedef struct listelem * listref;     /* pointer to a struct   */
typedef     struct     listelem  {     /* listelem is a tag     */
            int       value;
            listref   next;
            } listelem;                /* listelem is the type */
```

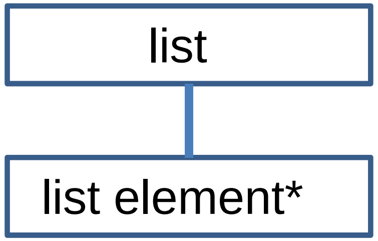
```
/* struct variable */
listelem   listnode;

listnode.value = 3;
listnode.next   = NULL;
```

```
/* struct pointer */
listref   plistelem;
listref = malloc(sizeof(listelem))
plistelem→value = 3;
plistelem→next   = NULL;
```

OO: struct → object or class

- In OO languages, the struct has become an object
  - struct → object: **attributes + methods**
  - Creation is done via a constructor
  - Allocation depends on the language (some use the heap)

- The list would become a list object + a collection of list element objects
- Elements are created with new
- A list is a predefined class


list
list element*