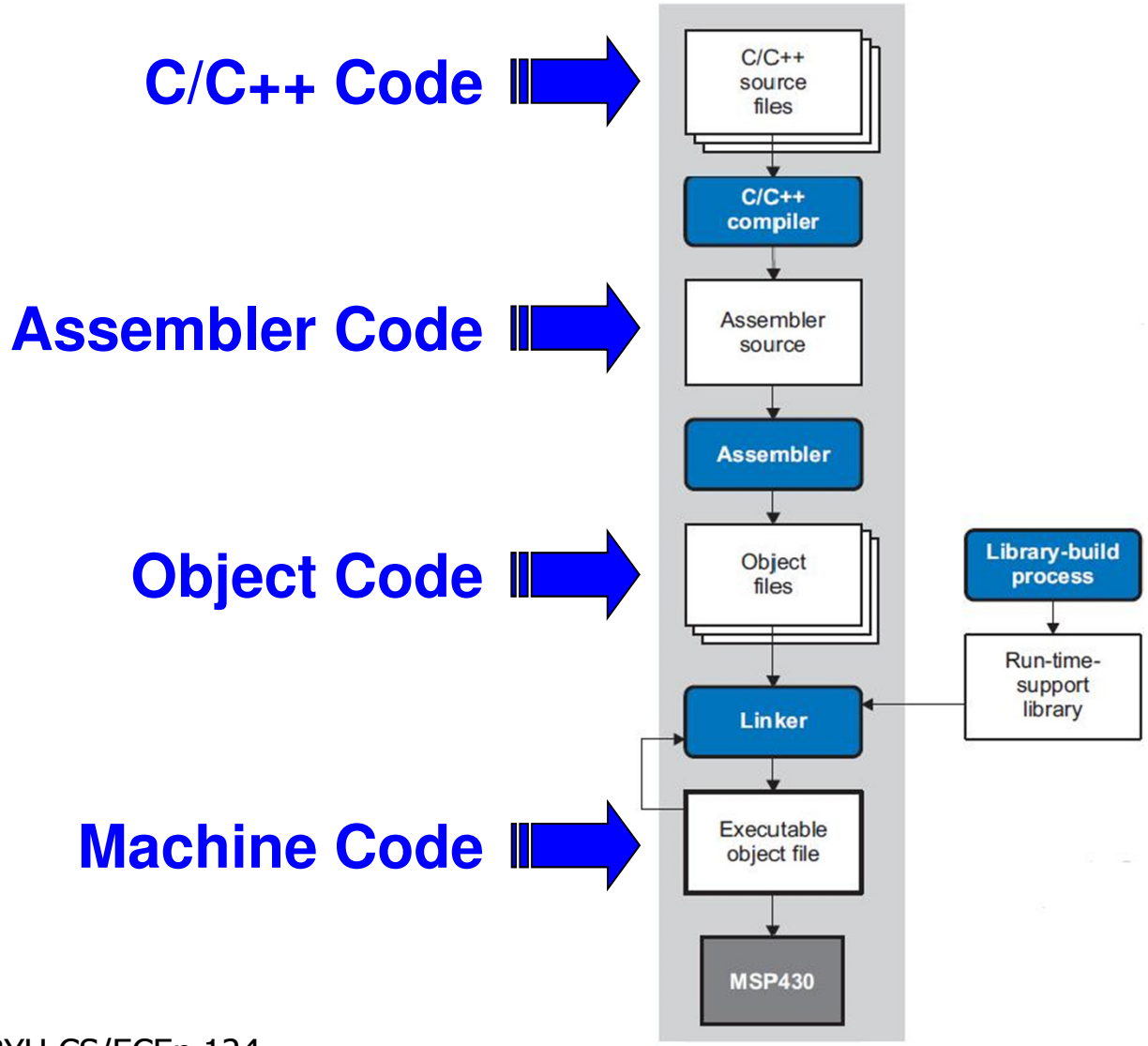
```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

-- Dishonorable mention, Obfuscated C Code Contest, 1984.
(Author requested anonymity.)

The C Language

Compiling a C Program



A First Program

```
//*************************
//  blinky.c: Software T
//*************************
#include "msp430x22x4.h"

void main(void)
{
  int i = 0;
  WDTCTL = WDTPW + WDTHOLD;  // Stop watchdog
  P4DIR |= 0x40;             // P4.6 output
  for (;;)                   // loop
  {
    P4OUT ^= 0x40;           // toggle P4.6
    while (--i);             // delay
  }
}
```

Tells compiler to *use* all the definitions found in the msp430x22x4.h library.
A .h file is called a *header* file and contains definitions and declarations.

All programs must have a *main()* routine. This one takes no arguments (parameters).

Stop WD w/Password

Set P4.6 as output

Loop forever

Delay 65,536

Toggle P4.6

Comments

- Use lots of comments

```
/* This is a comment */
 // This is a single line comment
```

- Comment each procedure telling:

```
/*-------------------------------*
 * ProcedureName - what it does  *
 * Parameters:                   *
 *   Param1 - what param1 is     *
 *   Param2 - what param2 is     *
 * Returns:                      *
 *   What is returned, if anything *
 *-------------------------------*/
```

- Use lots of white space (blank lines)

Indenting Style

- Each *new scope* is indented 2 spaces from previous
- Put { on end of previous line, or start of next line
- Line matching } up below

Style is something of a personal matter.

Everyone has their own opinions…

What is presented here is similar to that in common use and a good place to start...

Style 1

```
if(a < b) {
  b = a;
  a = 0;
}
else {
  a = b;
  b = 0;
}
```

Style 2

```
if(a < b)
{
  b = a;
  a = 0;
}
else
{
  a = b;
  b = 0;
}
```

The C Preprocessor

- #define *symbol code*
  - The preprocessor replaces *symbol* with *code* everywhere it appears in the program below

```
#define NUMBER_OF_MONKEYS 259
#define MAX_LENGTH 80
#define PI 3.14159
```

- #include *filename.h*
  - The preprocessor replaces the #include directive itself with the contents of header file *filename.h*

```
#include <stdio.h>      /* a system header file */
#include "myheader.h"   /* a user header file */
```

- Macros
  - Pass arguments

```
#define add(x,y) x+y
#define concatenate(x,y) x##y
```

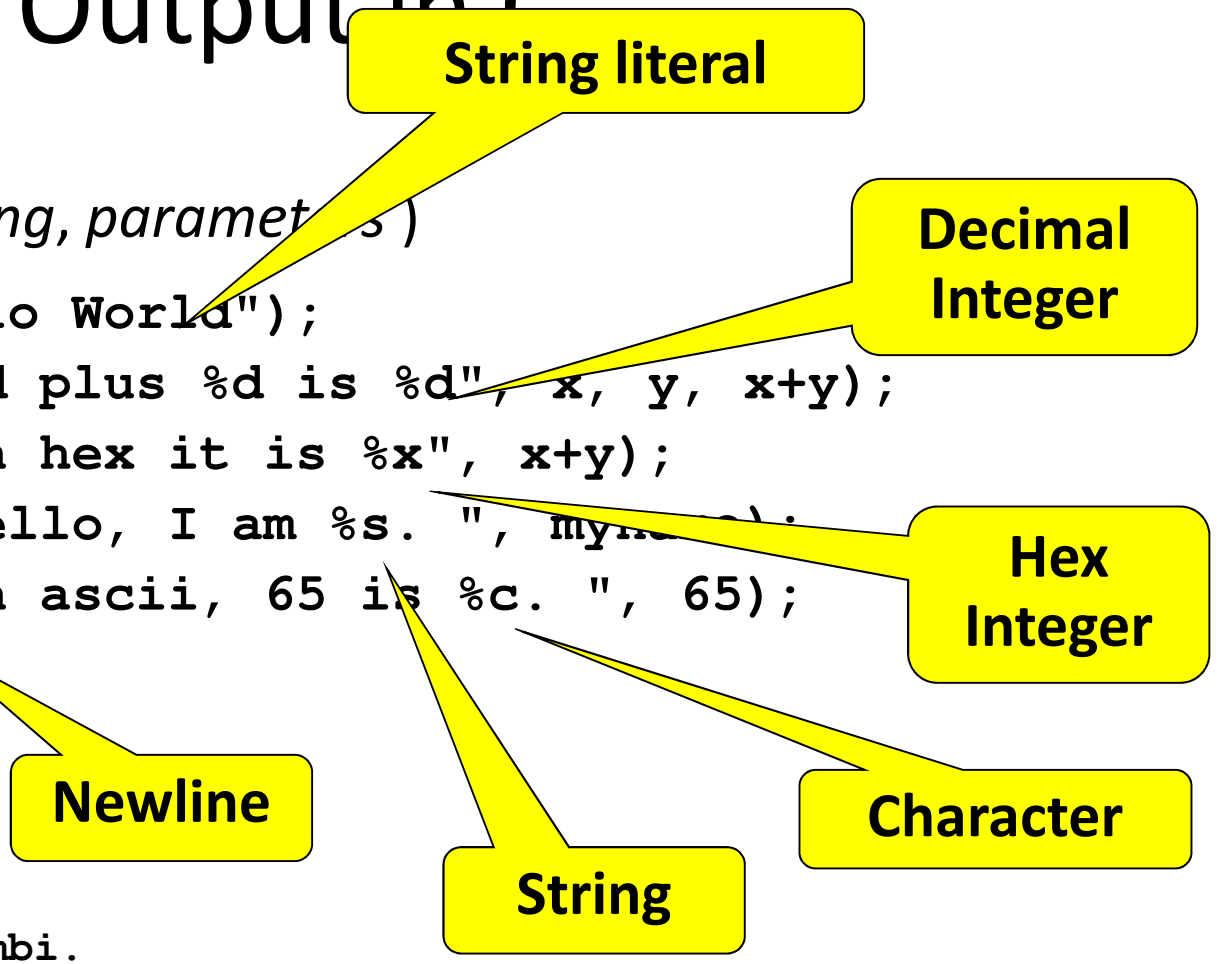
Output in C

- printf(*format_string*, *parameters*)

```
printf("Hello World");
printf("\n%d plus %d is %d", x, y, x+y);
printf("\nIn hex it is %x", x+y);
printf("\nHello, I am %s. ", myName);
printf("\nIn ascii, 65 is %c. ", 65);
```

- Output:

```
Hello world
5 plus 6 is 11
In hex it is b
Hello, I am Bambi.
In ascii, 65 is A.
```



MSP430 C Variable Data Types

Type	Size	Representation	Minimum	Maximum
char, signed char	8 bits	ASCII	-128	127
unsigned char	bool 8 bits	ASCII	0	255
short, signed short	16 bits	2's complement	-32768	32767
unsigned short	16 bits	Binary	0	65535
int, signed int	16 bits	2's complement	-32768	32767
unsigned int	16 bits	Binary	0	65535
long, signed long	32 bits	2's complement	-2,147,483,648	2,147,483,647
unsigned long	32 bits	Binary	0	4,294,967,295
enum	16 bits	2's complement	-32768	32767
float	32 bits	IEEE 32-bit	1.175495e-38	3.4028235e+38
double	32 bits	IEEE 32-bit	1.175495e-38	3.4028235e+38
long double	32 bits	IEEE 32-bit	1.175495e-38	3.4028235e+38
pointers, references	16 bits	Binary	0	0xFFFF
function pointers	16 bits	Binary	0	0xFFFF

Variable Declarations

```
int  i,j,k;                 // declaring more than one variable
int  i1, i2, i3, c3po;      // numbers OK, except for first letter

int  bananas = 10;          // using an initializer

int  monkey_count = 0;      // two ways of doing ...
int  monkeyCount = 0;       // ... multi-word names

int ab, Ab, aB, AB;         // case sensitive names
int _compilerVar;           // compiler uses _ as first char

char newline = '\n';        // a character with an initializer
char lineBuffer[32];        // an array of 32 chars (a string)

double bananasPerMonkey;          // floating point declarations
double hugeNumber = 1.0E33;       // positive exponent
double tinyNumber = 1.0E-33;      // negative exponent
double fractionThing = 3.33333; // no exponent
```

Scope: Local versus Global

- Extent of a variable/function's availability in a program
- Local Variables (automatic)
  - Declared at the beginning of a block
  - Stored in activation record on the stack
  - Scope is from point of declaration to the end of the block
  - Un-initialized

```
{
  // begin block
  int chimp;
  ...
}
```

- Global Variables (static)
  - Declared outside of a function
  - Stored in Global Data Section of memory
  - Scope is from point of declaration to the end of the program
  - May be initialized to zero

```
int chimp;
{
  // begin block
  ...
}
```

Literals/ Constants

- Literal Values
  - Unnamed constant values used in programs
  - **`area = 3.14159 * radius * radius;`**
- Constant Variables
  - Variable declarations prefixed with the `const` qualifier
  - Immutable named variables
  - **`const double pi = 3.14159;`**
- Symbolic Values
  - Created using preprocessor directive **`#define`**
  - **`#define PI 3.14159`**
- How are the above the same?
- How are the above different?

Operators and Expressions

- Expressions are formed by combining variables with operators and ALWAYS return a single value in C.

```
i = 5;
i < j;
a = (a < b);
```

- Operators
  - Assignment –
    - changes the values of variables
  - Arithmetic –
    - add, subtract, multiply, divide
  - Bitwise –
    - AND, OR, XOR, NOT, and shifts on Integers
  - Relational –
    - equality, inequality, less-than, etc.
  - Logical –
    - AND, OR, NOT on Booleans
  - Increment/Decrement

C supports a rich set of operators that allow the programmer to manipulate variables

Arithmetic / Relational Operators

- Arithmetic Operators
  - Add (+), subtract (–), multiply (*), divide (/)
    - Integer; 5/3 = 1 (truncated to int)
    - Floating point : 5.0 / 3.0 = 1.66666666
  - Modulus (%)
    - Integer; remainder after integer division; 5 % 3 = 2
- Relational operators return Boolean values:
  - 0 if relation is FALSE
  - 1 if relation is TRUE
  - Comparisons

x **==** y	equality
x **!=** y	inequality
x **<** y	less-than
x **<=** y	less-than-or-equal
x **>** y	greater-than
x **>=** y	greater-than-or-equal

```
x + y
x - y
x * y
x / y
x % y
```

Bitwise Operators

- Perform bitwise logical operations across individual bits of a value.
  - AND &
  - OR |
  - XOR ^
  - NOT ~

    (1's complement)

```
    x : 1 0 1 0 (binary)
    y : 1 1 0 0 (binary)

x & y : 1 0 0 0 (binary)

x | y : 1 1 1 0 (binary)

x ^ y : 0 1 1 0 (binary)

   ~x : 0 1 0 1 (binary)
```

- Shifts are bitwise operators
  - SHIFT LEFT <<
  - SHIFT RIGHT >>

`x << y` **shift x y-places to the left (add zeros)**

`x >> y` **shift x y-places to the right (sign extend)**

Logical Operators

- Logical operators evaluate to Boolean
  - AND &&
  - OR ||
  - NOT ! (2's complement)

```
10 && 20 → 1
10 && 0 → 0
```

- Don't confuse with Bitwise operators
  - Operate on Boolean inputs and produce Boolean outputs
  - Boolean inputs (how values are interpreted):
    - **Value not equal to zero → TRUE**
    - **Value equal to zero → FALSE**

```
if( 'a' <= x <= 'z' ) statement; // wrong!
if(('a' <= x) && (x <= 'z')) statement;

if(!x) statement;
if(x == 0) statement;      } Same
if(x) statement;
if(x != 0) statement;      } Same
```

Operator Precedence/Associativity

OPERATORS		ASSOCIATIVITY
`( )  [ ]  ->  .`		left to right
`!  ~  ++  --  +  -  *  &  (type)  sizeof`		right to left
`*  /  %`		left to right
`+  -`		left to right
`<<  >>`	Bitwise	left to right
`<  <=  >  >=`	Relational	left to right
`==  !=`	Relational	left to right
`&`	Bitwise	left to right
`^`	Bitwise	left to right
`|`	Bitwise	left to right
`&&`	Logical	left to right
`||`	Logical	left to right
`?:`		right to left
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=`		right to left
`,`		left to right

Combined Assignment Operators

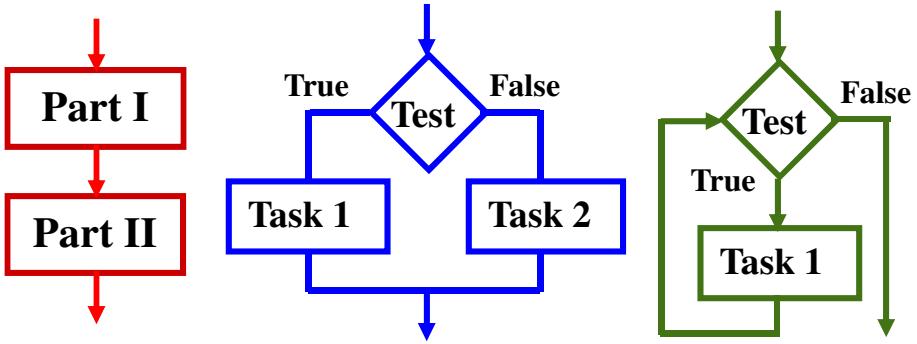
- Arithmetic and bitwise operators can be combined with the assignment operator.

```
x += y;   ⇒   x = x + (y);
x -= y;   ⇒   x = x - (y);
x *= y;   ⇒   x = x * (y);
x /= y;   ⇒   x = x / (y);
x %= y;   ⇒   x = x % (y);
x &= y;   ⇒   x = x & (y);
x |= y;   ⇒   x = x | (y);
x ^= y;   ⇒   x = x ^ (y);
x <<= y;  ⇒   x = x << (y);
x >>= y;  ⇒   x = x >> (y);
```

Note: All of the expression on the right is considered parenthesized.

Control Structures

- We looked at three constructs for systematic decomposition:
  - The sequential construct
  - The conditional construct
  - The iteration construct



- C has many conditional and iteration constructs:
  - if, if-else
  - switch
  - for
  - while, do-while

The if-else Statement

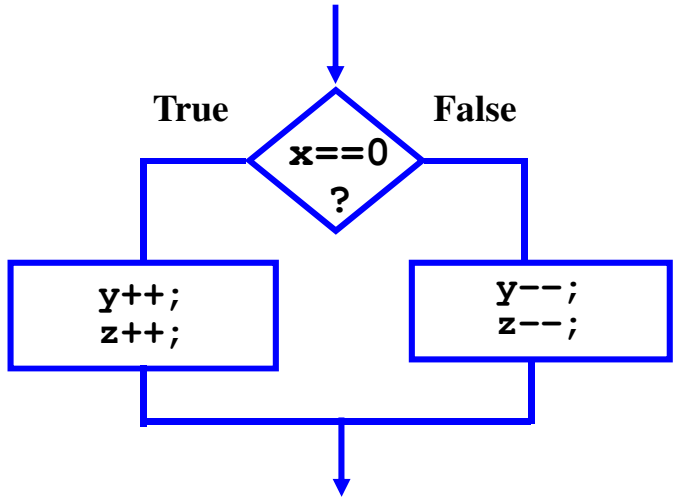
- Perform if-action if a condition is true. Otherwise, perform else-action.

- Form:

```
if (expression)
      statement1
   else
      statement2
```

- Example:

```
if (x)
{
      y++;
      z++;
}
else
{
      y--;
      z--;
}
```



The if-else statement

- You can connect conditional constructs to form longer sequences of conditional tests:

```
if (expression1)
        statement1
    else if (expression2)
        statement2
    else if (expression3)
        statement3
    else
        statement4
```

- An **else** is associated with the closest unassociated **if**.

```
if (expression1)
    if (expression2)
 statement2
    else
        statement3
```

```
if (expression1) {
   if (expression2)
 statement2
   else
 statement3
}
```

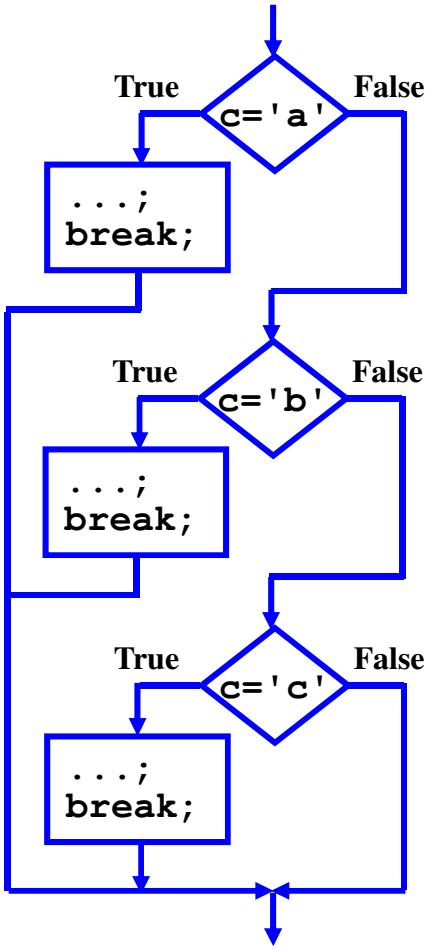
```
if (expression1) {
   if (expression2)
 statement2
}
else
   statement3
```

The switch Statement

- Performs actions based on a series of tests of the same variable.
- Form:

```
switch (expression)
    {
        case const-expr :    statements
        case const-expr :    statements
        case const-expr :    statements
        default :            statements
    }
```

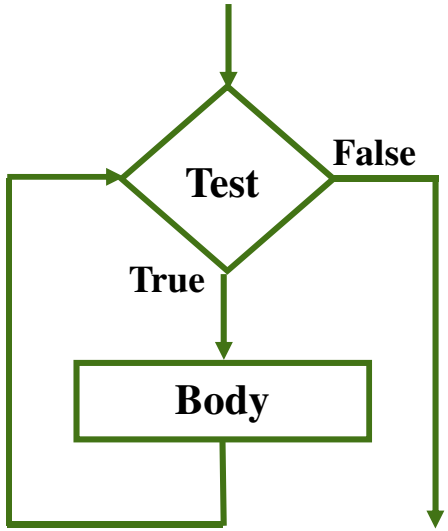
- The **break** statement causes an immediate exit from the switch.
- Because **cases** serve only as labels, execution falls through to the next unless there is explicit action to escape.



while loop

- Check test (sentinel) at *beginning* of loop
  - May (or may not) execute loop
- Form:

```
while (expression)
    statement
```



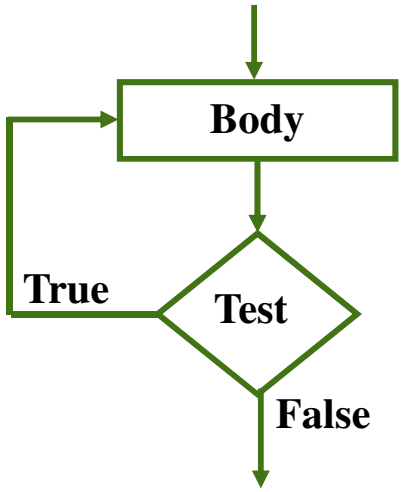

```
void main(void)
{
   // Print digits from 7 down to 1
   int a = 7;
   while (a)
   {
      printf("%c\n", a + '0');
      a--;
   }
   printf("All done...\n");
}
```

```
        main:
0x9e64: 8031 0006       SUB.W   #0x0006,SP
0x9e68: 40B1 0007 0004  MOV.W   #0x0007,0x0004(SP)
0x9e6e: 9381 0004       TST.W   0x0004(SP)
0x9e72: 2410            JEQ     (C$DW$L$main$2$E)
        C$DW$L$main$2$B, C$L1:
0x9e74: 40B1 A398 0000  MOV.W   #0xa398,0x0000(SP)
0x9e7a: 403F 0030       MOV.W   #0x0030,R15
0x9e7e: 511F 0004       ADD.W   0x0004(SP),R15
0x9e82: 4F81 0002       MOV.W   R15,0x0002(SP)
0x9e86: 12B0 A056       CALL    #printf
0x9e8a: 8391 0004       DEC.W   0x0004(SP)
0x9e8e: 9381 0004       TST.W   0x0004(SP)
0x9e92: 23F0            JNE     (C$L1)
        C$L2, C$DW$L$main$2$E:
0x9e94: 40B1 A39C 0000  MOV.W   #0xa39c,0x0000(SP)
0x9e9a: 12B0 A056       CALL    #printf
0x9e9e: 5031 0006       ADD.W   #0x0006,SP
0x9ea2: 4130            RET
```

do-while loop

- Check test (sentinel) at *end* of loop
  - Always executes loop once
- Form:

```
do
    statement
while (expression);
```

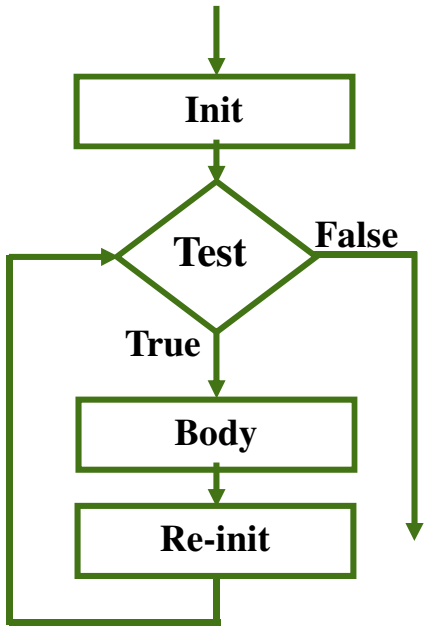



```
void main(void)
{
 // Print digits from 7 down to 1
   int a = 7;
   do
   {
      printf("%c\n", a + '0');
      a--;
   } while (a);
   printf("All done...\n");
}
```

```
        main:
0x9ee0: 8031 0006       SUB.W   #0x0006,SP
0x9ee4: 40B1 0007 0004  MOV.W   #0x0007,0x0004(SP)
        C$DW$L$main$2$B, C$L1:
0x9eea: 40B1 A392 0000  MOV.W   #0xa392,0x0000(SP)
0x9ef0: 403F 0030       MOV.W   #0x0030,R15
0x9ef4: 511F 0004       ADD.W   0x0004(SP),R15
0x9ef8: 4F81 0002       MOV.W   R15,0x0002(SP)
0x9efc: 12B0 A050       CALL    #printf
0x9f00: 8391 0004       DEC.W   0x0004(SP)
0x9f04: 9381 0004       TST.W   0x0004(SP)
0x9f08: 23F0            JNE     (C$L1)
        C$DW$L$main$2$E:
0x9f0a: 40B1 A396 0000  MOV.W   #0xa396,0x0000(SP)
0x9f10: 12B0 A050       CALL    #printf
0x9f14: 5031 0006       ADD.W   #0x0006,SP
0x9f18: 4130            RET
```

for loop

for Statement



- Check test at beginning of loop
  - May (or may not) execute loop
- Form:

`for` **(*expr*$_1$; *expr*$_2$; *expr*$_3$)**
statement

where ***expr*$_1$ executes at the beginning of loop (init)**
***expr*$_2$ is a relational expression (test)**
***expr*$_3$ executes at the end of loop (re-init)**

```
void main(void)
{
   // Print digits from 7 down to 1
   int a;
   for (a = 7; a > 0; a--)
   {
      printf("%c\n", a + '0');
   }
   printf("All done...\n");
}
```

```
        main:
0x9e64: 8031 0006       SUB.W   #0x0006,SP
0x9e68: 40B1 0007 0004  MOV.W   #0x0007,0x0004(SP)
0x9e6e: 9391 0004       CMP.W   #1,0x0004(SP)
0x9e72: 3810            JL      (C$DW$L$main$2$E)
        C$DW$L$main$2$B, C$L1:
0x9e74: 40B1 A398 0000  MOV.W   #0xa398,0x0000(SP)
0x9e7a: 403F 0030       MOV.W   #0x0030,R15
0x9e7e: 511F 0004       ADD.W   0x0004(SP),R15
0x9e82: 4F81 0002       MOV.W   R15,0x0002(SP)
0x9e86: 12B0 A056       CALL    #printf
0x9e8a: 8391 0004       DEC.W   0x0004(SP)
0x9e8e: 9391 0004       CMP.W   #1,0x0004(SP)
0x9e92: 37F0            JGE     (C$L1)
        C$L2, C$DW$L$main$2$E:
0x9e94: 40B1 A39C 0000  MOV.W   #0xa39c,0x0000(SP)
0x9e9a: 12B0 A056       CALL    #printf
0x9e9e: 5031 0006       ADD.W   #0x0006,SP
0x9ea2: 4130            RET
```

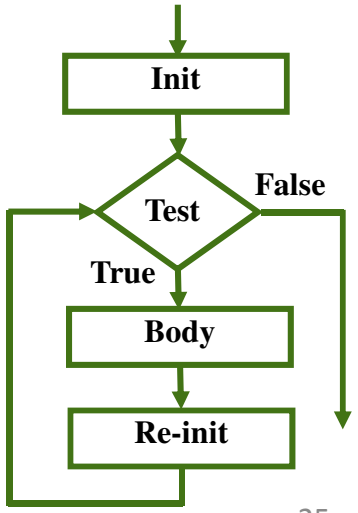
for loop

- A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand.
- Note: The commas that separate function arguments, variables in declarations, etc., are *not* comma operators, and do not guarantee left to right evaluation.
- Example:

```
for (i=0, j=10, k=-1;  i < j; j--)
{
    statement
}
```

- What do the following do?

```
if (x = y) y = 10;
for (;;) { body }
while (TRUE) { body }
```



Loops

- Loop style
  - To a large degree, all iteration constructs can be used interchangeably.
  - Stylistically, different constructs make sense for different situations.
  - The type of loop you choose will convey information about your program to a reader.
- Infinite Loops
  - The following loop will never terminate:

```
x = 0;
while (x < 10)
  printf("%d ", x);
```

  - Loop body does not change condition, so test never fails.
  - This is a common programming error that can be difficult to find.
- Break and Continue
  - **break** and **continue** can be used with iteration construct or with a switch construct
  - **break** exits the innermost loop or switch
  - **continue** restarts the innermost loop or switch
  - Both generate an unconditional branch in the assembly code

Horrors! GOTOs and Labels

- A goto statement is used to branch (transfer control) to another location in a program.
  - Only be used within the body of a function definition.
  - "The goto statement is never necessary; it can always be elimitated (sic) by rearranging the code."
  - Use of the goto statement violates the rules of structured programming.
- Label statement can be used anywhere in the function, above or below the goto statement.

```
int found = 0;
for (i=0; i<10; i++)
{  for (j=0; j<10; j++)
   {  if (same…)
      {
         found = 1;
         break;
      }
   }
   if (found) break;
}
if (found){ ... }
else { ... }
```

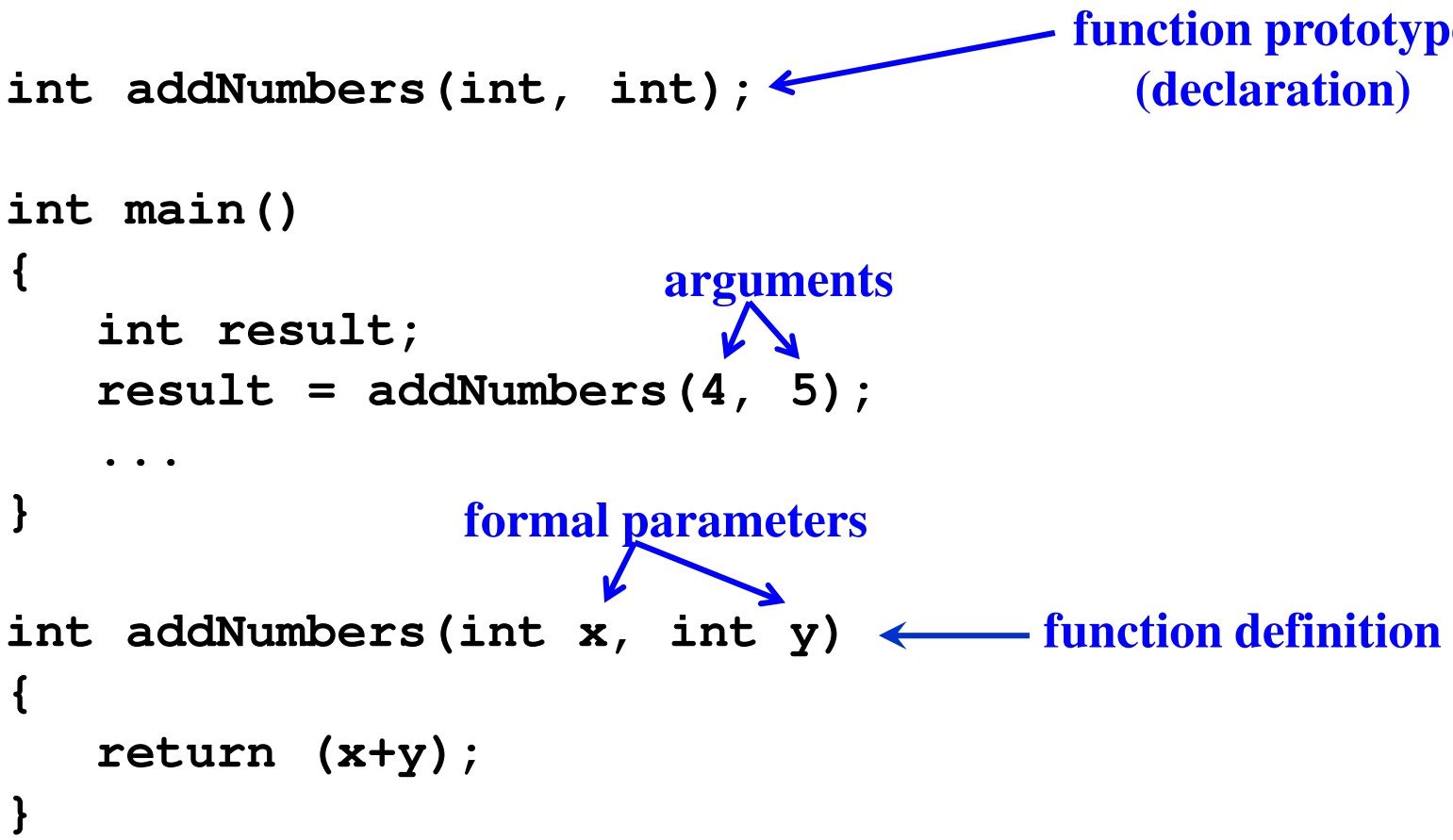
```
for (i=0; i<10; i++)
{  for (j=0; j<10; j++)
   {  if (same…) goto FOUND;
   }
}
NOT_FOUND:

FOUND:
```

Functions

- Functions have been included in all programming languages since the very early days of computing.
  - Support for functions is provided directly in all instruction set architectures.
  - In other languages, called procedures, methods, subroutines, ...
- C is heavily oriented around functions.
  - A C program is organized as a collection of functions.
  - Every C statement belongs to a function
  - All C programs start and finish execution in the function **`main`**
  - Functions may call other functions which, in turn, call more functions.
- Functions
  - Provides abstraction
    - hide low-level details
    - give high-level structure to program, easier to understand overall program flow
    - enables separable, independent development
  - Used for Systematic Decomposition
    - Functions break large computing tasks into smaller ones.
    - Functions enlarge the set of elementary building blocks.

C Function Example



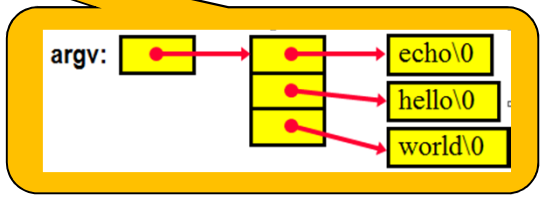
The *main* Function

- The **main** function
  - must be present in every C program
  - is "called" by the operating system.
  - exits the program with a return statement.
- The prototype of **main** is pre-declared as:

Return code for system

# of command line arguments

```
int main (int argc, char *argv[]);
```



- The definition doesn't have to match.

```
int main() { ... }
main() { ... }
```

These are OK

- The return statement can be omitted.

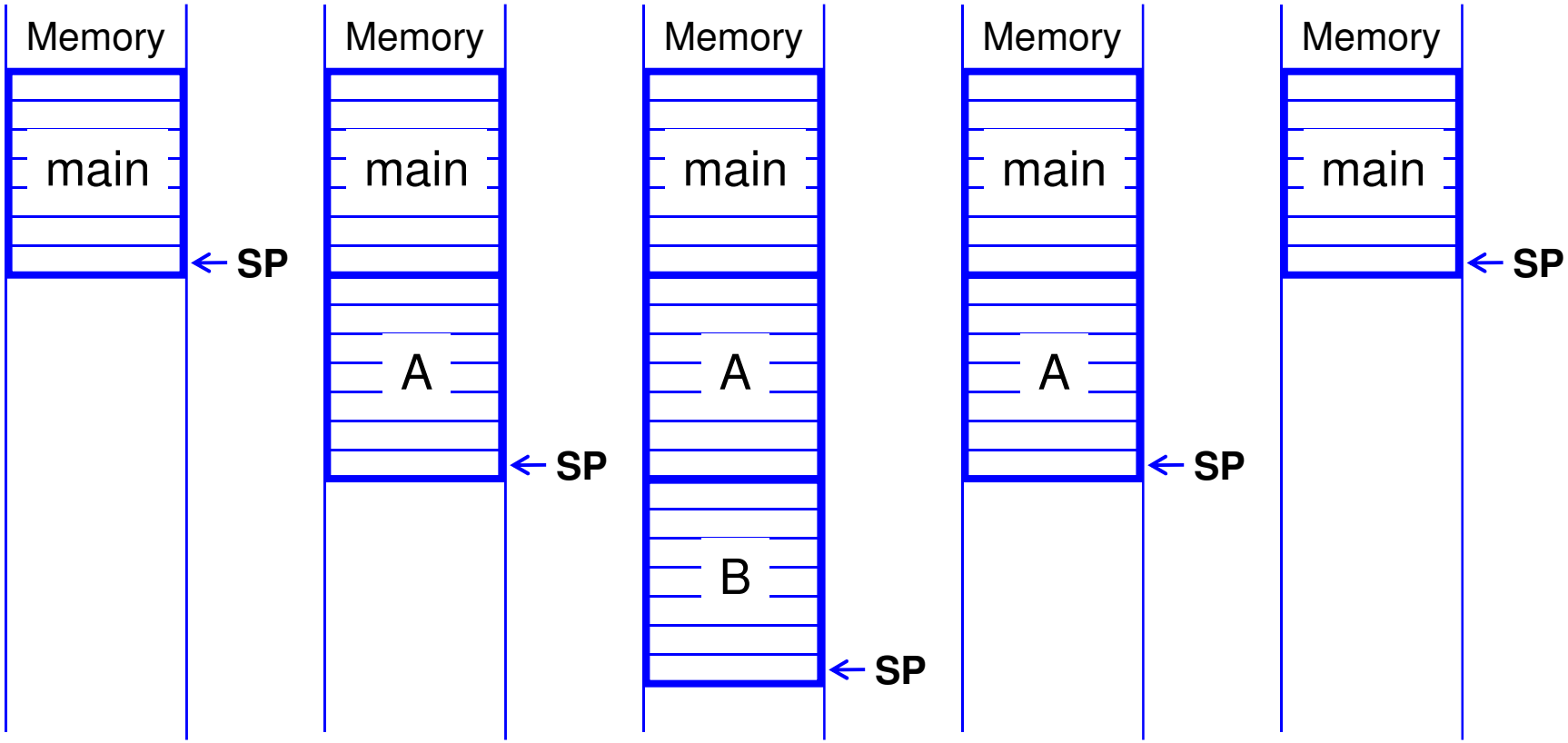
Activation Records

- Function calls involve three basic steps
  - Parameters from caller are passed to the callee.
  - Callee does the task
  - Return value is passed back to caller.
- When a function is called, a frame is activated, that is, a local data storage area is allocated on the stack.
- An activation record for a function is a template of the relative positions of the function's local variables within the frame.
- Frame variables are temporary storage and lost when the function returns.
- The stack pointer is used for the frame pointer.
- Data is stored and retrieved via indexed stack instructions.

```
mov.w     r12,0(sp) ; save parameter 1
mov.w     @sp,r12   ; return value
```

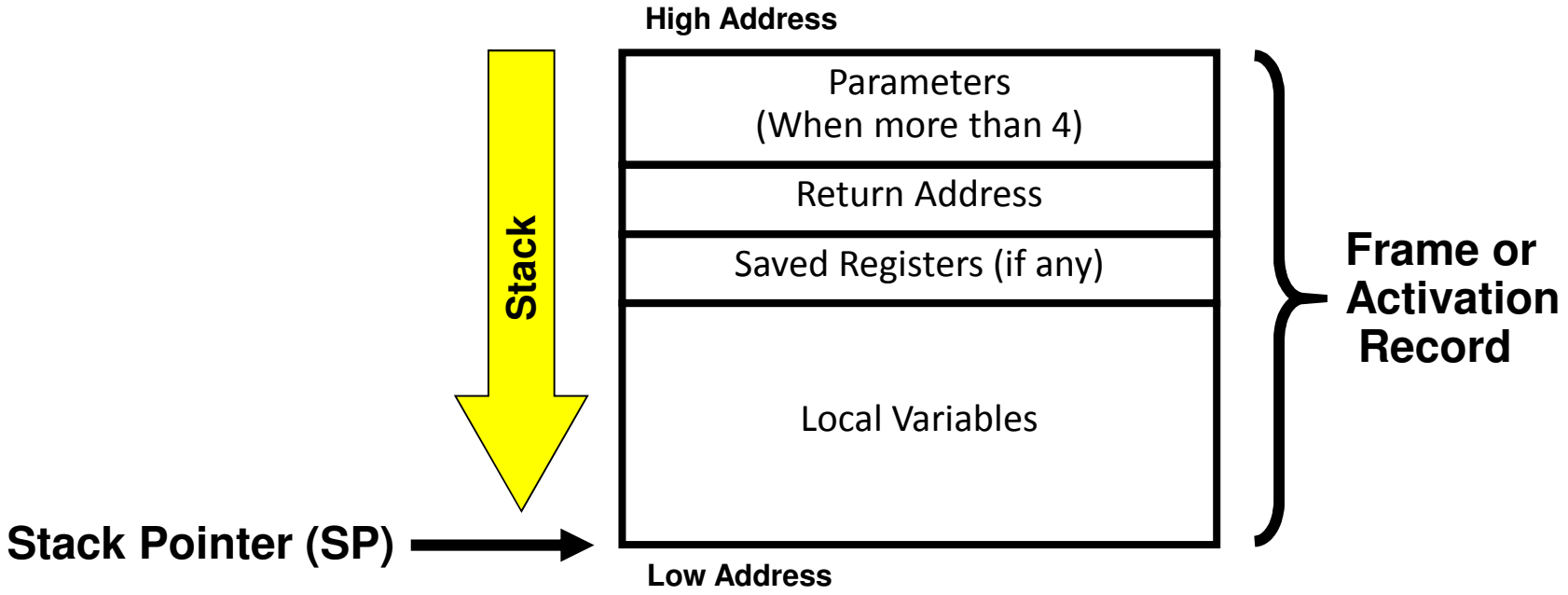
Run-time Stack Operation

Program starts → main calls A → A calls B → B returns to A → A returns to main



Stack Frames / Parameter Passing

- Each function call creates a new stack frame.
- Function arguments are evaluated right to left
  - R12 = 1st argument, R13 = 2nd argument
- The result of the function is returned in R12.

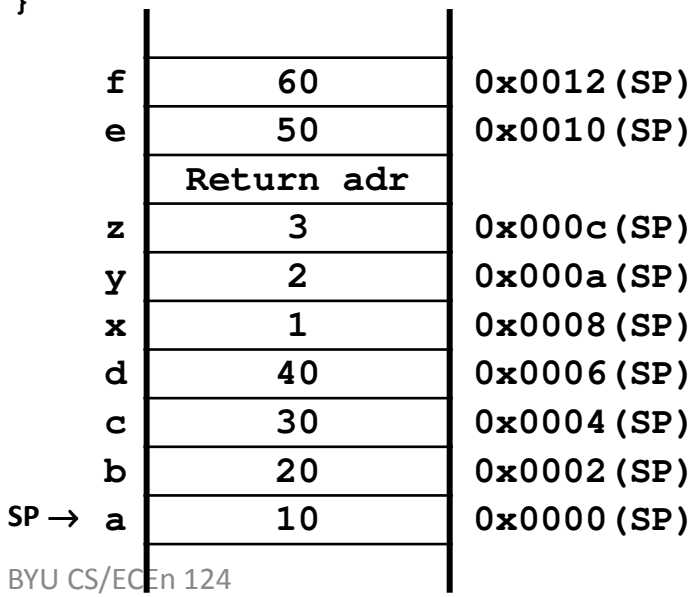


Stack Frames / Parameter Passing

- Example:

```
int func(int a, int b, int c,
         int d, int e, int f)
{
   int x, y, z;
   x = 1;
   y = 2;
   z = 3;
   return a+b+c+d+e+f+x+y+z;
}
int main()
{
   func(10, 20, 30, 40, 50, 60);
   return 0;
}
```

```
 main:
0x8762: 8221              SUB.W   #4,SP
0x8764: 40B1 0032 0000    MOV.W   #0x0032,0x0000(SP)
0x876a: 40B1 003C 0002    MOV.W   #0x003c,0x0002(SP)
0x8770: 403C 000A         MOV.W   #0x000a,R12
0x8774: 403D 0014         MOV.W   #0x0014,R13
0x8778: 403E 001E         MOV.W   #0x001e,R14
0x877c: 403F 0028         MOV.W   #0x0028,R15
0x8780: 12B0 853C         CALL    #func
0x8784: 430C              CLR.W   R12
0x8786: 5221              ADD.W   #4,SP
0x8788: 4130              RET
        func:
0x853c: 8031 000E         SUB.W   #0x000e,SP
0x8540: 4F81 0006         MOV.W   R15,0x0006(SP)
0x8544: 4E81 0004         MOV.W   R14,0x0004(SP)
0x8548: 4D81 0002         MOV.W   R13,0x0002(SP)
0x854c: 4C81 0000         MOV.W   R12,0x0000(SP)
0x8550: 4391 0008         MOV.W   #1,0x0008(SP)
0x8554: 43A1 000A         MOV.W   #2,0x000a(SP)
0x8558: 40B1 0003 000C    MOV.W   #0x0003,0x000c(SP)
0x855e: 411C 0002         MOV.W   0x0002(SP),R12
0x8562: 512C              ADD.W   @SP,R12
0x8564: 511C 0004         ADD.W   0x0004(SP),R12
0x8568: 511C 0006         ADD.W   0x0006(SP),R12
0x856c: 511C 0010         ADD.W   0x0010(SP),R12
0x8570: 511C 0012         ADD.W   0x0012(SP),R12
0x8574: 511C 0008         ADD.W   0x0008(SP),R12
0x8578: 511C 000A         ADD.W   0x000a(SP),R12
0x857c: 511C 000C         ADD.W   0x000c(SP),R12
0x8580: 5031 000E         ADD.W   #0x000e,SP
0x8584: 4130              RET
```



Arrays

- An array is a sequence of like items.
  - Int's, float's, long's, etc
- Like any other variable, arrays must be declared before they are used.
  - General form:

    type variable-name[number_of_elements];
  - The array size must be explicit at compile time – needed to reserve memory space
- Array elements individually accessed.
  - General form:

    ***variable-name*[*index*];**
  - Zero based subscripts
  - No compile-time or run-time limit checking

Initialization of Arrays

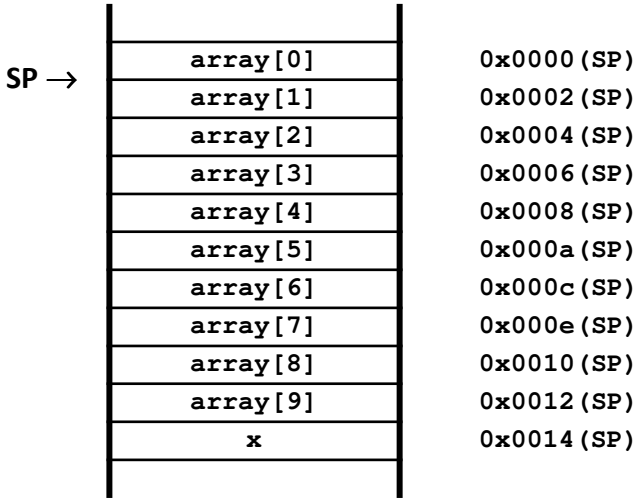
- Elements can be initialized when they are declared in the same way as ordinary variables.
  - type array_name[size]={ list of values };

    ```
    int number[3] = {0, 0, 0};
    ```
  - Remaining uninitialized elements will be set to zero automatically.
- Array declarations may omit the size.

  ```
  int counter[] = {1, 1, 1, 1};
  ```
- Problems with C initialization of arrays
  - No convenient way to initialize selected elements.
  - No shortcut to initialize large number of elements.

Local Array Example

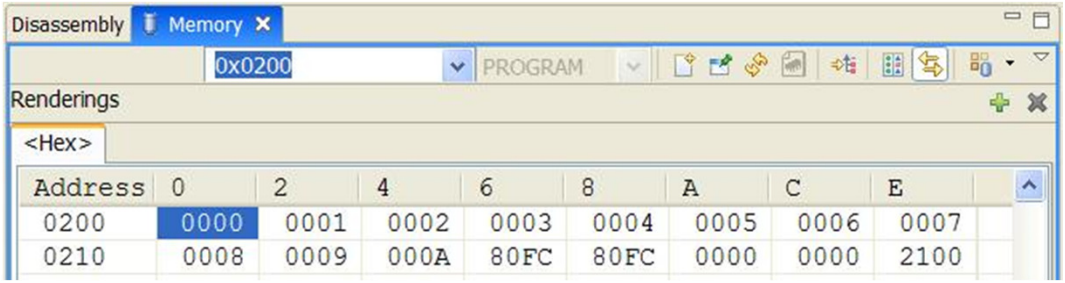
```
int main()
{
   int array[10];
   int x;
   for (x = 0; x < 10; x++)
   {
      array[x] = x;
   }
   return 0;
}
```




```
 main:
0x8040: 8031 0016        SUB.W   #0x0016,SP
0x8044: 4381 0014        CLR.W   0x0014(SP)
0x8048: 90B1 000A 0014   CMP.W   #0x000a,0x0014(SP)
0x804e: 340D             JGE     (C$DW$L$main$2$E)
        C$DW$L$main$2$B, C$L1:
0x8050: 411F 0014        MOV.W   0x0014(SP),R15
0x8054: 5F0F             RLA.W   R15
0x8056: 510F             ADD.W   SP,R15
0x8058: 419F 0014 0000   MOV.W   0x0014(SP),0x0000(R15)
0x805e: 5391 0014        INC.W   0x0014(SP)
0x8062: 90B1 000A 0014   CMP.W   #0x000a,0x0014(SP)
0x8068: 3BF3             JL      (C$L1)
        C$L2, C$DW$L$main$2$E:
0x806a: 430C             CLR.W   R12
0x806c: 5031 0016        ADD.W   #0x0016,SP
0x8070: 4130             RET
```

Global Array Example

```
int array[10];
int x;
int main()
{
   for (x = 0; x < 10; x++)
   {
      array[x] = x;
   }
   return 0;
}
```




```
 main:
0x806a: 4382 0214          CLR.W   &x
0x806e: 90B2 000A 0214     CMP.W   #0x000a,&x
0x8074: 340C               JGE     (C$DW$L$main$2$E)
        C$DW$L$main$2$B, C$L1:
0x8076: 421F 0214          MOV.W   &x,R15
0x807a: 5F0F               RLA.W   R15
0x807c: 429F 0214 0200     MOV.W   &x,0x0200(R15)
0x8082: 5392 0214          INC.W   &x
0x8086: 90B2 000A 0214     CMP.W   #0x000a,&x
0x808c: 3BF4               JL      (C$L1)
        C$L2, C$DW$L$main$2$E:
0x808e: 430C               CLR.W   R12
0x8090: 4130               RET
```

C Strings

- A C string is an array of characters:
  - `char outputString[16];`
- C strings are terminated with a zero byte.
- C strings can be initialized when defined:

```
char outputString[] = "Text";
```

  which is the same as:

```
outputString[0] = 'T';
outputString[1] = 'e';
outputString[2] = 'x';
outputString[3] = 't';
outputString[4] = 0;
```

- C has no string operators.
  - String functions in <string.h> library
  - strcpy, strlen, strcmp, strstr, …

Compiler computes the size of the array (4 + 1 = 5 bytes)

Passing Arrays as Arguments

- C passes parameters to functions by value.
- C passes the address of the 1st element of an array.

```
#define MAX_NUMS 5
int average(int values[])
{
    int i, sum = 0;
    for (i = 0; i < MAX_NUMS; i++)
        sum = sum + values[i];
    return (sum / MAX_NUMS);
}

int main()
{
    int nums[MAX_NUMS] =
        { 1, 2, 3, 4, 5 };
    int mean = average(nums);
    return 0;
}
```