Some History of C

- CPL (???) -> BCPL (c. ‘67)
- BCPL -> C (‘72)
  - struct in ‘73
- K&R C (‘78) or C78
  - stdio lib
  - long/unsigned
- ANSI C/ISO C (‘89-’90)
  - void functions, void ptrs
  - assg/return of structs, return of union
  - enumerated types
  - function prototypes (from C++)
  - i18/UNICODE
  - __STDC__ macro
- C99: -std=<standard>
  - c89,c90,c99,c11,c17,also gnu*
- C11: atomic ops, multi-threading, bounds-checked funcs, C++ compatibility, better UNICODE support…
- C17: very minor changes
- C2x: ?

- C99:
  - Inline funcs, long long, complex, //
  - dynamic arrays

```
#include <stdio.h>
int main() {
  int n;
  scanf("%i", &n);
  int A[n];
  printf ("%lu \n", sizeof(A));
}
```

Include vs namespace

- Include in C/C++ same as import
  - In C/C++ we typically include .h files and libraries
  - If .h file included, it will pick up the corresp lib with the code (impl)
    - eg. include <string.h> => strlen can be used
- Namespace: to create different non-conflicting “islands” of names
  - Eg. Russia::Moscow vs US::Moscow if Russia and US are 2 diff namespaces

Pointers

- For a type **T**, **T**∗ is the type ‘‘pointer to **T**’’
  - a variable of type **T**∗ can hold the address of an object of type **T**
    - HW can only address bytes, not bits. So C/C++ ptrs can point to a byte (char) or a sequence of bytes only
  - **char c = 'a’**
  - **char∗ p = &c**
  - **char** ppc = &p**
  - **int* p2IntVal = &IntVal**
  - **int Arr[100]**
  - **int* p2Arr= Arr**
  - **int* p2p2Arr[100] // array of 100 ptrs to integers**
  - Can do arithmetic on pointers
- void* : ptr to obj of unknown type
  - For “low-level” programming; typically exist at the very lowest level of the system, where real hardware resources are manipulated
- Aliases

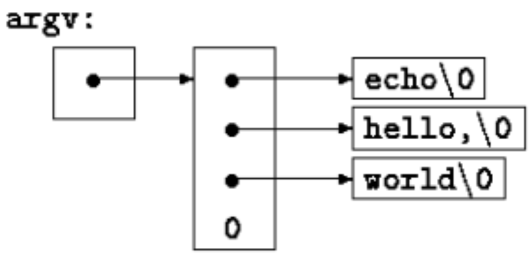
int *pi = nullptr

Earlier, 0 or NULL used
In C: NULL is (void*)0 but
this not OK in C++

argc, argv

```
%cat argvc.c
#include <stdio.h>
int main (int argc, char *argv[]) {
  int count;
  printf ("This program was called with \"%s\".\n",argv[0]);
  if (argc > 1) {
     for (count = 1; count < argc; count++)
        printf("argv[%d] = %s\n", count, argv[count]);
  else printf("The command had no other arguments.\n"); }
  return 0;
}
%cc argvc.c
%./a.out 5m6 7778 99
This program was called with "./a.out".
argv[1] = 5m6
argv[2] = 7778
argv[3] = 99
```

```
%echo hello, world
```



K &R

varargs

K&R

```
#include <stdarg.h>
//minprintf: minimal printf with varargs
void minprintf(char *fmt, ...) {
  char *p, *sval; int ival; double dval;
  va_list ap;
  /* points to each unnamed arg in turn */

  va_start(ap, fmt);
  /* make ap point to 1st unnamed arg */

  for (p = fmt; *p; p++) {
    if (*p != '%'){putchar(*p); continue; }
    switch (*++p) {
      case 'd': ival = va_arg(ap, int);
                printf("%d", ival); break;
      case 'f': dval = va_arg(ap, double);
                printf("%f", dval); break;
      case 's': sval = va_arg(ap, char *);
                for( ;*sval; sval++)
                    putchar(*sval); break;
      default: putchar(*p); break; }
  }
  va_end(ap); /* clean up when done */
}
```

Casting

```
void* pInt = p2IntVal;
*pInt;          //error
++pInt;         //error
int* pInt2 = static_cast<int*>(pint);

double d
double* pd = &d
int* pid= static_cast<int*>(pd); //error
```

```
#include <stdio.h>

int main() {

 double d = 5.0;
 double* pd = &d;

 void* vpd= pd;
 int* pid=static_cast<int*>(vpd);

 printf("%d   %d \n", *pid, *(pid+1));
}
```

```
#include <stdio.h>

int main() {

  typedef double T;

  T t[10], u[10];
  T* pt = t;
  T* pu = u;

  printf("%d \n", *pu); //warning !
  printf("%d \n", *(pu+1));  //warning !

  int x = *pu;  // implicit conv from double2int
  int y = *(pu++);
  int z = *(++pt);

  printf("%d %d %d \n", x, y, z);
}
```

Pointer arithmetic
T t[10], u[10];
T* pt = t;
T* pu = u;
t++, t-- incr/decr by sizeof(T)
t-u, t+u???

Cast in C vs C++'s
Regular cast ("C")
Static_cast (betw related classes)
    but no runtime check
Dynamic cast (RTTI needed at runtime)
Const_cast (+ or – const as needed;
    eg. const actual to a non-const formal
Reinterpret_cast (interpret anything as anything else!)

References vs Pointers

- C has pointers: & for taking address and * for dereferencing
  - free and malloc
  - Pointer arithmetic
- C++ has both pointers and refs
  - int *p = &i // note & on RHS: p pointer
  - int& S = a // note & on LHS: S ref
  - void* p = &i OK but not void& S= a
  - A ref cannot be updated or reassigned
    - `int a=10; int b=100;`
    - `void* ap = &a; // OK`
    - `int* ap1 = &b;`
    - `ap1 = &a; // OK`
    - `void& vr = a; //error`
    - `int& ar = a; // a <> nullptr!`
    - `int& ar = b; //error`
    - `int& br; // error: need init!`
    - `ar++; ar+=20; // both OK but eq to a++`
    - `for (int& x: vect) f(x);` // iterates over all elems of vect
  - Refs cannot be used to implement lists,trees.
- Java has more powerful (explicit) refs but not pointers
  - Can implement lists, etc.
- Python has implicit refs

```
#include <iostream>
using namespace std;
int main() {
    int i = 10; // simple or ordinary variable.
    int* p = &i; // single pointer
    int** pt = &p; // double pointer
    int*** ptr = &pt; // triple pointer
    // All above ptrs differ in value they store or point to
    cout << "i = " << i << "\t"
            << "p = " << p << "\t"
            << "pt = " << pt << "\t"
            << "ptr = " << ptr << '\n';
    int a = 5; // simple or ordinary variable
    int& S = a; S = 7;  // a is now 7
    int& S0 = S; S0 = 8; // a is now 8, S also
    int& S1 = S0; S1 = 5; // a is now 5, S and S0 too
    cout << "a = " << a << "\t"
            << "S = " << S << "\t"
            << "S0 = " << S0 << "\t"
            << "S1 = " << S1 << '\n';
    // All the above references do not differ in their
    // values as they all refer to the same variable.
}
```

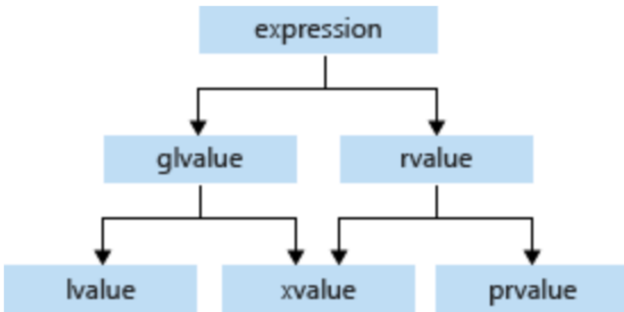
Prints: `i = 10 p = 0x16fd575`**`cc`**` pt = 0x16fd575`**`c0`**` ptr = 0x16fd575`**`b8`**

`a = 5 S = 5 S0 = 5 S1 = 5`

lvalue, rvalue, call by value, call by ref, …

- a *lvalue* is an object reference and a *rvalue* is a value
  - A lvalue is an expression that yields an object reference, eg. a variable name, an array subscript reference, a dereferenced pointer, or a function call that returns a reference.
  - A lvalue always has a defined region of storage, so it has an address.
  - A rvalue is an expression that is not a lvalue. Eg. literals, the results of most operators, and function calls that return nonreferences.
  - A rvalue does not necessarily have any storage associated with it. “No address”
- C++ borrows *lvalue* from C:
  - only a *lvalue* can be used on LHS of an assignment stmt
  - rvalue is a logical counterpart for an expression that can be used only on RHS of an assignment.
- a function is strictly a lvalue, but only used for calling the function, or determining the function’s address. Hence, mostly, the term lvalue means object lvalue
- C parameters: call by value (except arrays)
  - Use explicit pointers to get call by ref
- C++ parameters: call by value, call by ref
  - Also C model…

C++



Identifier Bindings or Scope

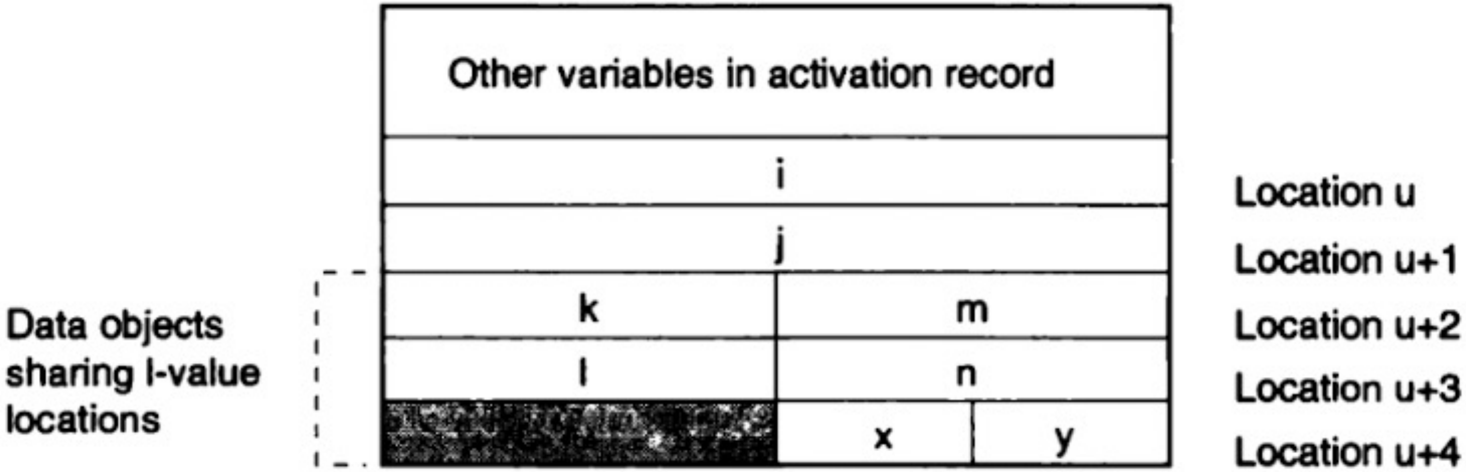
- Static: based on source
  - Need a stack of activation frames but “display” base addressing enough
- Dynamic: based on execution
  - Need a stack of activation frames but need to search them LIFO for an identifier

Methods: similar (static and dynamic)

- C (static)
- C++ (dynamic but only thru class derivations)
- Python (dynamic + thru class derivations)
  - Method resolution order

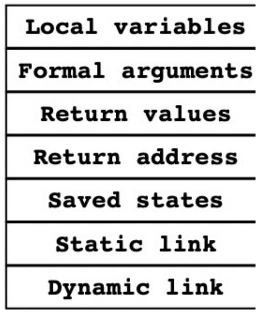
C block str and runtime

```
real func1 (params) {
    int i,j;
      …
    { int k,l; ….}
    {int m,n;
        …
        { int x; …}
        { int y; …}
    }
}
```



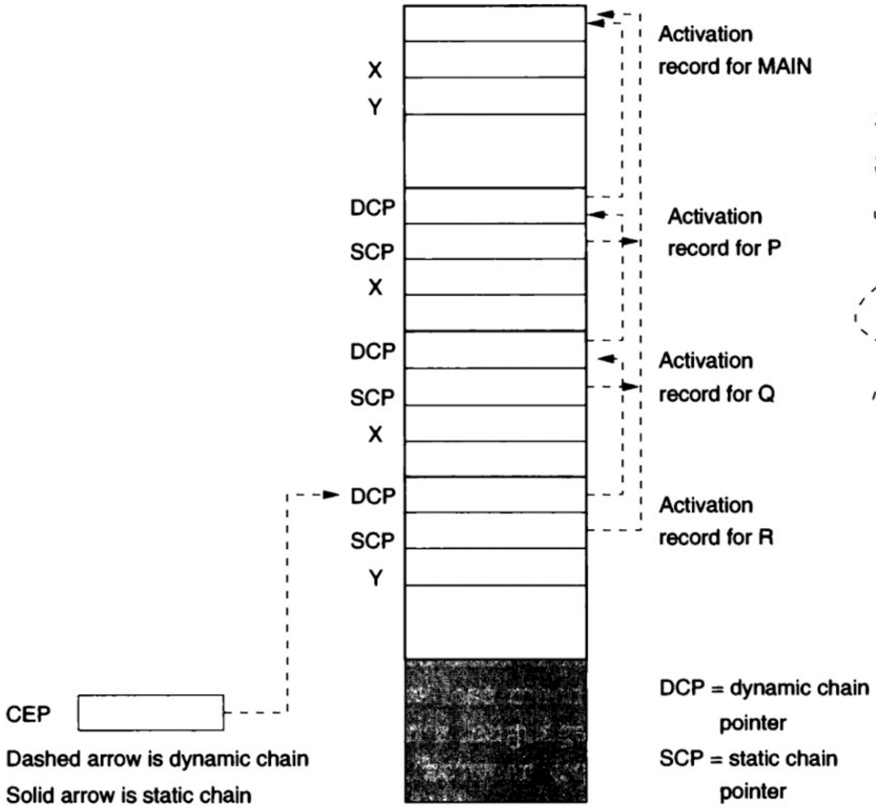
Block Structured Langs Runtime Data Str

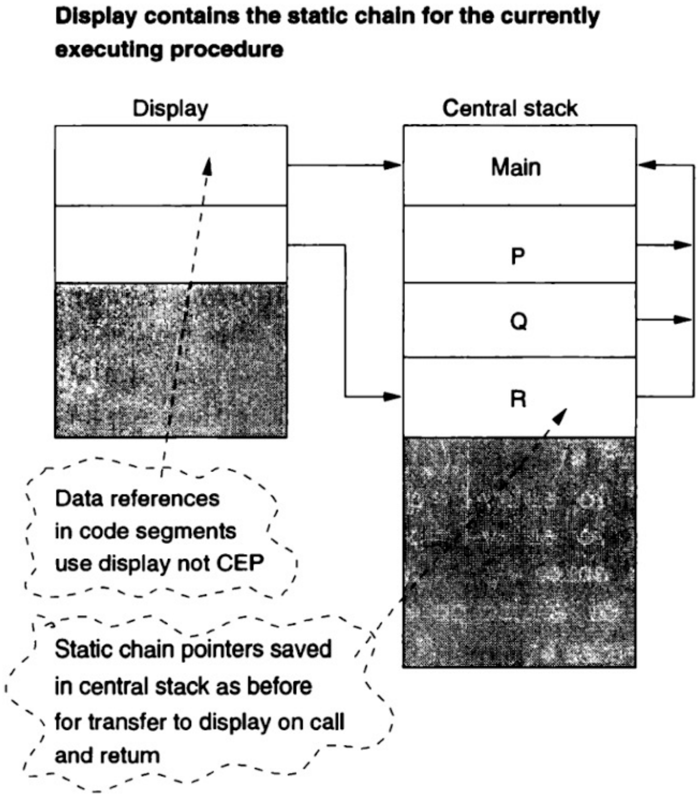
```
program Main;
    var X, Y: integer;
    procedure R;
        var Y: real;
        begin
        ...
        X := X+1; { Nonlocal reference to X }
        ...
        end {R};
    procedure Q;
        var X: real;
        begin
        ...
        R; { Call procedure R }
        ...
        end {Q};
    procedure P;
        var X: Boolean;
        begin
        ...
        Q; { Call procedure Q }
        ...
        end {P};
begin { begin Main }
...
P; { Call procedure P }
    ...
end.
```



AR

Pratt

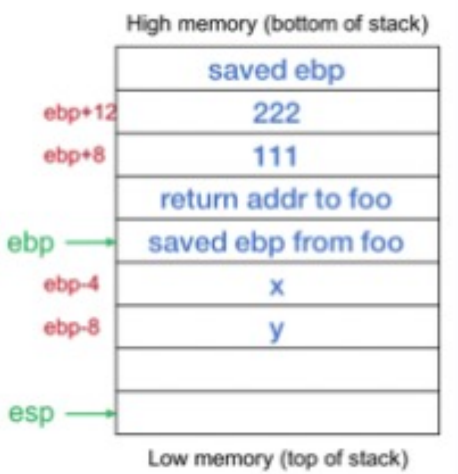




Display not needed in C!
Only local, global and static vars

Static scope: use SCP
Dynamic scope: use DCP

```
%cat try.c
void bar(int a, int b) {
   int x, y;
  x = 555; y = a+b;
}
void foo(void) {
  bar(111,222);
}
%gcc -S -m32 try.c
// 32 bit arch, gen asm
```




```
bar:     # --------- start of the function bar()
  pushl  %ebp # save the incoming frame pointer
  movl   %esp, %ebp # set the frame pointer to the current top of stack
  subl   $16, %esp # increase the stack by 16 bytes (stacks grow down)
  movl   $555, -4(%ebp) # x=555 a is located at [ebp-4]
  movl   12(%ebp), %eax # 12(%ebp) is [ebp+12], which is the second parameter
  movl   8(%ebp), %edx # 8(%ebp) is [ebo+8], which is the first parameter
  addl   %edx, %eax # add them
  movl   %eax, -8(%ebp) # store the result in y
  leave  #
  ret    #

foo:     # --------- start of the function foo()
  pushl  %ebp # save the current frame pointer
  movl   %esp, %ebp # set the frame pointer to the current top of the stack
  subl   $8, %esp # increase the stack by 8 bytes (stacks grow down)
  movl   $222, 4(%esp) # this is effectively pushing 222 on the stack
  movl   $111, (%esp) # this is effectively pushing 111 on the stack
  call   bar # call = push the instruction pointer on the stack and branch to foo
  leave  # done
  ret    #
```

Krzyzanowski

Security problem? Buffer overflow…

Procedures and functions in Algol-like languages

activation record dynamically allocated upon procedure call

activation record includes:
•**dynamic link** -- a pointer to the caller's activation record
•**formal** parameters
•**static link** -- a pointer to the environment of definition of the procedure or function (another activation record)

subprogram invocation:

- pass parameters
- save the caller's state
- set up the dynamic link
- set up the static link
- start executing the callee

return from the subprogram:

- restore the caller's state
- resume the execution of the caller

subprograms are scope defining constructs.

local variables:
•simple variables are stored at a known offset from the base of the activation record
•arrays of dynamic size are accessed indirectly, through an array descriptor at a known location

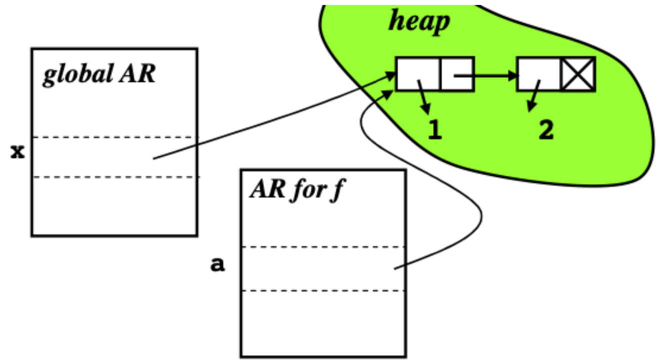
nonlocal variables:
•need to access the activation record of the procedure where this variable is local, using the static links

variables are represented as <static nesting level, offset> pairs in the symbol table.
static nesting level: number of containing scopes surrounding the the environment in which the name is defined

static distance: calculate the difference between the static nesting level of the use of the variable and that of its definition. To access a nonlocal variable v with static distance d, the compiler sets up code to traverse d static links, then load v at the known offset in that activation record

Optimization: statistically, most variables are local or global. Local variable access is already efficient. Also make it efficient to access the global stack frame, for example, by putting a pointer to the global stack frame in a register.

Parameter Binding Strategies



- Call-by-value
  - Array is treated as a ref but structs copied in C
  - Call-by-ref simulated by taking addr of objects in C but using call-by-value
- Call-by-value-result (Fortran or in multicore langs)
  - in-only; out-only
- Call-by-ref (C thru ptrs and &, Java thru explicit refs, C++ thru ptrs and (weak) ref)
- Call-by-sharing (Python: mutable vs immutable)
  - mutations to mutable objects within the called function are visible to the caller

```
def f(a_list):
  a_list.append(1)
m = []
f(m)
```

  - print(m) #prints 1
  - But with “a_list = a_list + [1]” instead of append, prints []

Call-by-sharing

```
x=[1,2]
def f(a):
   pass
f(x)
```

- Call-by-name: actual not evaluated until point of use
  - Sum(i, 1, 100, V[i]) (sum of V’s elements)
  - Sum(i, l, m, Sum(j, l, n, A[i,j]))
  - Sum(i,1,100,i) (sum of integers)
  - Sum(i,1,100,i*i) (sum of sq)
  - Problem: swap(i,x[i]) with std def of swap
- Call by need: used in functional languages

```
real procedure Sum(k, l, u, ak)
  value l, u; integer k, l, u; real ak;
  comment k and ak are passed by name;
begin
  real s; s := 0;
  for k := l step 1 until u do
    s := s + ak; Sum := s
end;
```