SSD
6

icarnegie

curriculum powered by Carnegie Mellon

Lecture 3

C Programming Model

武汉大学

国际软件学院

Lecture 3

C Programming Model

The Contents cover:

Lecture 3

C Programming Model

The Wonder of Program Execution

- Programming is a difficult task requiring a strict exercise of logic and a capacity to organize substantial amounts of information -- MIRACULOUS !!!
- Programs end up commanding innumerable devices and mechanisms in complicated sequences of tiny events
- These events, which appear insignificant when considered individually, behave according to the specification of our programs when considered together – CEO!

Lecture 3 C Programming Model

The Wonder of Program Execution

- Process *i.e.* $c = a + b$
- Levels of Abstraction
  - classic thinking
  - physical issues

Lecture 3

C Programming Model

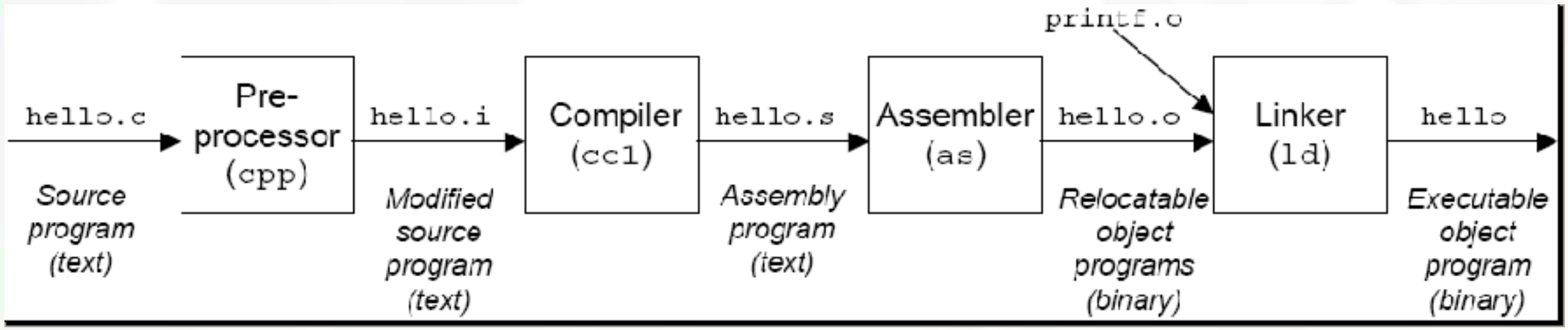
The Wonder of Program Execution

- Levels of Abstraction
  - C programming model is itself an abstraction
  - One can name variables with mnemonic names, instead of having to specify where the variables are stored
  - One can write *array[i],* instead of having to compute the location of the *ith* element of array yourself
  - One can write *c = a + b* instead of having to give the CPU specific instructions on how to carry out the addition.
- C compiler takes care of translating C programs into *machine code*.

Lecture 3 C Programming Model

The Wonder of Program Execution

```
unix> gcc –o hello hello.c
```



Lecture 3

C Programming Model

The Wonder of Program Execution

Code Example:

```
1 int accum = 0;
2
3 int sum(int x, int y)
4 {
5     int t = x + y;
6     accum += t;
7     return t;
8 }
```

```
sum:
   pushl %ebp
   movl %esp,%ebp
   sub   oxc,%esp
movl 12(%ebp),%eax
addl 8(%ebp),%eax
addl %eax,accum
movl %ebp,%esp
popl %ebp
ret
```

Lecture 3

C Programming Model

The Wonder of Program Execution

Code Example:

```
    Disassembly of function sum in file code.o
1 00000000 <sum>:
   Offset   Bytes                    Equivalent assembly language
2     0:    55                        push   %ebp
3     1:    89 e5                     mov    %esp,%ebp
4     3:    8b 45 0c                  mov    0xc(%ebp),%eax
5     6:    03 45 08                  add    0x8(%ebp),%eax
6     9:    01 05 00 00 00 00         add    %eax,0x0
7     f:    89 ec                     mov    %ebp,%esp
8    11:    5d                        pop    %ebp
9    12:    c3                        ret
10   13:    90                        nop
```

```
sum:
  pushl %ebp
  movl %esp,%ebp
  sub   oxc,%esp
  movl 12(%ebp),%eax
  addl 8(%ebp),%eax
  addl %eax,accum
  movl %ebp,%esp
  popl %ebp
  ret
```

Lecture 3

C Programming Model

The Wonder of Program Execution

Code Example:

```
    Disassembly of function sum in executable file prog
1 080483b4 <sum>:
2  80483b4:  55                          push   %ebp
3  80483b5:  89 e5                       mov    %esp,%ebp
4  80483b7:  8b 45 0c                    mov    0xc(%ebp),%eax
5  80483ba:  03 45 08                    add    0x8(%ebp),%eax
6  80483bd:  01 05 64 94 04 08           add    %eax,0x8049464
7  80483c3:  89 ec                       mov    %ebp,%esp
8  80483c5:  5d                          pop    %ebp
9  80483c6:  c3                          ret
10 80483c7:  90                          nop
```

Lecture 3

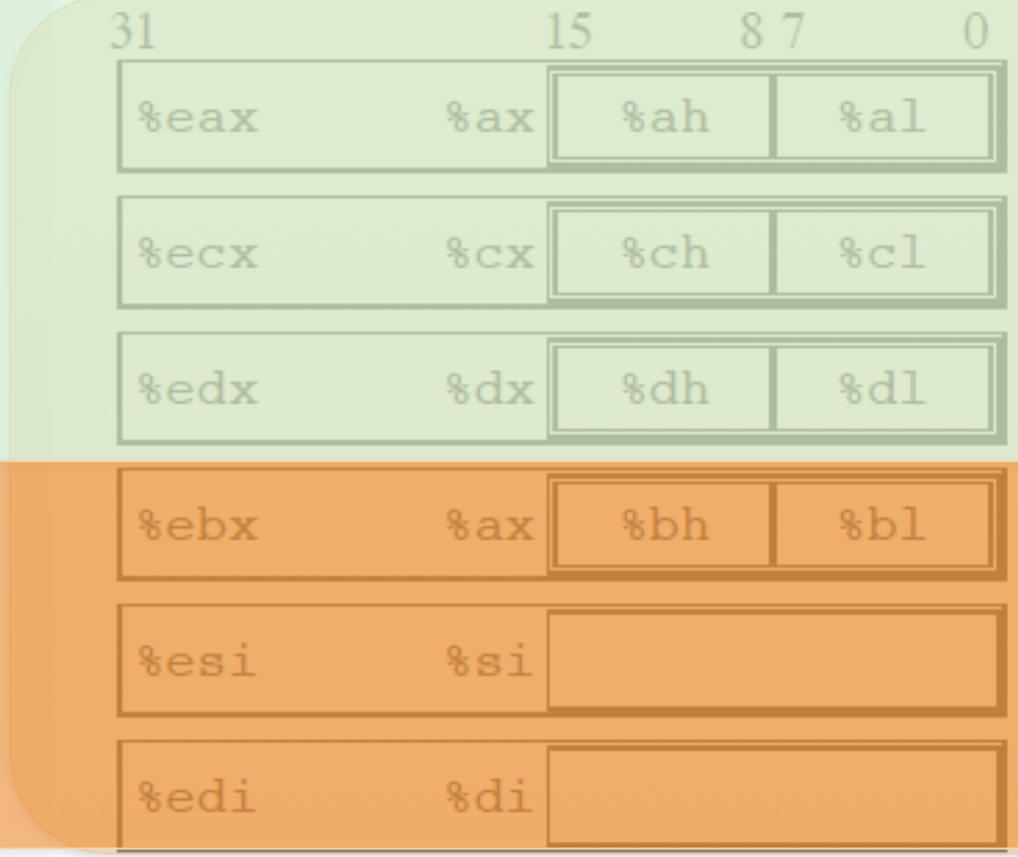
C Programming Model

The Wonder of Program Execution

Machine-Level Code: Integer Registers

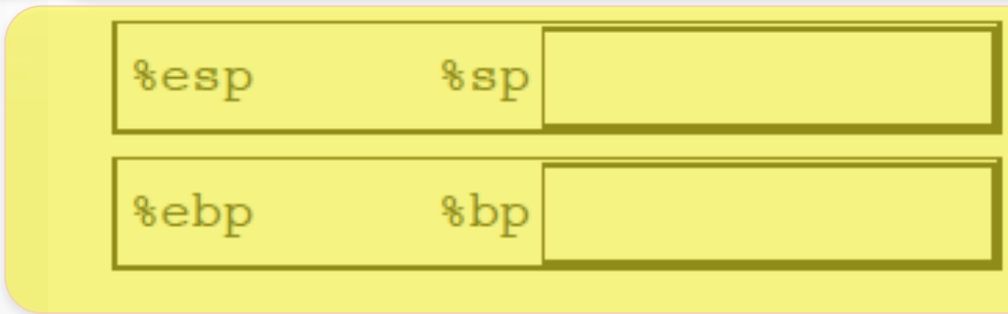
EXPAND

Caller save



Callee save

Commonly used



Stack Pointer

Procedure related

Frame Pointer

Lecture 3

C Programming Model

The Wonder of Program Execution

Machine-Level Code

- The program counter (called *%eip*) indicates the address in memory of the next instruction to be executed.
- The integer register file contains eight named locations storing 32-bit values. These registers can hold addresses (corresponding to C pointers) or integer data. Some registers are used to keep track of critical parts of the program state, while others are used to hold temporary data, such as the local variables of a procedure.
- The condition code registers hold status information about the most recently executed arithmetic instruction. These are used to implement conditional changes in the control flow, such as is required to implement if or while statements.
- The floating-point register file contains eight locations for storing floating-point data

Lecture 3

C Programming Model

The Wonder of Program Execution

- Execution as Physical Process
- We think
  - of variables and data types rather than of memory chips;
  - of algorithms rather than of moving data among those chips;
  - of program statements rather than of where and how these statements are stored.
- We don't give much thought to *execution process* because we don't have to

Lecture 3

C Programming Model

The Wonder of Program Execution

- One reason to distinguish the program from its execution is that the translation is ***not*** quite perfect – ***NOT*** expected results
- For instance:
  - Out of memory
  - pointers
- Compilers and Debuggers
- The Visual C++ Environment

Lecture 3

C Programming Model

Some misunderstandings for C programmers

- Focused on logic models instead of programming models
- Memory is infinite
- Do not care about deployment of memories
- Ignore physical process of programs
- Surprised on:
  - a + b <> (a+b)
  - Pointers misleading to wrong way
  - This is not what I want!

Lecture 3

C Programming Model

Some misunderstandings for C programmers

```
#define ARRAY_SIZE 10
void natural_numbers (void) {
    int i;
    int array[ARRAY_SIZE];

    i=1;

    while ( i <= ARRAY_SIZE) {
    array[i] = i -1;
    i = i +1;
  }
}
```

Lecture 3

C Programming Model

Some misunderstandings for C programmers

```
#define ARRAY_SIZE 10
void natural_numbers (void) {
    int i;
    int array[ARRAY_SIZE];

    i=0;

    while ( i < ARRAY_SIZE+2) {
    array[i] = i -1;
    i = i +1;
  }
}
```

Lecture 3

C Programming Model

Some misunderstandings for C programmers

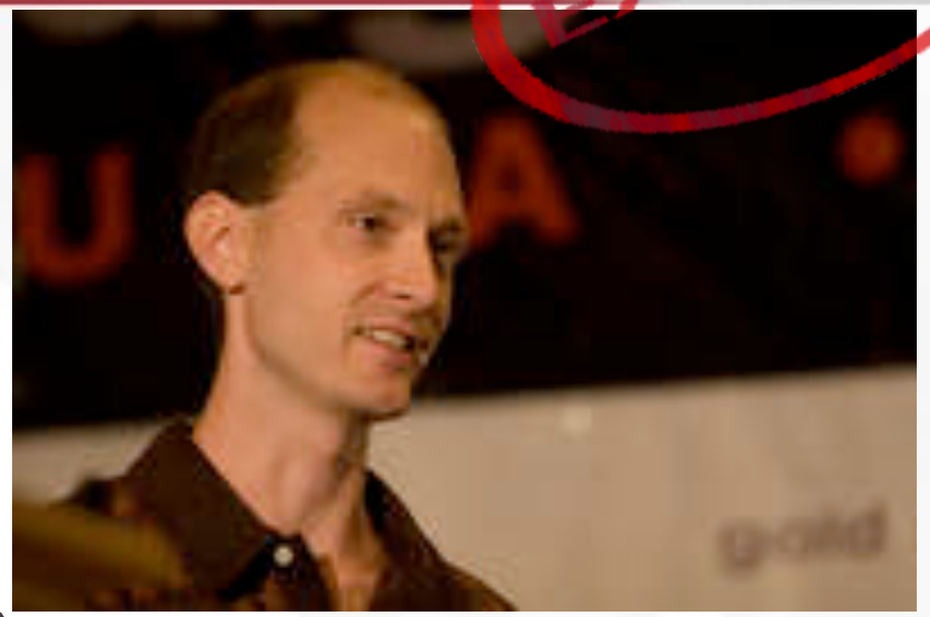
See also: Unit 3. Memory Layout and Allocation

Lecture 3 C Programming Model

A Hacker's Viewpoint:

Charlie Miller

hacked Apple Safari 4 times!



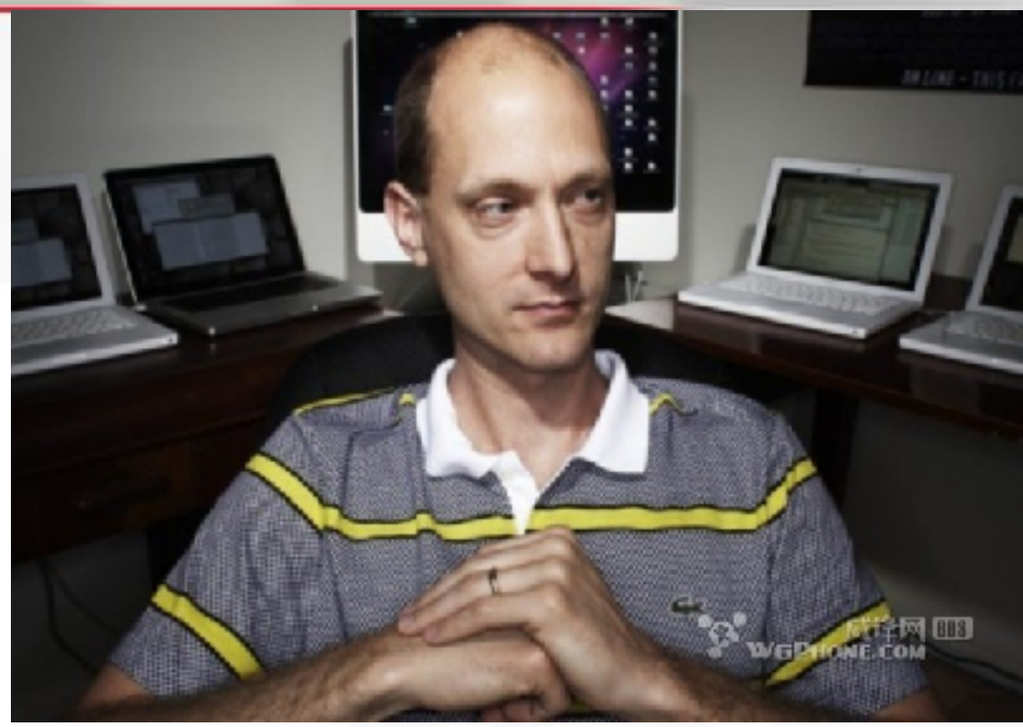
Data Execution Protection (DEP)
Address Space Layout Randomization (ASLR)
Non-eXecutable memory (NX)

Charlie: **The NX bit is very powerful. When used properly, it ensures that user-supplied code cannot be executed in the process during exploitation. Researchers (and hackers) have struggled with ways around this protection. ASLR is also very tough to defeat. This is the way the process randomizes the location of code in a process. Between these two hurdles, no one knows how to execute arbitrary code in Firefox or IE 8 in Vista right now. In the exploit I won Pwn2Own with, I knew right where my shellcode was located and I knew it would execute on the heap for me.**

Lecture 3

C Programming Model

Charlie Miller:

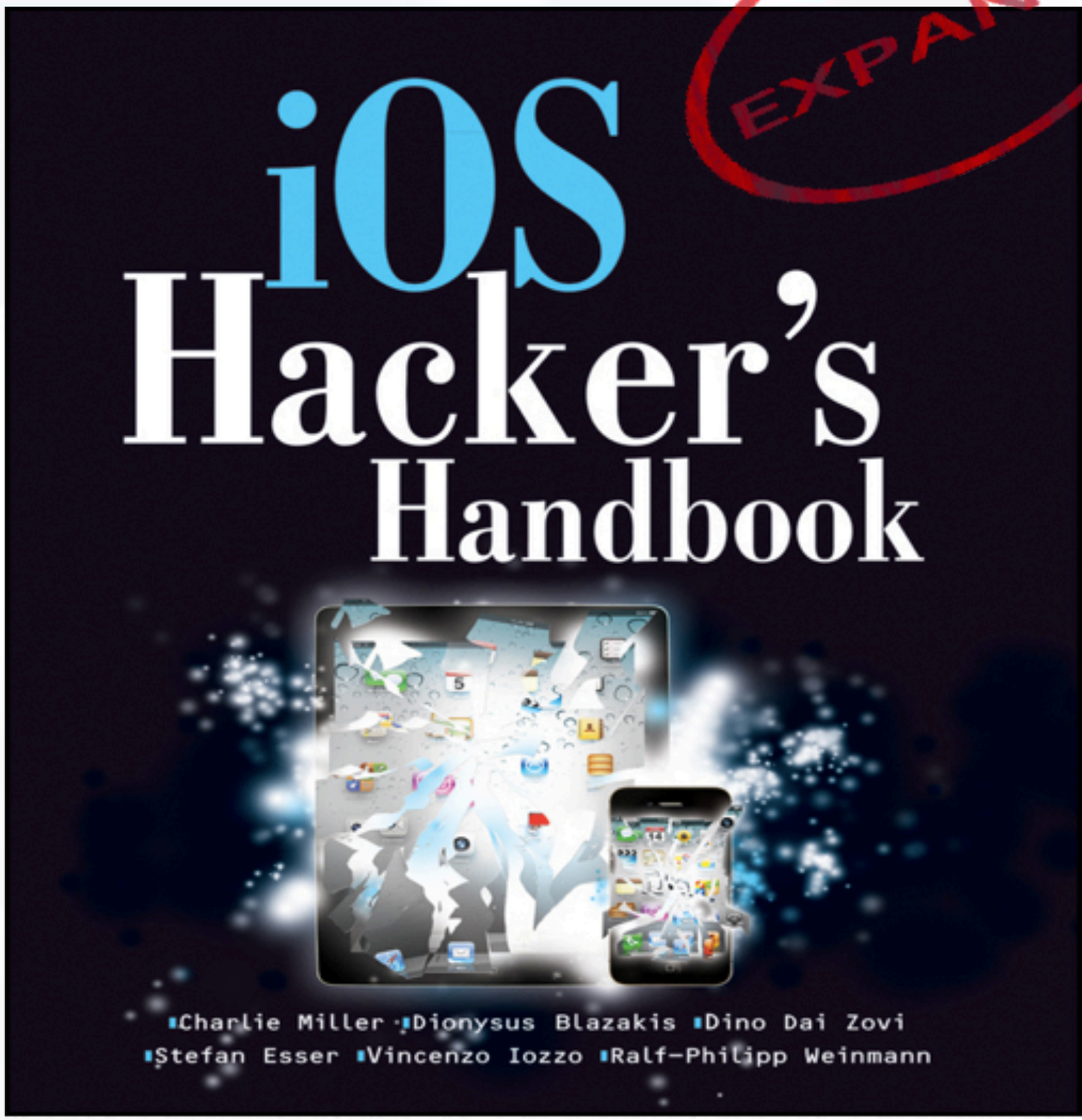


In the exploit I won Pwn2Own with, I knew right where my shellcode was located and I knew it would execute on the heap for me.

EXPAND

Lecture 3 C Programming Model

Deep inside:



icarnegie
curriculum powered by Carnegie Mellon

武漢大学 国际软件学院

Lecture 3 C Programming Model

EXPAND

Lecture 3

C Programming Model

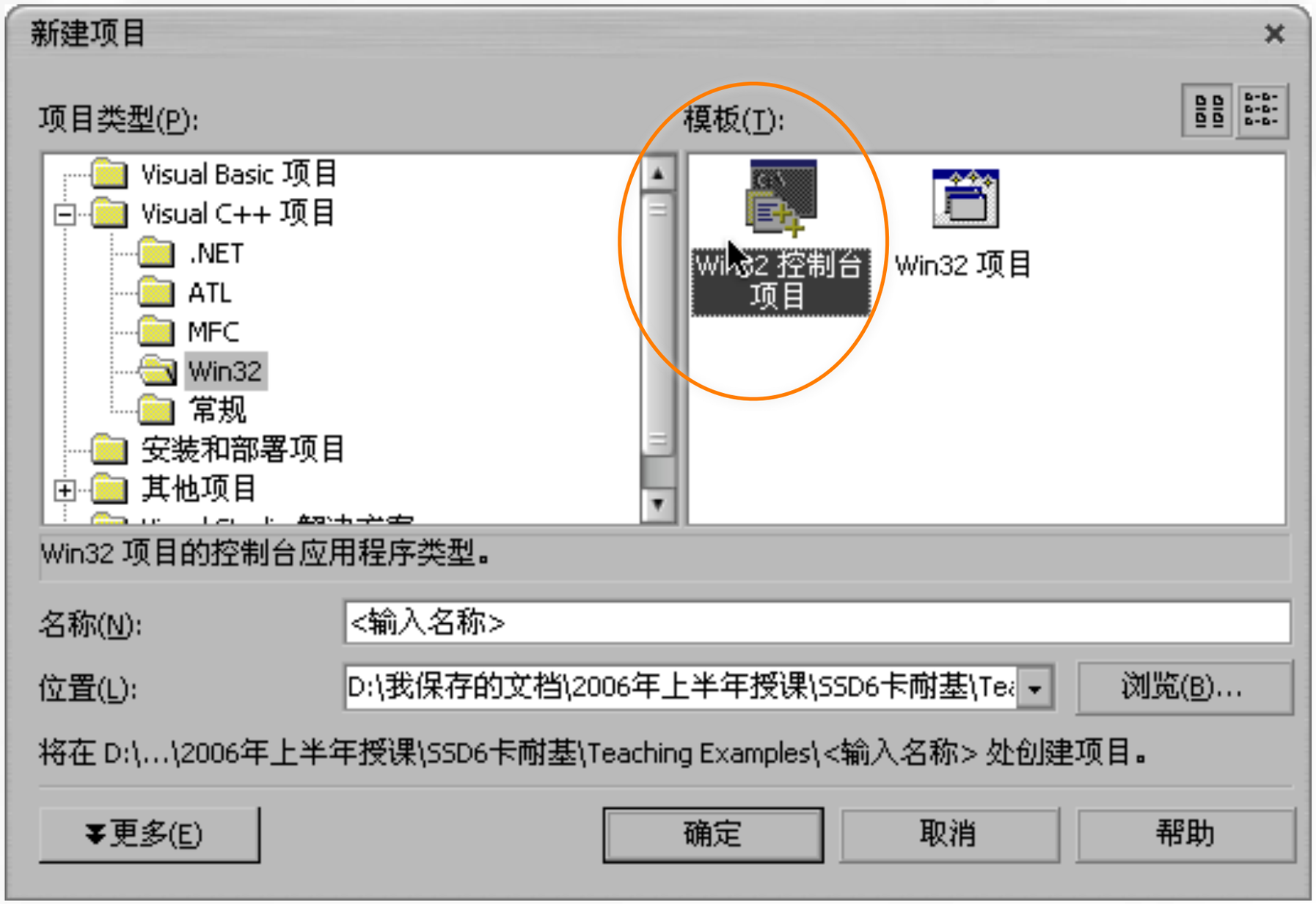
The Compiling & Debugging Environment

- Creating a C Program Using Visual C++
- or, other tools, e.g. sublime text
- Get your compiling & debugging tools:
- gcc\gdb\Xcode\Clion\etc.
- Breakpoints and Steps
- Examining Data
- Example

Lecture 3

C Programming Model

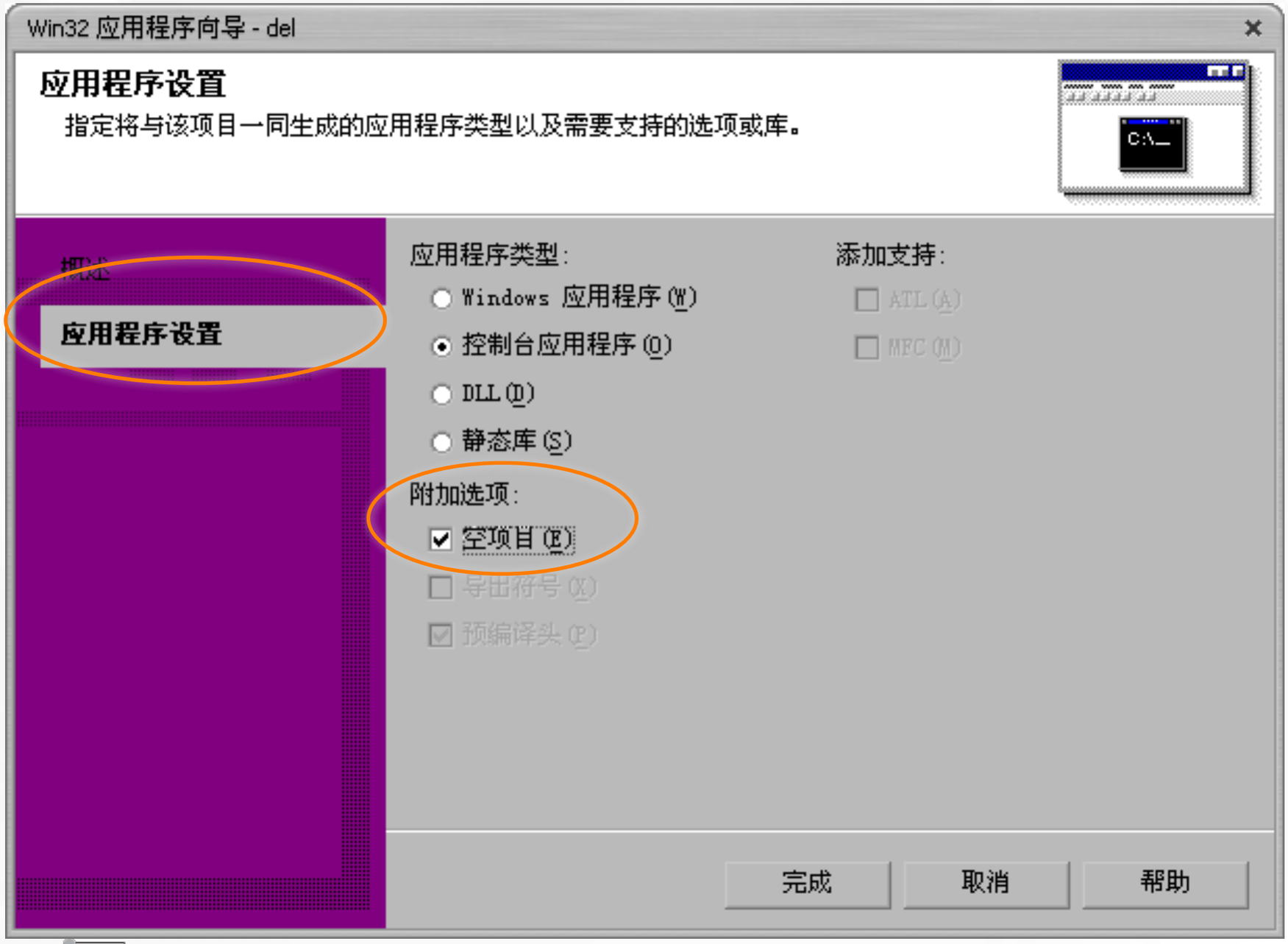
The Visual .NET C++ Environment



Lecture 3

C Programming Model

The Visual .NET C++ Environment



Lecture 3

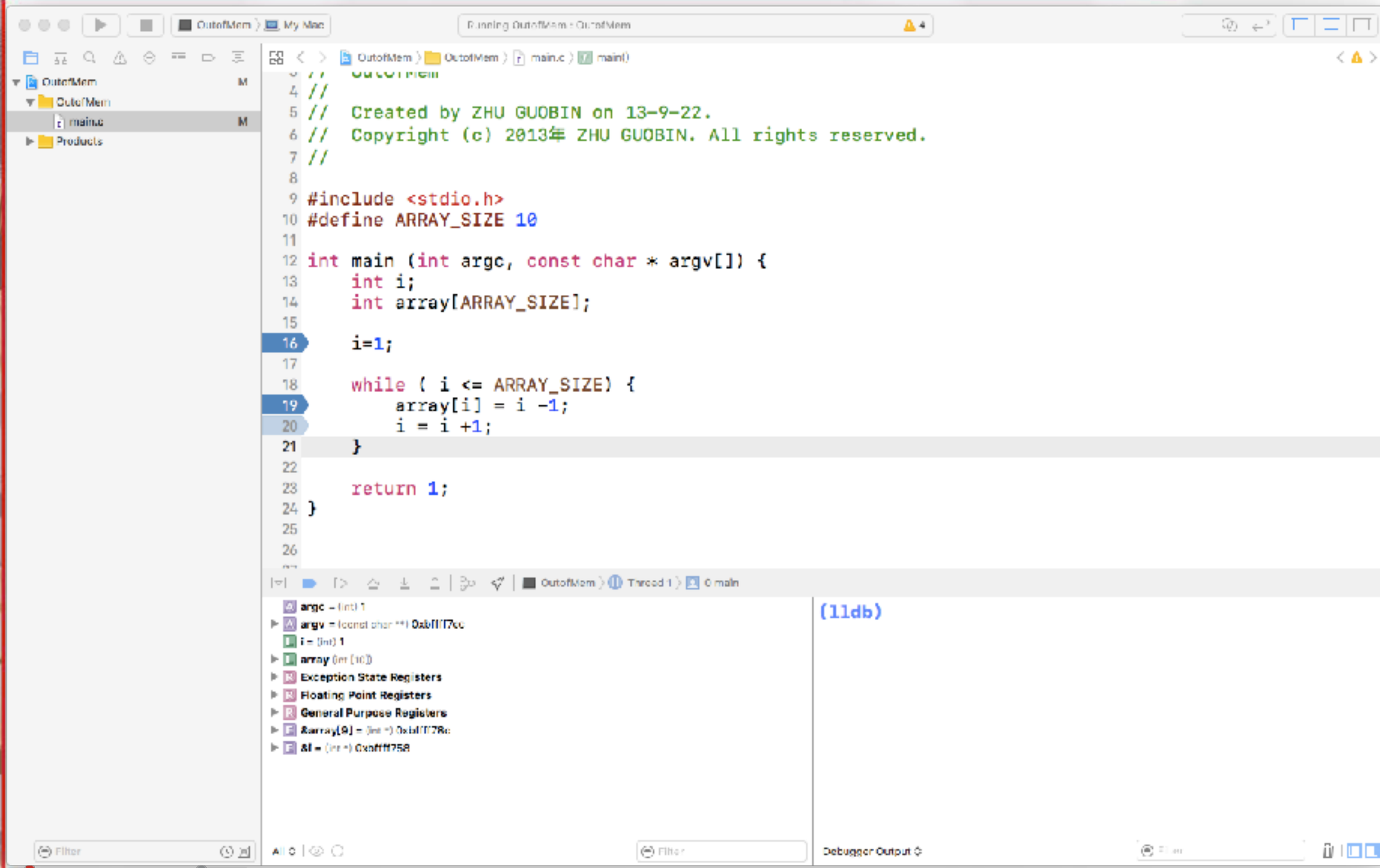
C Programming Model

OutofMem — -bash — 80×28

```
[ZHUtekiMacBook-Air:OutofMem GZhu$ gcc --version                              ]
Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr --with-
gxx-include-dir=/usr/include/c++/4.2.1
Apple LLVM version 8.0.0 (clang-800.0.38)
Target: x86_64-apple-darwin16.7.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault
.xctoolchain/usr/bin
[ZHUtekiMacBook-Air:OutofMem GZhu$ gdb --version                              ]
GNU gdb (GDB) 8.0.1
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin16.7.0".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
ZHUtekiMacBook-Air:OutofMem GZhu$
```

Lecture 3

C Programming Model



Lecture 3

C Programming Model

Camera.cpp - neo - [~/prj/dhewm3-master/neo]

Project

neo (~/prj/dhewm3-master/neo)

- cm
  - CollisionModel.h
  - CollisionModel_contacts.cpp
  - CollisionModel_contents.cpp
  - CollisionModel_debug.cpp
  - CollisionModel_files.cpp
  - CollisionModel_load.cpp
  - CollisionModel_local.h
  - CollisionModel_rotate.cpp
  - CollisionModel_trace.cpp
  - CollisionModel_translate.cpp
- d3xp (header search path)
- framework
- game (header search path)
- idlib
- MayaImport
- renderer
  - Cinematic.cpp
  - Cinematic.h
  - draw_arb2.cpp
  - draw_common.cpp
  - GuiModel.cpp
  - GuiModel.h

Camera.cpp × | Physics.cpp ×

```
/*
==================
idCameraAnim::GetViewParms
==================
*/
void idCameraAnim::GetViewParms( renderView_t *view ) {
    int             realFrame;
    int             frame;
    int             frameTime;
    float           lerp;
    float           invlerp;
    cameraFrame_t   *camFrame;
    int             i;
    int             cut;
    idQuat          q1, q2, q3;

    assert( view );
    if ( !view ) {
        return;
    }

    if ( camera.Num() == 0 ) {
        // we most likely are in the middle of a restore
        // FIXME: it would be better to fix it so this doesn't get
        return;
    }

#ifdef _D3XP
    SetTimeState ts( timeGroup );
#endif
```

545:26 LF UTF-8 Git: master Context: d3

Lecture 3

C Programming Model

Variables and Addresses

- Addresses and Naming
- Pointers: Stored Addresses (Read)
- Examining Memory (Read)
- Arrays and Strings (Read)
- Naughty Pointers

Lecture 3

C Programming Model

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

#define MAXLINE_LENGTH 80

char Buffer[MAXLINE_LENGTH];

char * readString(void)
{
    int nextInChar;
    int nextLocation;

    printf("Input> ");
    nextLocation = 0;
    while ((nextInChar = getchar()) != '\n' &&
       nextInChar != EOF) {
        Buffer[nextLocation++] = nextInChar;
    }
    return Buffer;
}


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

  do {
      newString = readString();
      printf("%s\n", newString);
  } while (strncmp(newString, "exit", 4));
  return 0;
}
```

Try this code when lab time!

Lecture 3

C Programming Model

Addresses and Naming

- In hardware, all data is stored in memory
- Memory is a sequence of bytes that is numbered, starting with 0
- The machine code that the CPU executes operates on memory locations, identified only by their *Addresses*
- Compilers take care of *translating* the operations our programs perform on variables into operations performed on addresses
- Neither the name nor the type of the variables survives this translation

Lecture 3

C Programming Model

Addresses and Naming

- For the most part, programmers don't care about the addresses dealing with their variables
- For example, Java ensure that the programmer lives by its rules
- But C gives the programmer the *freedom* to do just about anything
- With freedom comes responsibility
- C lets programmers "*shoot themselves in the foot*" by writing code that subverts the clean abstractions programming languages usually provide

Lecture 3

C Programming Model

Addresses and Naming

- *&* and *** unary operators
- The & operator returns the address in which a variable or expression is stored
- The * operator does the reverse: it returns the value stored in the address given to it
- This leads tricky: a C programmer can make two variable names refer to the same location in memory ←→ *the hardware cannot distinguish*
- And the above will not be apparent in the source code

Lecture 3

C Programming Model

Variables and Addresses

- Addresses and Naming

```
#include <stdio.h>
char globalchar1;
char globalchar2 = 'g';
int globalint1;
int globalint2 = 9;
char globalchar3;

int main (int argc, char * argv[])
{
  char localchar1;
  char localchar2 = 'l';;
  int localint1;
  int localint2 = 1;
  char localchar3;

.....
}
```

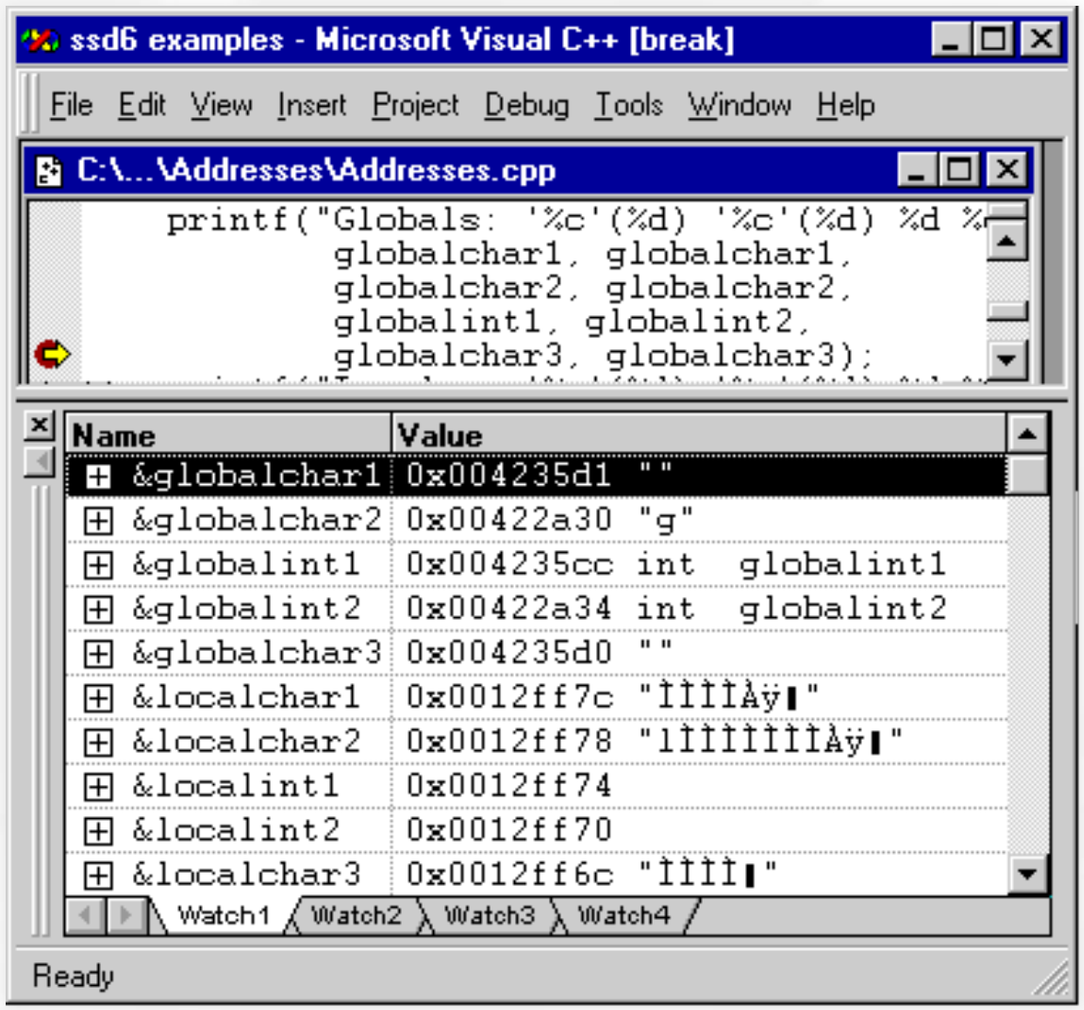
Click to: show the path

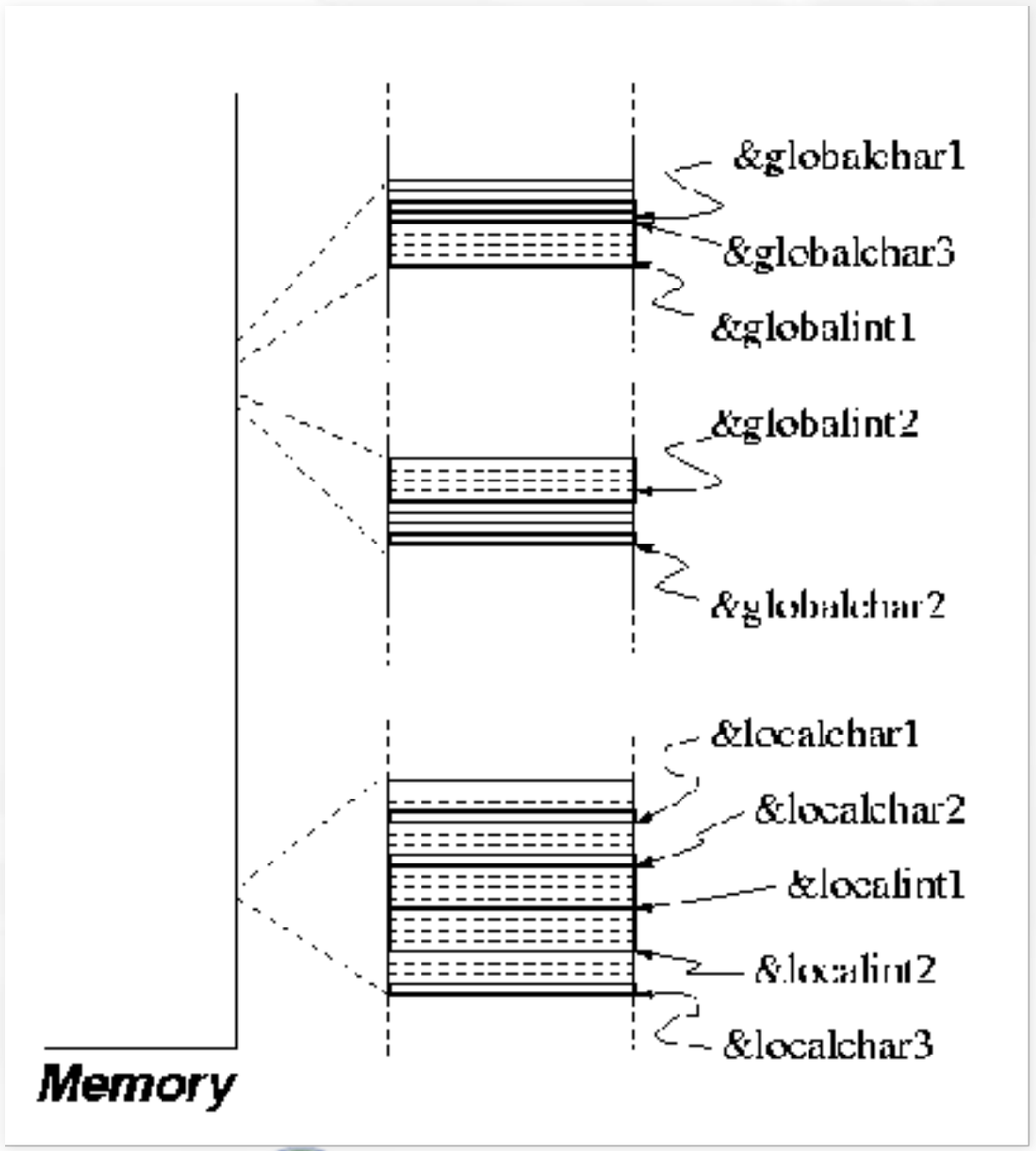
Lecture 3

C Programming Model

Variables and Addresses

- Addresses and Naming





Lecture 3

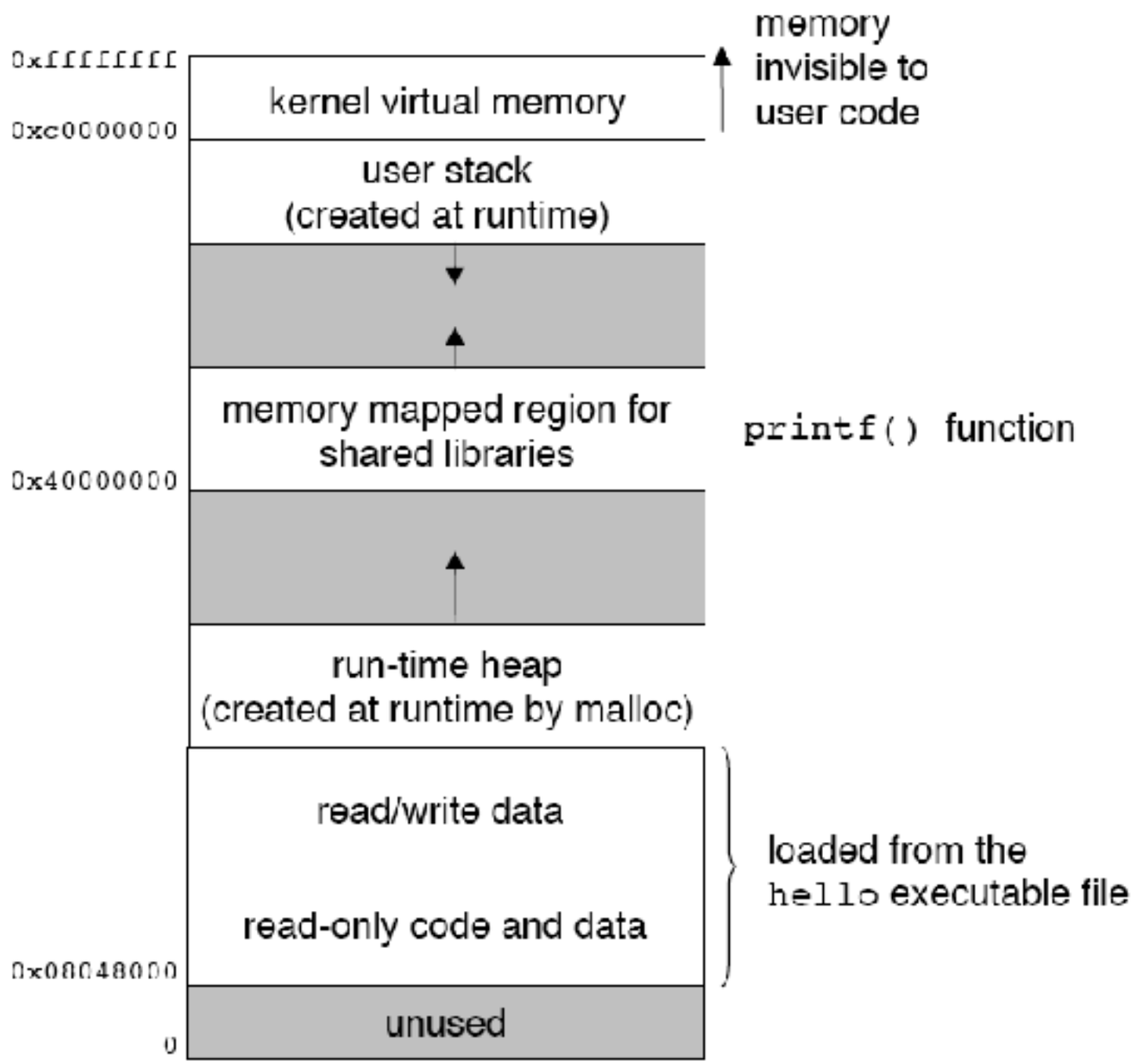
C Programming Model

Variables and Addresses

Program code and data. Code begins at the same fixed address, followed by data locations that correspond to global C variables.

Heap. The code and data areas are followed immediately by the run-time *heap*.

Shared libraries. Near the middle of the address space is an area that holds the code and data for *shared libraries*



Lecture 3

C Programming Model

Addresses

- Typical addresses can be as high as 4 billion or, in newer computers, as high as 16 quintillion (2 elevated to the 64 power.)
- Represent of *addresses:*
  - *Base 2*
  - *Base 10*
  - *Base 16 ←→ 0x……*
- Addresses are clustered in three groups:
  - all the local variables are clustered together
  - of the global variables, those that are initialized when declared are in one cluster, and that those that are not initialized are in another

Lecture 3

C Programming Model

Addresses

- Uninitialized global variables seem to be somehow initialized to *ZERO*
- Local variables start up with arbitrary values
- Padding

Lecture 3

C Programming Model

Pointers or References

- Integers / Pointers
- C allows us to declare variables that hold the addresses of data, rather than holding data directly

Lecture 3

C Programming Model

Variables and Addresses

- Naughty Pointers

```
#include <stdio.h>
main (int argc, char * argv[])
{
    char a = 'a';
    char * ptr = &a;

    while (1);
}
```

Click to: show the path

Lecture 3

C Programming Model

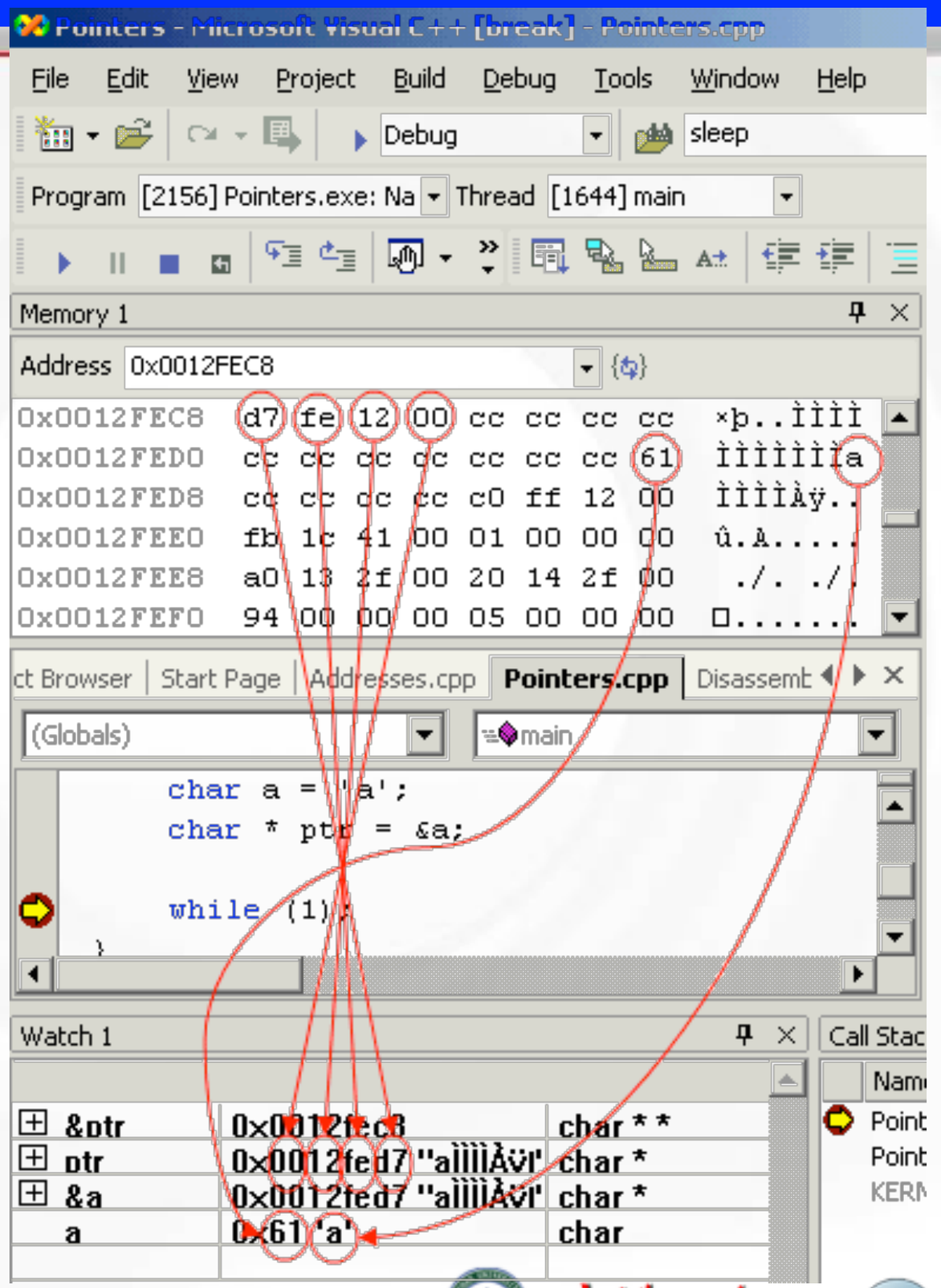
Pointers or References

- Pointers, which are themselves locations, need to reside at some location ←→ &ptr
- C allows us to declare variables that hold the addresses of data, rather than holding data directly
- In a real machine, outside the programming language abstraction, data of any type is represented with nothing but bits
- It is the compiler that interprets those bits as integers or as characters

Lecture 3

C Programming Model

How to represent an address in Mem Tab



Lecture 3

C Programming Model

Arrays and Pointers

- Array Subscripting / Pointer Accessing

subscript:	[0]	[1]	[2]	[3]	[4]	[5]	. . . [19]
cell reference:	A[0]	A[1]	A[2]	A[3]	A[4]	A[5}	. . . A[19]
memory loc:	M	M+1	M+2	M+3	M+4	M+5	. . . M+19

A[i]	*(A + i)
&A[i]	A+ i
A[i + j]	*(A + i + j)
&A[i + j]	A+ i + j

Arrays and Strings ---- How to Initialize Arrays

```
float banana [5] = { 0.0, 1.0, 2.72, 3.14, 25.625 };
```

```
int cantaloupe[2][5] = {
        {10, 12, 3, 4, -5},
        {31, 22, 6, 0, -5},
    };
```

```
int rhubarb[ ][3] ={ {0,0,0}, {1,1,1}, };
```

Arrays and Strings ---- How to Initialize Arrays

```
char vegetables[ ][9] = { "beet",
                          "barley",
                          "basil",
                          "broccoli",
                          "beans" };

char *vegetables[ ] = { "carrot",
                        "celery",
                        "corn",
                        "cilantro",
                        "crispy fried potatoes" };
```

Arrays and Strings ---- How to Initialize Arrays

```
int *weights[ ] = {
                    { 1, 2, 3, 4, 5 },
                    { 6, 7 },
                    { 8, 9, 10 }
                  };
```

```
int row_1[ ] = {1,2,3,4,5,-1}; /* -1 is end-of-row marker */
int row_2[ ] = {6,7,-1};
int row_3[ ] = {8,9,10,-1};
int *weights[ ] = {
                    row_1,
                    row_2,
                    row_3
                  };
```

C Programming Model

Advanced about Point & Array

✗

```
File 1:
    int guava;

File 2:
    extern float guava;
```

```
File 1:
    int guava;

File 2:
    extern int guava;
```

✓

✗

```
File 1:
    int mango[100];

File 2:
    extern int *mango;
```

```
File 1:
    int *mango;

File 2:
    extern int mango[];
```

✓

Why?

Advanced about Point & Array

Why? A page-break fault

The C Programming Language, 2nd Ed, K & R, foot of page 99:

As formal parameters in a function definition

turn leaf to page 100:

```
char s[ ];
```

and

```
char *s;
```

are equivalent; . . .

Lecture 3

C Programming Model

How Arrays and Pointers Are Accessed

L-value → **X = Y** ← R-value

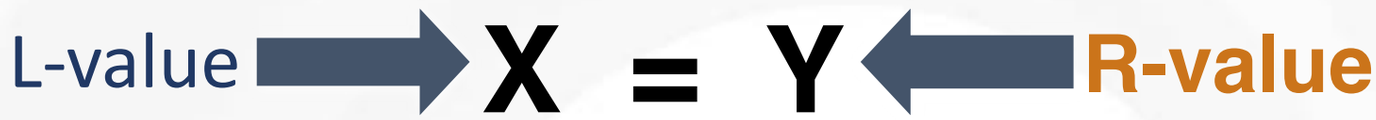
The symbol x, in this context, means the *address* that x represents.

The symbol y, in this context, means the *contents of the address* that y represents.

A "modifiable L-value" is a term introduced by C. Hence, an arrayname is an L-value but not a modifiable L-value. The standard stipulates that an assignment operator MUST have a modifiable L-value as its left operand.

C Programming Model

How Arrays and Pointers Are Accessed



The key point here is that the address of each symbol is known at compile-time. So if the compiler needs to do something with an address (add an offset to it, perhaps), it can do that directly and does not need to plant code to retrieve the address first. In contrast, the current value of a pointer must be retrieved at runtime before it can be dereferenced.

C Programming Model

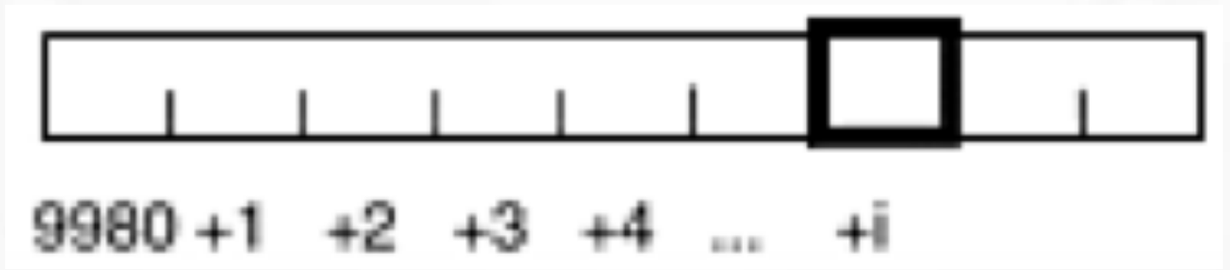
Case A → A Subscripted Array Reference

char a[9] = “abcdefgh”; … … c=a[i];

Compiler symbol table has a as address 0x9980

Runtime step 1: get value *i*, and add it to 9980;

Runtime step 2: get the contents from address 0x(9980+*i*)



C Programming Model

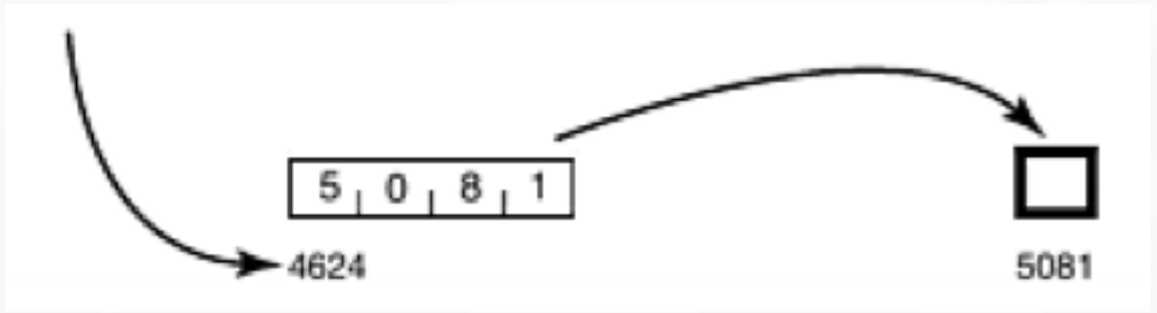
Case B → A Pointer Reference

char *p; … … c = *p;

Compiler symbol table has p as address 0x4624

Runtime step 1: get the contents from address 0x4624, say '5081';

Runtime step 2: get the contents from address 0x5081



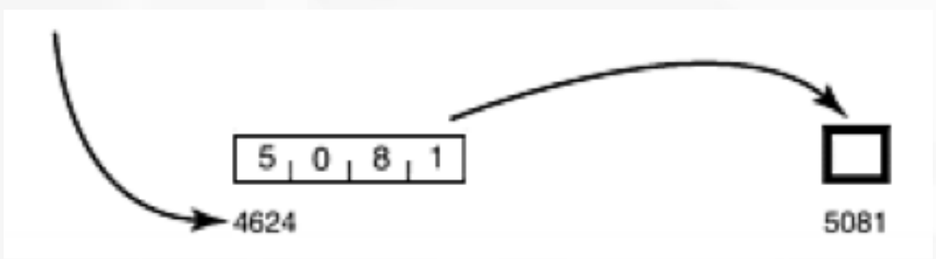
C Programming Model

Here comes answer!

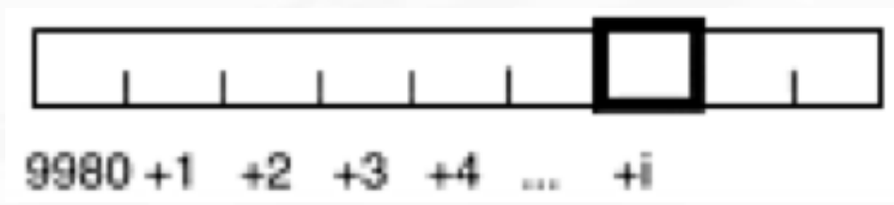
```
File 1:
    int *mango;

File 2:
    extern int mango[];
```

✓

File 1:
int *mango;



File 2:
extern int mango[];



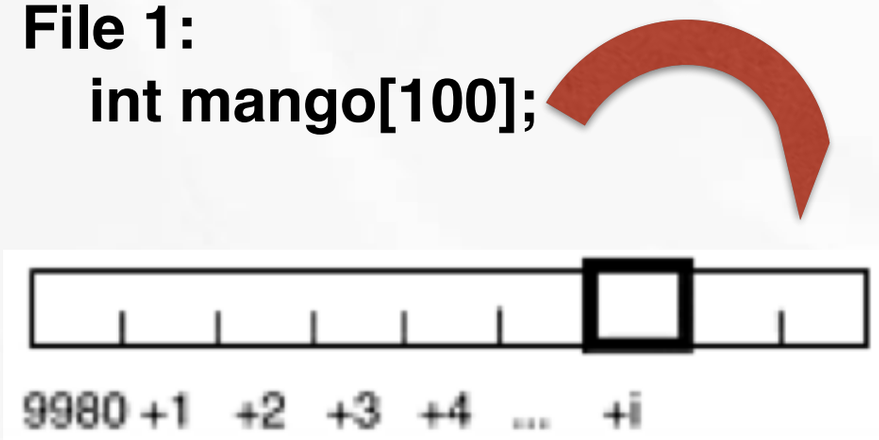
C Programming Model

Here comes answer!

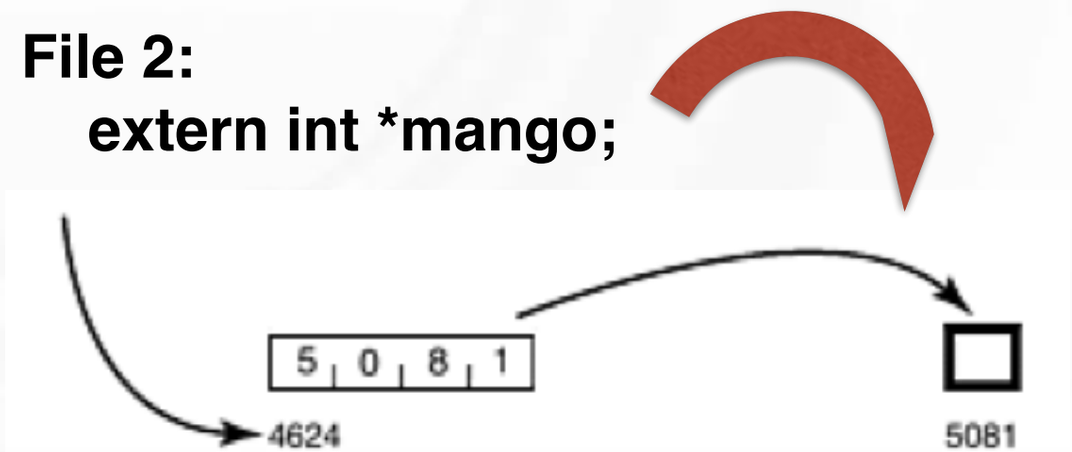
```
File 1:
    int mango[100];

File 2:
    extern int *mango;
```

✗

File 1:
int mango[100];



File 2:
extern int *mango;



Lecture 3 C Programming Model

we move further…


```
// in file 1:
   long n;

// while in file 2:
   extern int n;
```

what will happen ?

C Programming Model

FOUR possibilities:

- if compiler smarter enough, type conflict could be detected
- if your core assigns *long* with the same length with *int*, i.e., 32 bit-system, occasionally, it works
- even if your *long* occupies various space with the *int*, while they share the same memory space, say assigning lower bits of *long* to the *int*, occasionally, it works
- if above, *long* instance doesn't equal to the *int* instance, then, an error arises.

```
// in file 1:
    long n;

// while in file 2:
    extern int n;
```

Lecture 3 C Programming Model

find a solution:

```
File 1:
   int mango[100];

File 2:
   extern int *mango;
```

✗

```
File 1:
   int mango[100];

File 2:
   extern int mango[];
```

```
File 1:
   int * mango;

File 2:
   extern int *mango;
```

C Programming Model

especially when in function calls:

```
main (int argc, char * argv[])
{
    // something else
}
```

✓

```
main (int argc, char ** argv)
{
    // something else
}
```

✓

C Programming Model

an un-declared function return type

EXPAND

```
main ()
{
    double s;
    s = sqrt (2);
    printf ("%g\n", s);
}
```

(a)

```
extern int sqrt();
main ()
{
    double s;
    s = sqrt (2);
    printf ("%g\n", s);
}
```

(b)

in (a), sqrt() was not declared. If that, compiler has to decide the return type by context. In this example, by C defaulted, an undeclared identifier followed by a "(" symbol, will return a *int* type value, just like in (b) declaration.

Lecture 3 C Programming Model

Arrays and Strings

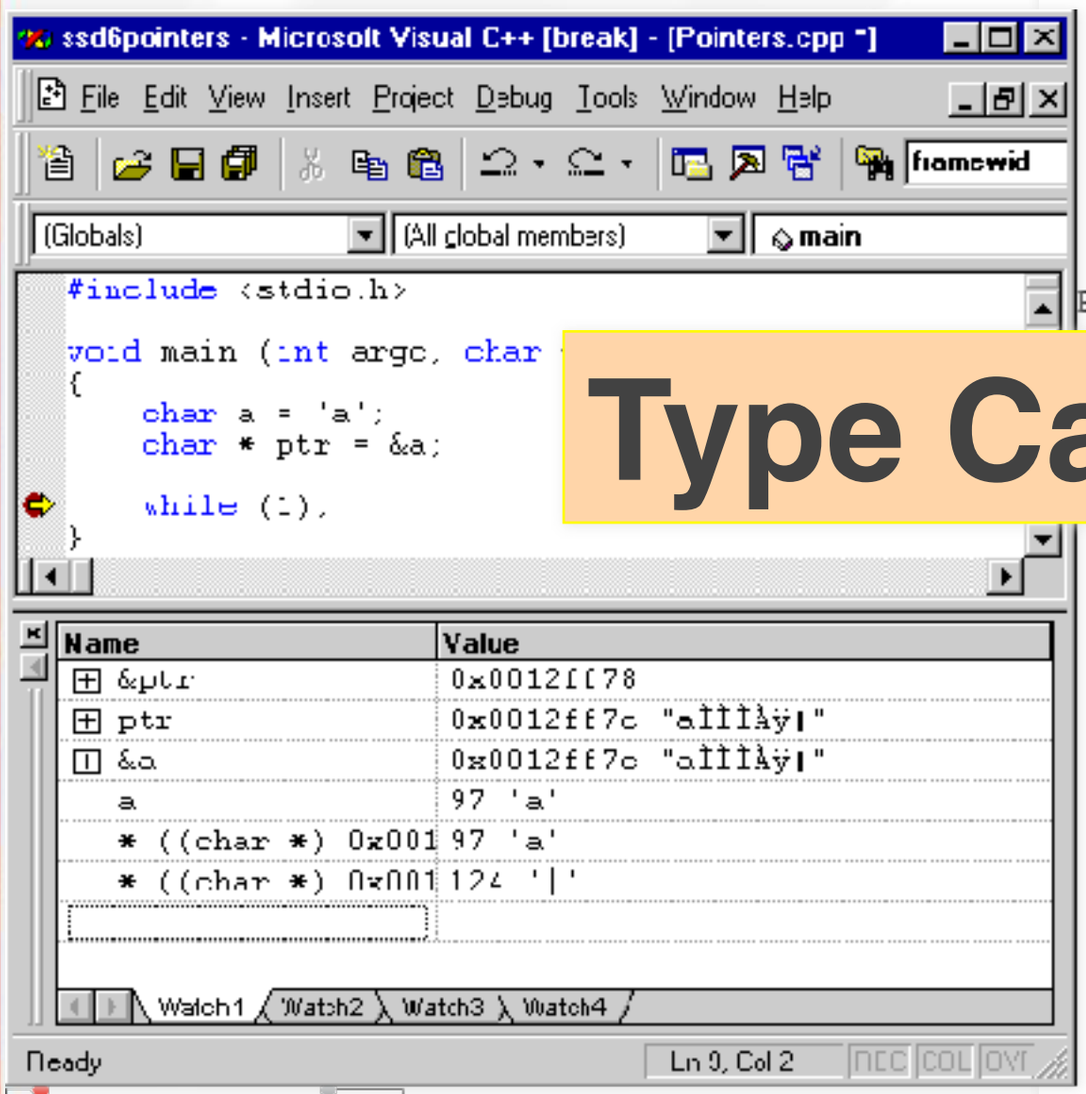
- Allocation and
- Reference
- *see example 1.3.4*

```
1  #include <stdio.h>
2
3  void Initialize (char * a, char * b)
4  {
5          a[0] = 'T'; a[1] = 'h'; a[2] = 'i';
6          a[3] = 's'; a[4] = ' '; a[5] = 'i';
7          a[6] = 's'; a[7] = ' '; a[8] = 'A';
8          a[9] = '\0';
9          b = a;
10         b[8] = 'B';
11 }
12
13 #define ARRAY_SIZE 10
14 char a[ARRAY_SIZE];
15 char b[ARRAY_SIZE];
16
17 int main(int argc, char * argv[])
18 {
19         Initialize(a, b);
20         printf("%s\n%s\n", a, b);
21         return 0;
22 }
23
```

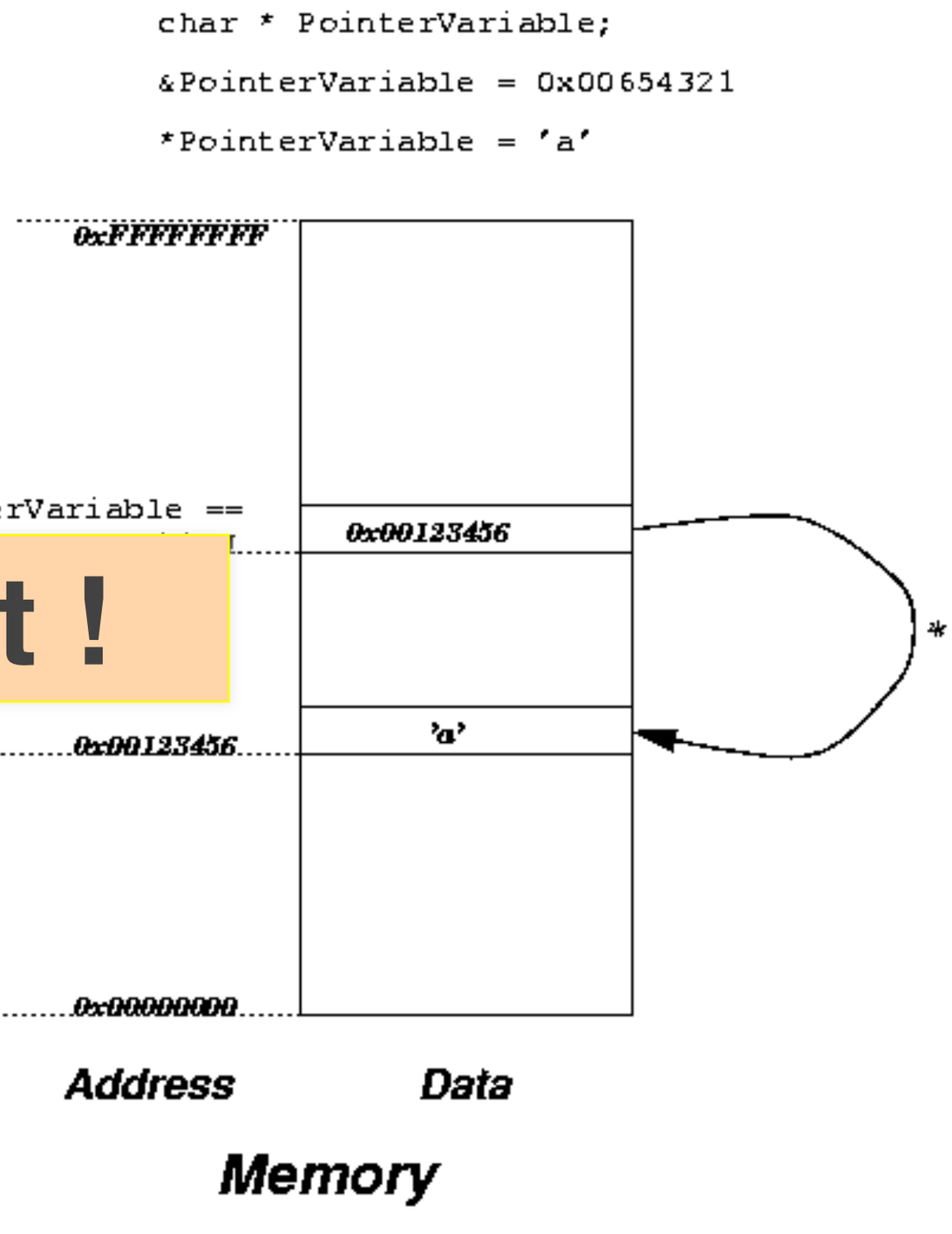
Lecture 3 C Programming Model

Conclusions:

- Naughty Pointers



Type Cast !



Lecture 3

C Programming Model

Conclusions

- Naughty Pointers

As far as the hardware is concerned, memory is a featureless sequence of bytes that gets interpreted as single bytes or as groups of bytes, depending on how the compiler is trying to interpret them. The same underlying bytes can be interpreted as one int or four chars. The C compiler usually keeps all this straight for us, interpreting memory locations the right way. But if we circumvent the safeguards of the compiler, as we did by using a cast, the compiler can no longer guarantee the correct interpretation of memory.

Lecture 3

C Programming Model

Data and Function Calls: Local Variables

- **Scopes**
  - **Scoping**
  - **See example in callee_by_value (scope)**
- **Formal Parameters: Value and Reference**
  - **By value**
  - **By reference (callee_by_reference)**
  - **See example in 1.4.1a**
- **Recursion**
  - **Dynamic vs. Statical allocation**
  - **See example in 1.4.1c(recursion)**

Lecture 3

C Programming Model

Scopes:

- Variables have scopes: a variable name may refer to different data items depending on where it is used
- Compilers implement scoping by manipulating different memory addresses from within different scopes: one name refers to different mem allocations
- The hardware does not know about scoping, just as it doesn't know about variable names or types. The hardware merely manipulates memory addresses blindly


Stack

Lecture 3

C Programming Model

Scopes: An example

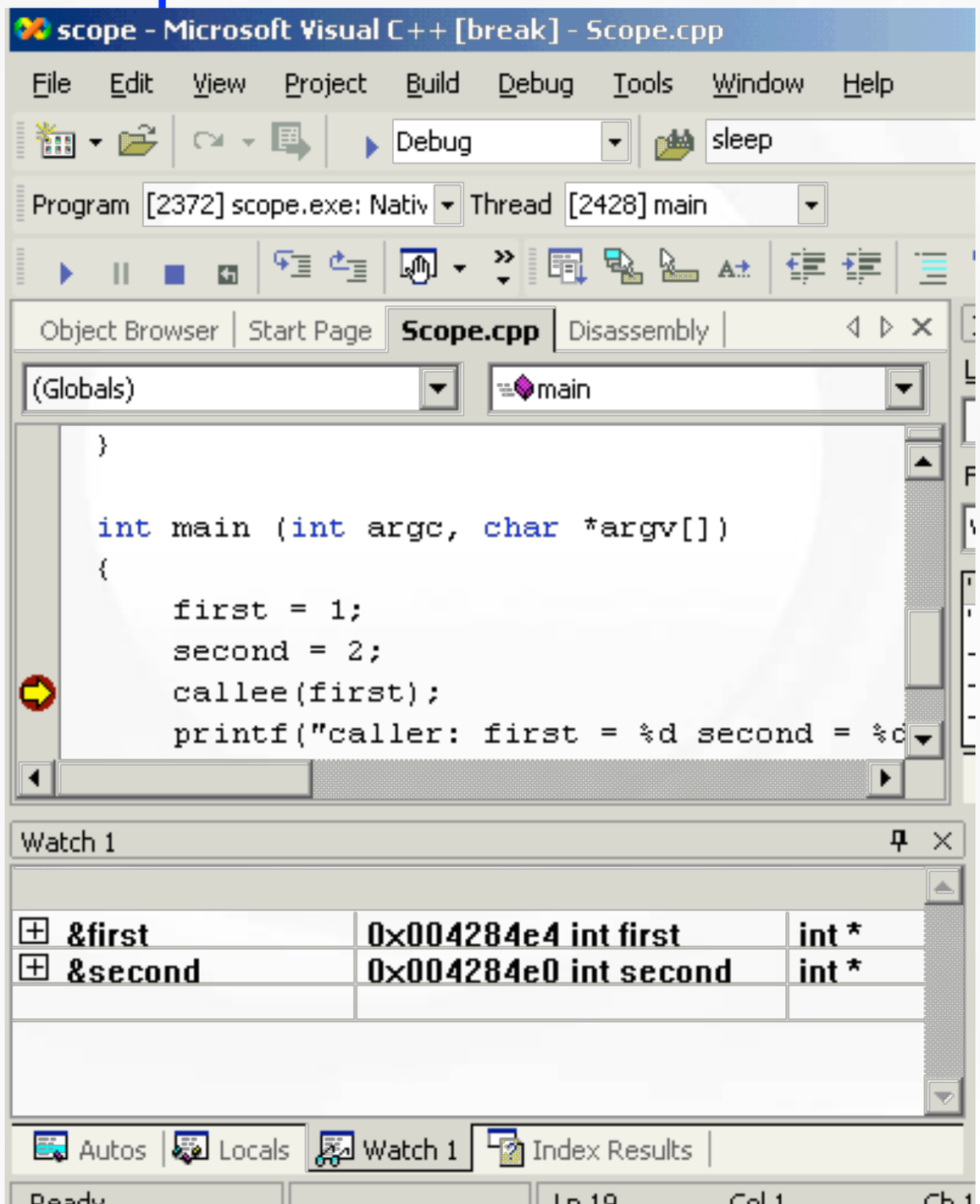
```
#include <stdio.h>

int first;
int second;

void callee ( int first )
{
        int second;

        second = 1;
        first = 2;
        printf("callee: first = %d second = %d\n", first, second);
}

int main (int argc, char *argv[])
{
        first = 1;
        second = 2;
        callee(first);
        printf("caller: first = %d second = %d\n", first, second);
        return 0;
}
```

Click to: show the path

Lecture 3

C Programming Model

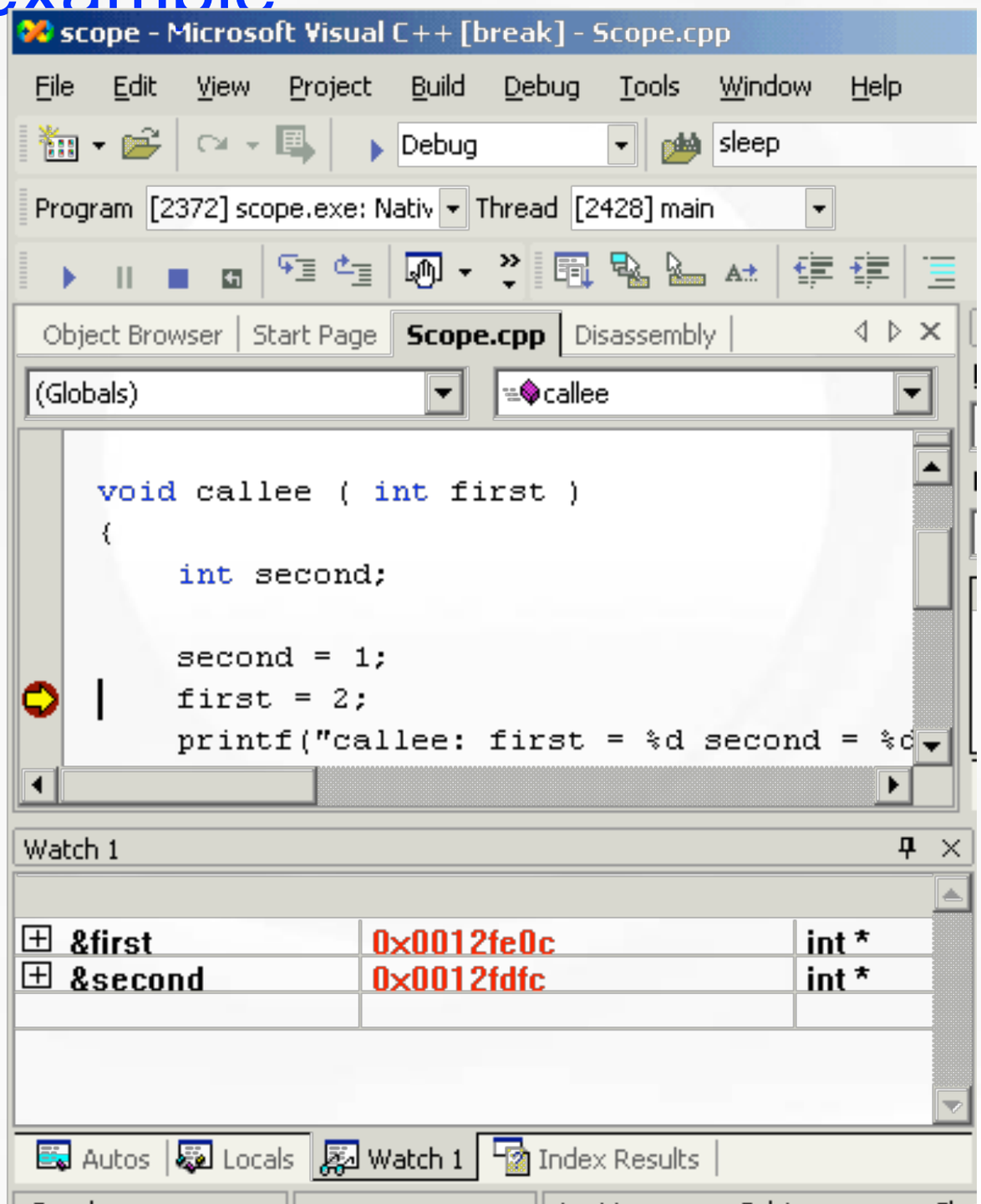
Scopes: An example



Lecture 3

C Programming Model

Scopes: An example



Lecture 3

C Programming Model

Formal Parameters: Value and Reference:

- As seen above, the parameters' address is very near that of the local variables
- That happens because ***first*** is a parameter passed by ***value***
- ***Conclusion***: The scope of a parameter that is passed by value is identical to the scope of a variable that is local to the function (callee)
- This allows the actual parameter to remain unperturbed by any assignment made to the formal parameter in the function (callee)

C Programming Model

Formal Parameters: Reference:

```
#include <stdio.h>

int first;
int second;

void callee ( int * first )
{
        int second;

        second = 1;
        *first = 2;
        printf("callee: first = %d second = %d\n", *first, second);
}

int main (int argc, char *argv[])
{
        first = 1;
        second = 2;
        callee(&first);
        printf("caller: first = %d second = %d\n", first, second);
        return 0;
}
```

Click to: show the path

Lecture 3

C Programming Model

Formal Parameters: Reference:

- **Parameters to functions can also be passed by reference**
- **In this case, the compiler would not allocate a new address for the formal parameter, but would use the address of the parameter specified by the caller**
- **This way, an assignment to the formal parameter within the function *(callee)* would end up changing the value of the variable passed by the caller**

Lecture 3

C Programming Model

Recursion:

- **Question 1:** How does the compiler decide where to allocate variables ?
- **Question 2:** Why are local variables and globals allocated in such different places ?
- Take a look at the following example …

Lecture 3

C Programming Model

Recursion:

Click to: show the path

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

void callee (int n)
{
        if (n == 0) return;
        printf("%d (0x%08x)\n", n, &n);
        callee (n - 1);
        printf("%d (0x%08x)\n", n, &n);
}

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

        if (argc < 2)
        {
                printf("USAGE: %s <integer>\n", argv[0]);
                return 1;
        }

        n = atoi(argv[1]);

        callee(n);
        return 0;
}
```

```
10 (0x0065fda4)
9 (0x0065fd4c)
8 (0x0065fcf4)
7 (0x0065fc9c)
6 (0x0065fc44)
5 (0x0065fbec)
4 (0x0065fb94)
3 (0x0065fb3c)
2 (0x0065fae4)
1 (0x0065fa8c)
1 (0x0065fa8c)
2 (0x0065fae4)
3 (0x0065fb3c)
4 (0x0065fb94)
5 (0x0065fbec)
6 (0x0065fc44)
7 (0x0065fc9c)
8 (0x0065fcf4)
9 (0x0065fd4c)
10 (0x0065fda4)
```

Lecture 3

C Programming Model

Recursion: Tips

- This shows that the compiler allocates one address for each call to callee
- But how did the compiler know that it had to allocate 10 instances of n?
- In dilemma: The compiler would not be able to decide how many to allocate until after the program was already running. But if the program is running, the compiler has already finished its work!

Lecture 3

C Programming Model

Recursion: Tips

- The compiler inserts additional code for every function call and every function return
- This is called *dynamic* allocation, because the local variables are allocated at runtime, as needed
- Global variables can be allocated *statically*: the compiler can fix specific addresses for global variables before the program executes
- The compiler does not know how many variables it will need to allocate dynamically, it reserves a lot of space for expansion.
- This is why the local variables are allocated to addresses that are very far away from the addresses that hold the global variables

Lecture 3

C Programming Model

Data and Function Calls: Activation Records & Stacks

- **Activation Records**
  - **The chunk of memory allocated for each function invocation is called an activation record.**
- **Stacks**
  - **stack frames**
  - **Push & Pop**

Lecture 3

C Programming Model

Activation Records

- As seen above: local variables have to be allocated dynamically. This slows down the execution of the program a little bit
- To minimize the cost of dynamic allocation, the compiler tries to allocate local variables in large groups: ***one single chunk***
- The calculation is done at compile time, when it does not slow down the program's execution, and it results in a single allocation operation at run time

Lecture 3

C Programming Model

Activation Records

- The chunk of memory allocated for each function invocation is called an ***activation record***
- An ***activation record*** for a function invocation is created when the function is called, and it is destroyed when the function returns
- This operation is so common that hardware is evolved: as simple as possible

Lecture 3

C Programming Model

Activation Records: terms

- Activation records are organized in a ***stack***
- They are often also called ***stack frames***
- the activation record for the callee is ***push***ed on top of that of the caller
- Only the activation record at the ***top of the stack*** (or simply TOS) can be accessed
- to access one below, we need to first ***pop*** the activation records that are on top of it
- Making things simple lets hardware be evolved

Lecture 3

C Programming Model

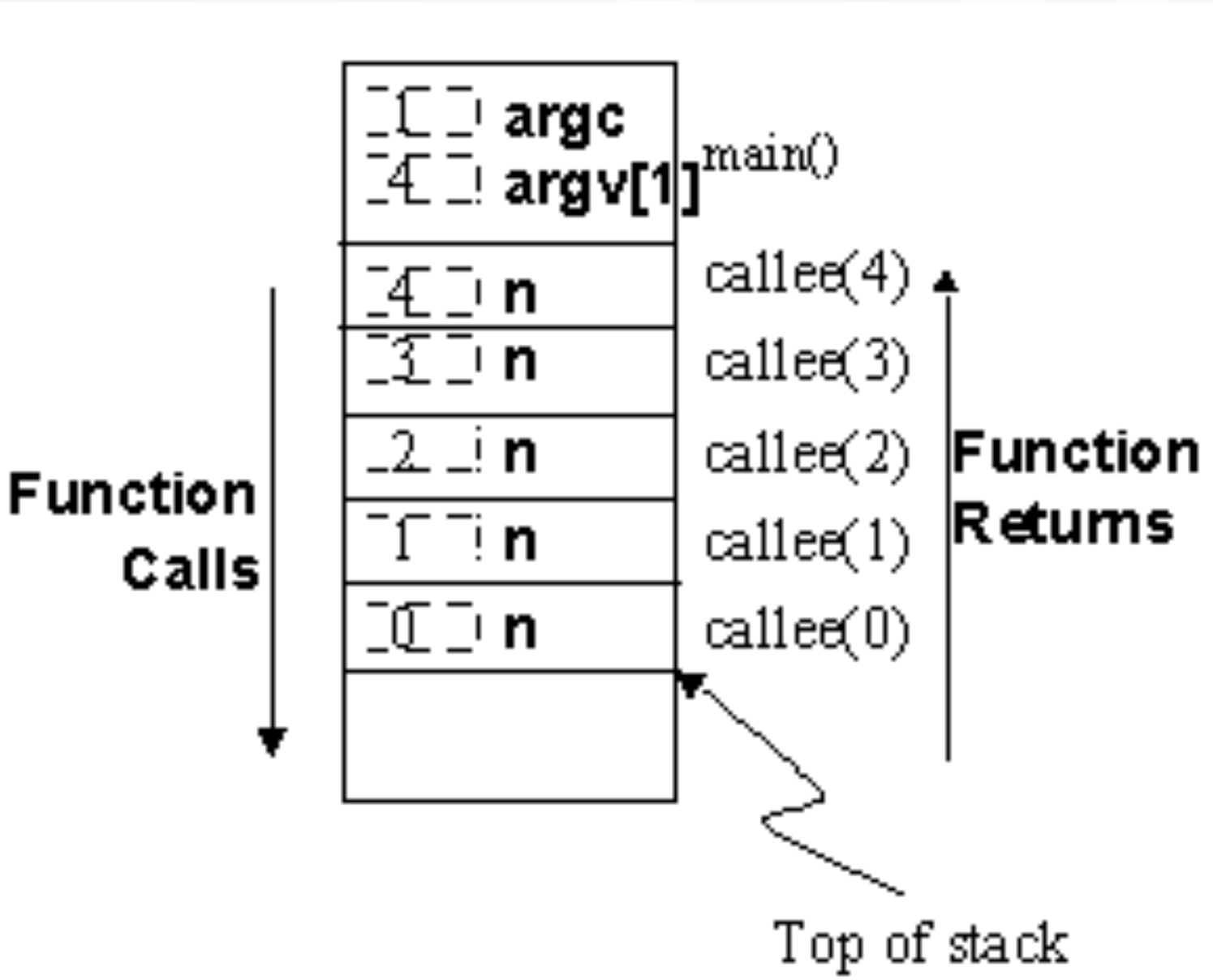
Activation Records: Simple rules

- A function never returns before all of its active subroutines return-that is to say, we never need to delete an activation record that is not at the ***top of the stack***.
- The rules of scoping imply that a function cannot access the local variables of its parent-that is to say, we never need to access data from an ***activation record*** that is not at the ***top of the stack***

Lecture 3

C Programming Model

Activation Records: An example



Lecture 3

C Programming Model

Activation Records: Two important values

- The ***stack pointer*** holds the address where the stack ends-it is here that a new activation record will be allocated.
- The ***frame pointer*** holds the address where the previous activation record ends-it is to this value that the stack pointer will return when the current function returns

Lecture 3

C Programming Model

When a function is called, compiler and hardware:

- Push the frame pointer into the stack.
- Set the frame pointer equal to the stack pointer.
- Decrement the stack pointer by as many memory addresses as are required to store the local state of the callee

Lecture 3

C Programming Model

When a function returns, compiler and hardware:

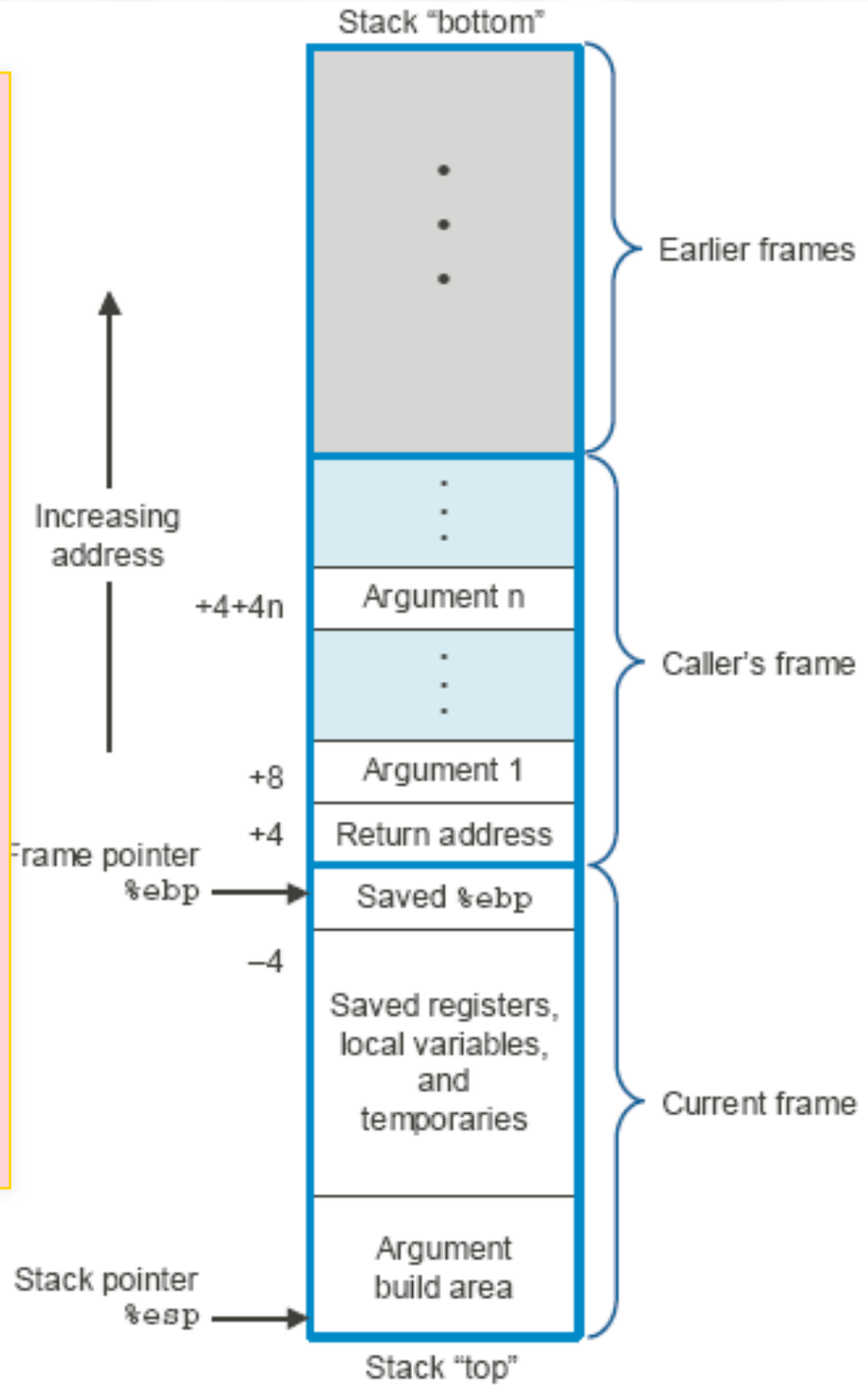
- Set the stack pointer equal to the frame pointer.
- Pop the value of the old frame pointer from the stack

Lecture 3

C Programming Model

Data and Function Calls: Conclusions

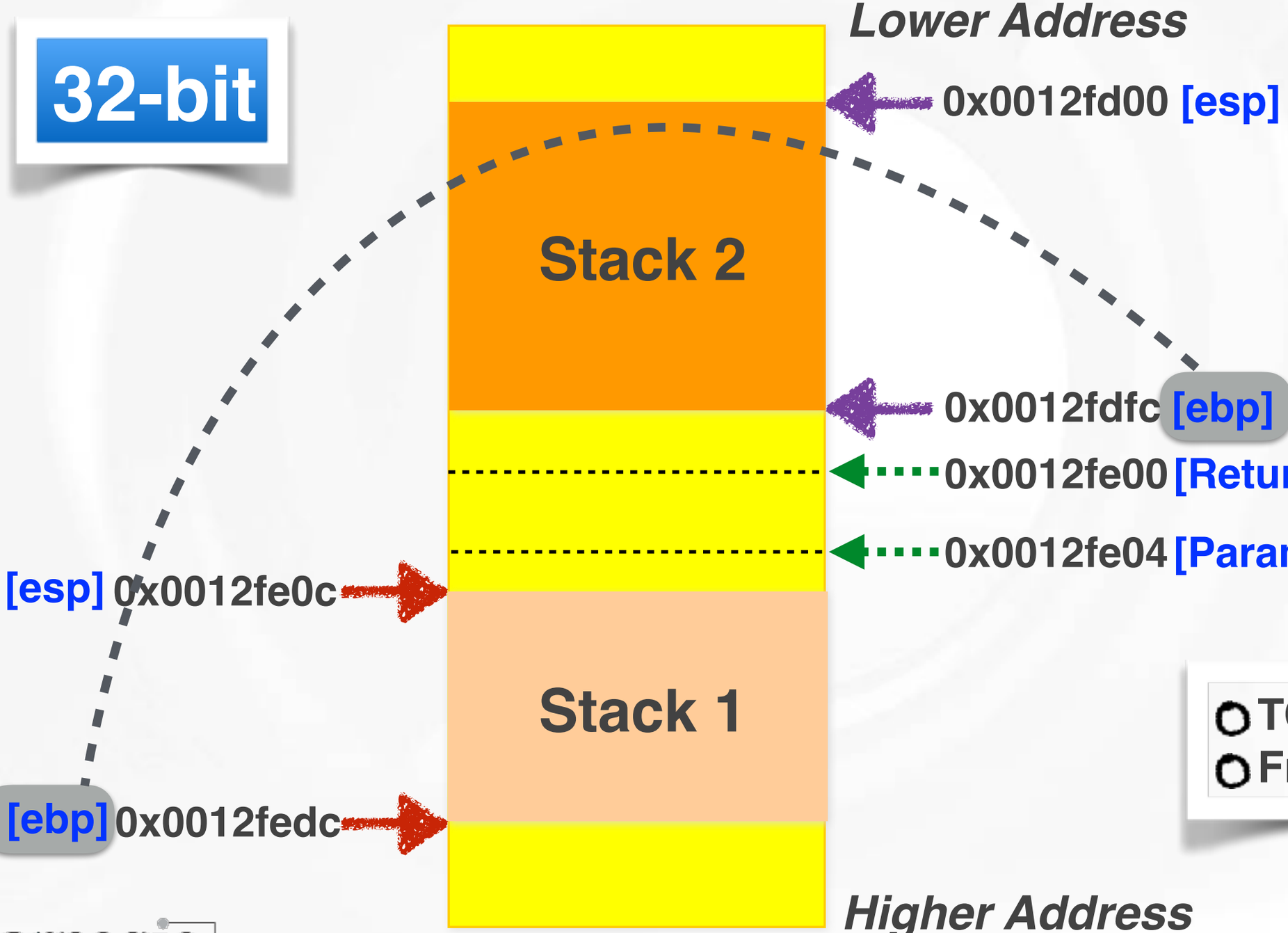
- By so manipulating the stack pointer and the frame pointer, the potentially expensive task of dynamically allocating local variables becomes simple, well organized, and fast
- Not only local variables, but also args, return addresses and transient temporary values are stored in the stack
- All compilers for a particular type of hardware follow the same rules



Lecture 3

C Programming Model

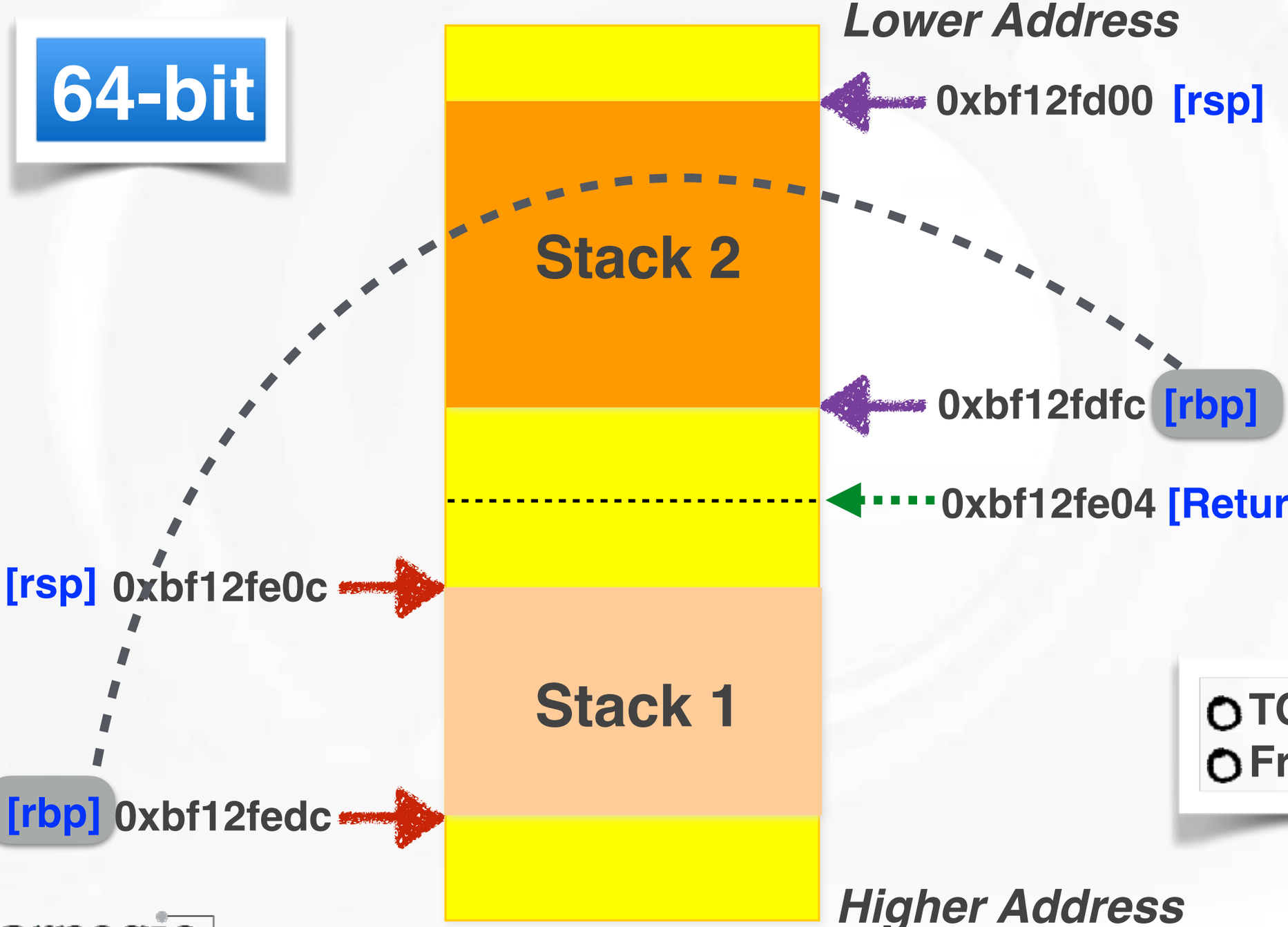
Data and Function Calls: Conclusions {32-bit}



Lecture 3

C Programming Model

Data and Function Calls: Conclusions {64-bit}



Lecture 3

C Programming Model

Code is in Memory, Too! : Code in mem.

- A CPU understands instructions as:
  - "Move the byte at address X into location Y."
  - "Add the contents of these two addresses together and put the result in Z."
  - "Is the byte at address X zero?"
- A CPU doesn't understand instructions as:
  - Calculate the address of the fifth element of this array.
  - c = (a + b) * d
  - while (i < MAX)

Lecture 3

C Programming Model

Code is in Memory, Too! : Code in mem.

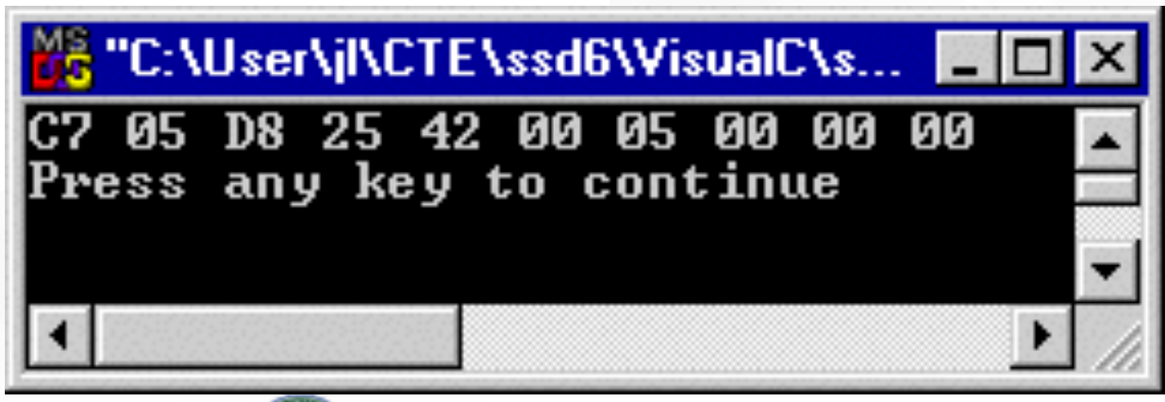
- The CPU design concentrates on doing a few simple things really fast, and leaves the complexity and flexibility to the compiler
- The CPU executes one such simple instruction at a time
- In memory, instructions are indistinguishable from data, because, in memory, everything is represented as raw bits
- Just like an integer could be accidentally read as if it were a character and vice-versa, so can instructions be accidentally read

Lecture 3

C Programming Model

Code is in Memory, Too! : An example

```
#include <stdio.h>
int a;
int main (int argc, char *argv[])
{
        int i; unsigned char * c;
        a = 5;
        c = ((unsigned char *) 0x0040b7D8);
        for (i = 0; i < 10; i++)
        {
                printf("%02X ", *c);
                c++;
        }
        printf("\n");
}
```



Lecture 3

C Programming Model

Code is in Memory, Too! : Code has addr., too!

- **Question**: what determines which data item or which CPU instruction will be fetched at each step of a program's execution?
- **Answer**: There is a special value kept by the CPU, called the program counter (often, simply PC), which contains the address of the instruction that is about to be executed (for example: *stack pointer and frame pointer*)
- a fetch-decode-execute cycle:
  - Fetches memory from the address contained in the program counter.
  - Interprets the "opcode" just fetched and decides what the instruction means.
  - Fetches operands, executes the instruction, and stores the result.

Lecture 3

C Programming Model

Code is in Memory, Too! : Code has addr., too!

```
if (a == 0) {
    a = 5;
}
```

Example 1 if-statement

Address	Instruction
0x40B7D8	fetch a
0x40B7E0	branch-if-not-zero 0x40B7F0
0x40B7E6	mov a, 5
0x40B7F0	....

Lecture 3

C Programming Model

Code is in Memory, Too! : Code has addr., too!

```
if (a == 4) {
     a = 5;
}
else {
     a = 4;
}
```

Example 3 if-else-statement

Address	Instruction
0x40B7D8	temp = a - 4
0x40B7E0	branch-if-not-zero 0x40B7F6
0x40B7E6	mov a, 5
0x40B7F0	jump 0x40B800
0x40B7F6	mov a, 4
0x40B800	....

Lecture 3

C Programming Model

Function Calls and the Program Counter

- **Question:** where is the old program counter stored so that it can be restored when the callee returns?
- **Answer:** Each activation record in the stack also contained a copy of the frame pointer. The frame pointer was stored in the stack so that a different value of it could be restored for each function return.
- Each function call needs to return to the address from which it was invoked

Lecture 3

C Programming Model

Function Calls and the Program Counter

- When making a function call, the compiler:
  - Pushes the return address (the current program counter) into the stack.
  - Pushes the frame pointer into the stack.
  - Sets the frame pointer equal to the stack pointer.
  - Decrements the stack pointer by as many memory addresses as are required to store the local state of the callee.
- when returning from a function, the compiler:
  - Sets the stack pointer equal to the frame pointer.
  - Pops the value of the old frame pointer from the stack.
  - Pops the value of the return address from the stack.
  - Jumps to the return address.

Lecture 3

C Programming Model

Function Pointers

- C doesn't provide a general way of manipulating the addresses of code, but the addresses of data
- For example:

```
int  (*newvar)(char);
```

Example 5 Variable declaration

```
newvar = &existing_function;
```

Example 6 Function pointer

```
int existing_function (char c) {

}
```

Example 7 Function skeleton

```
if ((*newvar)('a') == 5) {
    .....
}
```

Example 8 Calling a function via its pointer

C Programming Model

Function Pointers

- C doesn't provide a general way of manipulating the addresses of code, but the addresses of
- For example:

```
int  (*newvar)(char);
```

Example 5 Variable de

```
int exi        ction (char c) {
```

Example 7 Function skeleton

```
        &existing_function;
```

Example 6 Function pointer

```
if ((*newvar)('a') == 5) {
    .....
}
```

Example 8 Calling a function via its pointer

changing the function pointer would also change the method invoked

Lecture 3

C Programming Model

Registers: The CPU Also Has Memory

EXPAND

- The CPU also maintains its own banks of memory
- These memories cannot be directly manipulated from the programming language (for example, C).
- They usually do this by:
  - Providing a shorthand to refer to data that is currently in use.
  - Keeping data handy when it is needed repeatedly
- For instance, a value is allocated to address 0x00123456 in real memory, could be given a small address 0x3, or r3, in assembly, or EAX, in registers

Lecture 3

C Programming Model

Registers: The CPU Also Has Memory

- In any case, the values that the CPU keeps in its small memories are surrogates for the values that are stored in the real, big, main memory
- Who or what decides when to make a surrogate copy of a data item and how long the surrogate should exist ?

Lecture 3

C Programming Model

Registers: The CPU Also Has Memory

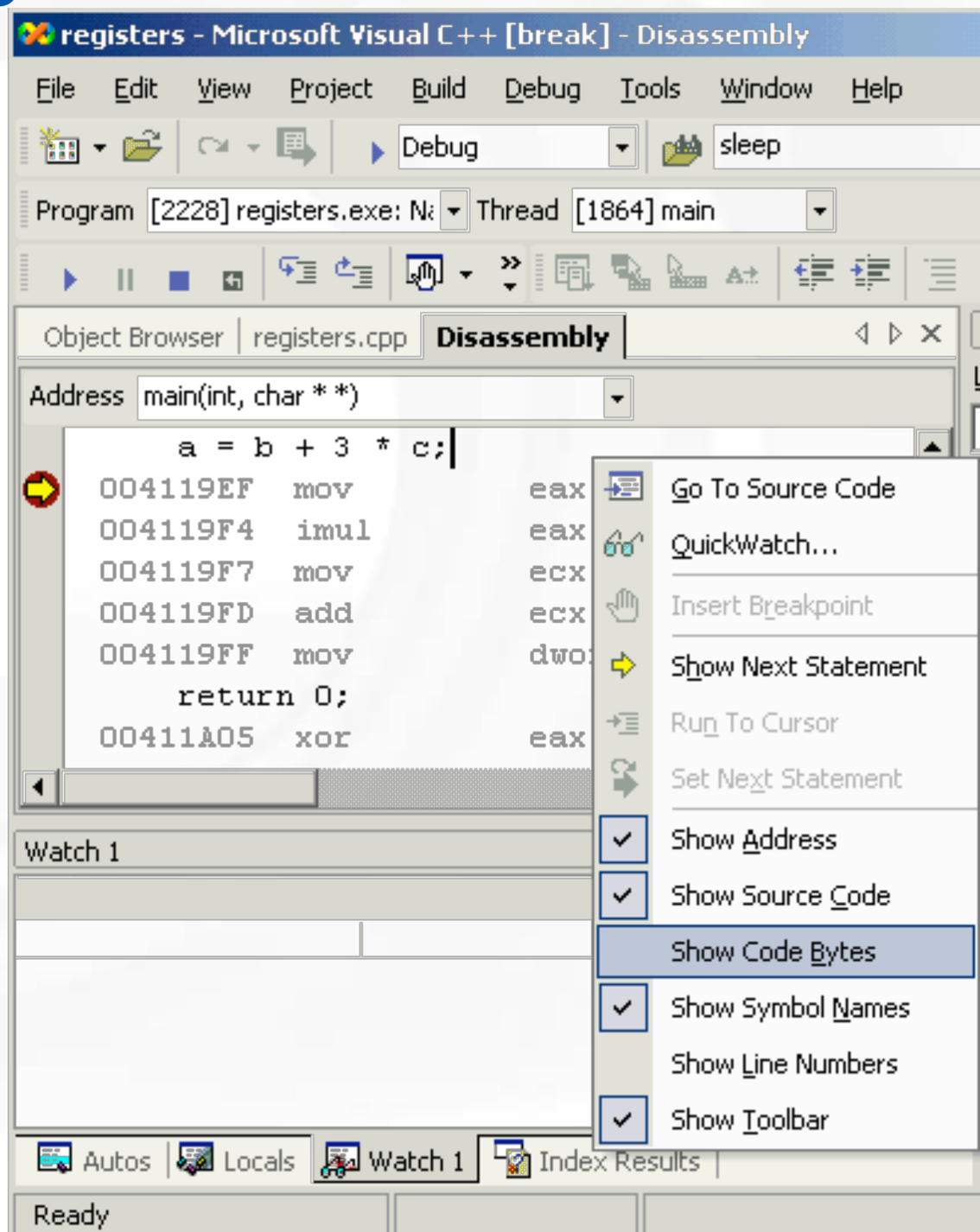
- The Compiler: The CPU memory that is managed explicitly by the compiler is usually called a ***register bank***, and the individual memory cells that store each data item are called ***registers***
- The Hardware: the hardware can create and destroy surrogates faster than the compiler, but it often creates surrogates for the wrong things; These memories are typically called ***caches***

Lecture 3

C Programming Model

Registers: Spying on Registers

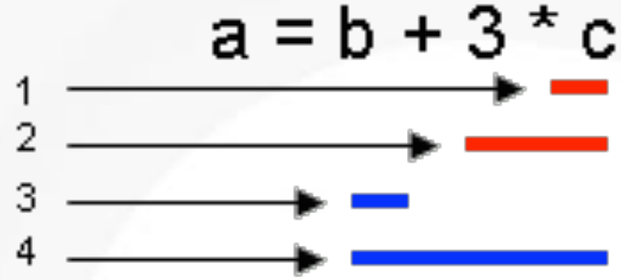
```
int a;
int b = 4;
int c = 3;

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

        a = b + 3 * c;
        return 0;
}
```

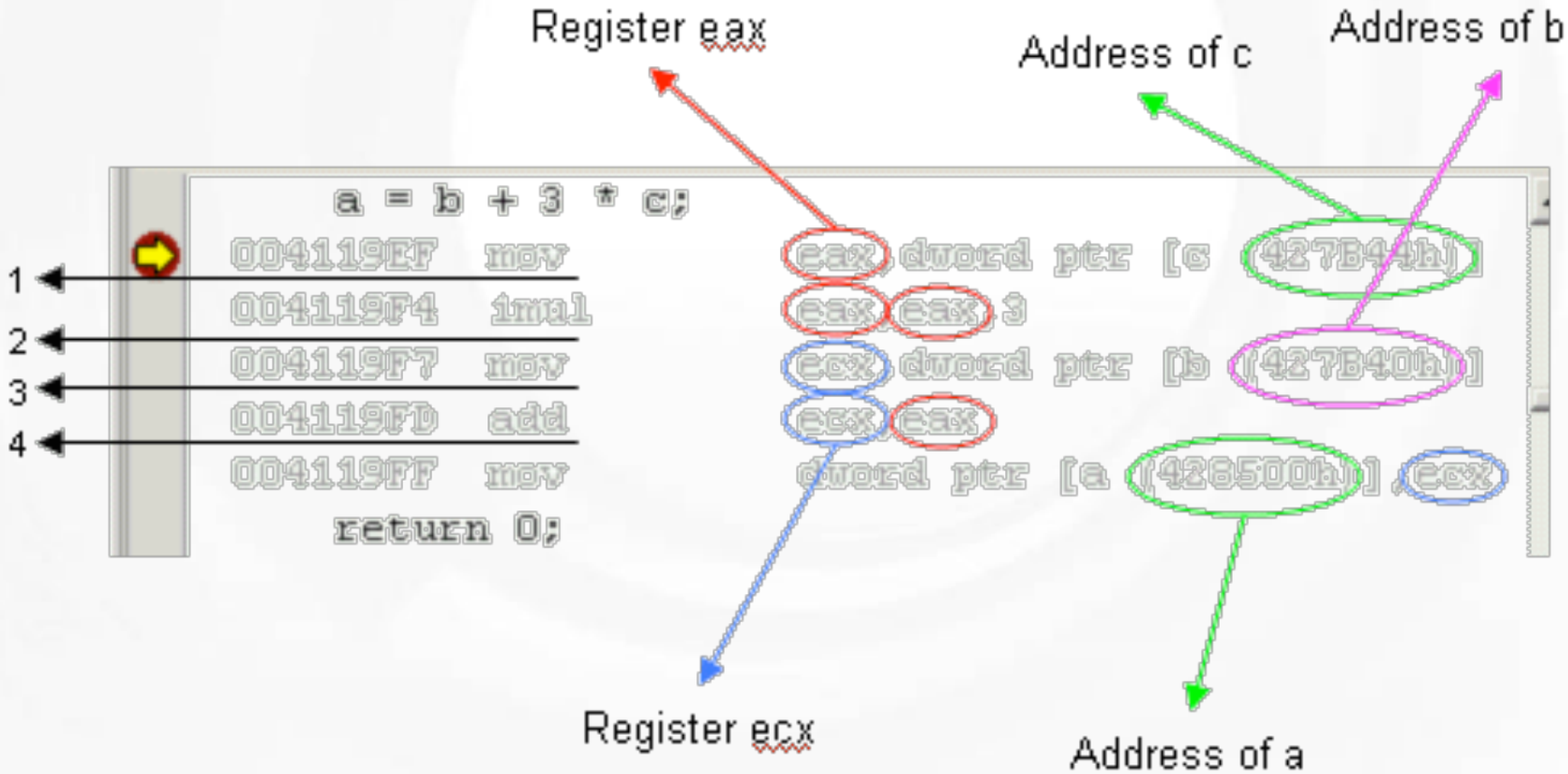


Lecture 3

C Programming Model

Registers: Spying on Registers





Lecture 3

C Programming Model

Register Allocation and Compiler Optimization

- CPUs only have a few registers, so the compiler needs to decide when and how to use each register as a surrogate for which variable
- This task, called register allocation, is so difficult that compilers do not try to allocate registers in the best way possible: it would take too long time
- Instead, they use rules of thumb to allocate registers to variables in ways that typically result in pretty good, though not the best, performance

Lecture 3

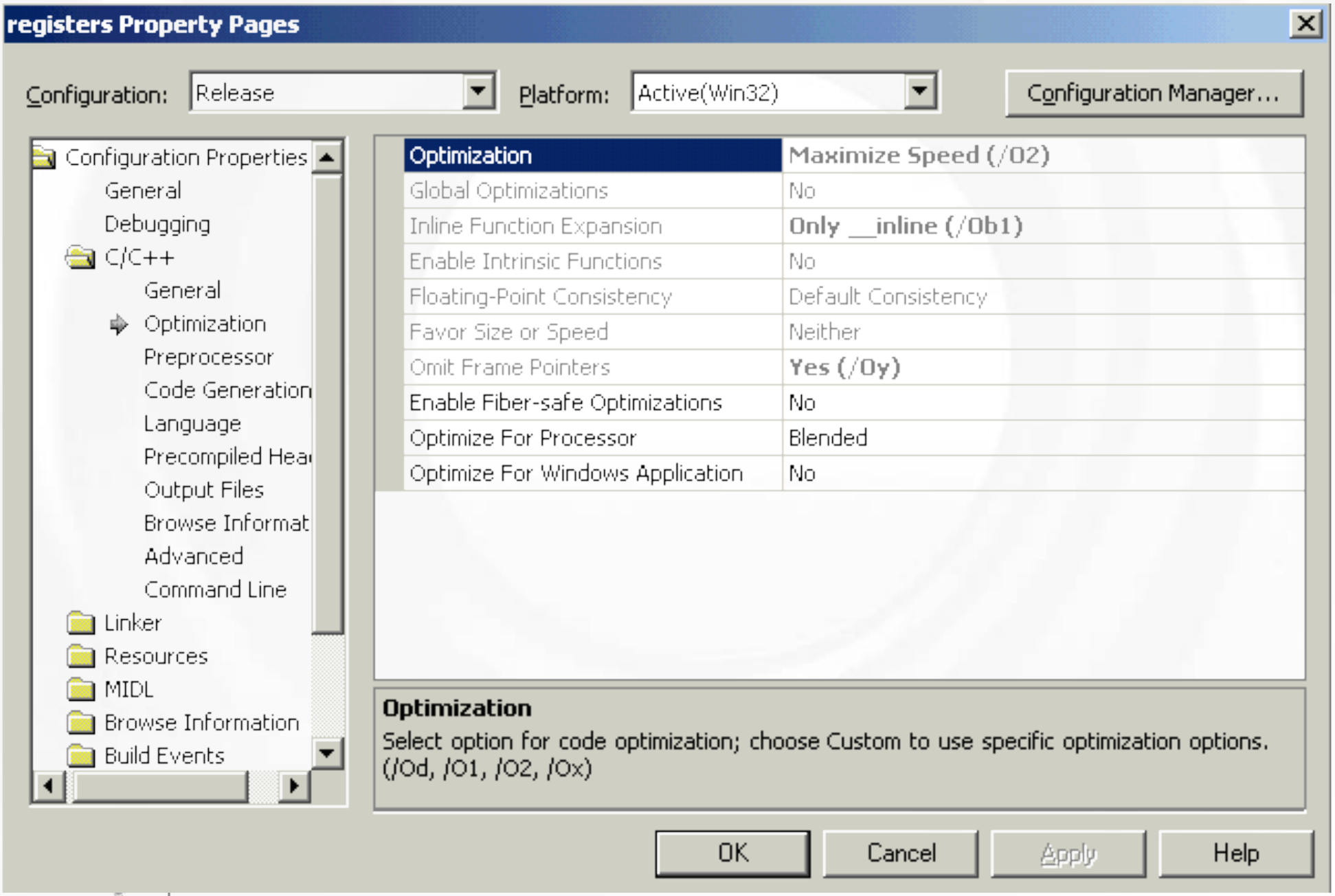
C Programming Model

Register Allocation and Compiler Optimization

- The extent to which compilers optimize register allocation depends on the ***optimization level***
- Higher optimization levels result in more aggressive optimizations which take longer to compile, whereas lower optimization levels result in slow execution
- Compilers do not always use their most aggressive optimizations because:
  - Compilation with aggressive optimizations takes a long time.
  - Aggressive optimizations change the code substantially: It may be difficult, both for people and for debuggers, to relate the resulting machine code to the original source code. Debugging, in particular, becomes almost impossible at high optimization levels

Lecture 3 C Programming Model

Register Allocation and Compiler Optimization



Lecture 3 C Programming Model

Programming competition in CMU

Goals:

Read numbers from a file (10,000 values)
Print out the average value of these numbers

Criteria:

Minimum executive time (CPU Time)
A must-be of Pascal/C coding
Allow to submit more than one solutions

Programming competition in CMU

The champion: -3”

The runner-up: n ms

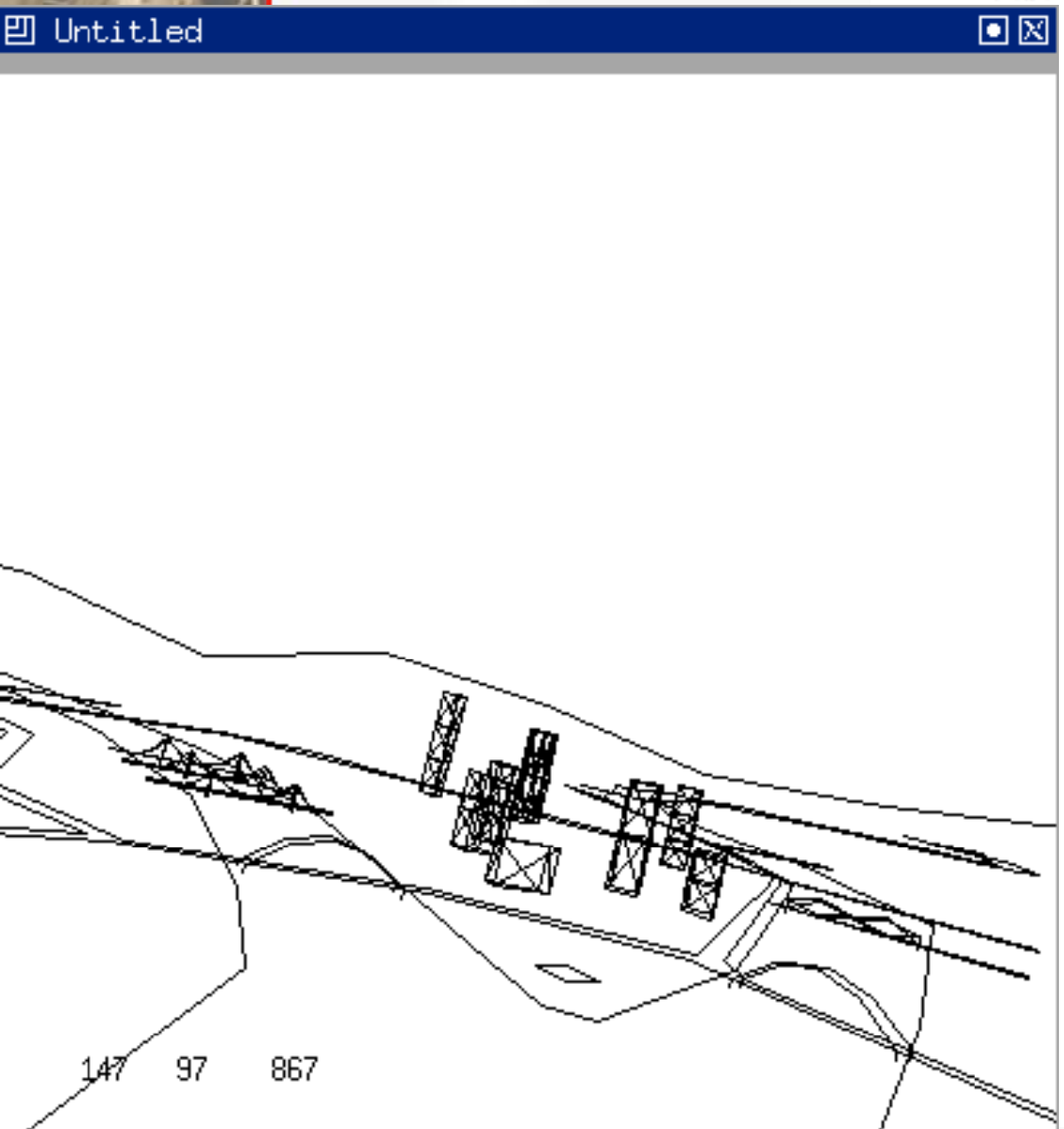
The third: 10”

Lecture 3

C P

IOCCC

The Internatio




```
#include                    <math.h>
#include                   <sys/time.h>
#include                   <X11/Xlib.h>
#include                  <X11/keysym.h>
                           double L ,o ,P
                          ,_=dt,T,Z,D=1,d,
                          e[999],E,h= 8,I,
                          J,K,w[999],M,m,O
                          ,n[999],j=33e-3,i=
                          1E3,r,t, u,v ,W,S=
                          74.5,l=221,X=7.26,
                          a,B,A=32.2,c, F,H;
                          int N,q, C, y,p,U;
                         Window z; char f[52]
                       ; GC k; main(){ Display*e=
 XOpenDisplay( 0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC (e,z,0,0),BlackPixel(e,0))
 ; scanf('%lf%lf%lf",y +n,w+y, y+s)+1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,0,400,400,
0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); ; T=sin(O)){ struct timeval G={ 0,dt*1e6}
; K= cos(j); N=1e4; M+= H*_; Z=D*K; F+=_*P; r=E*K; W=cos( O); m=K*W; H=K*T; O+=D*_*F/ K+d/K*E*_; B=
sin(j); a=B*T*D-E*W; XClearWindow(e,z); t=T*E+ D*B*W; j+=d*_*D-_*F*E; P=W*E*B-T*D; for (o+=(I=D*W+E
*T*B,E*d/K *B+v+B/K*F*D)*_; p<y; ){ T=p[s]+i; E=c-p[w]; D=n[p]-L; K=D*m-B*T-H*E; if(p [n]+w[ p]+p[s
]== 0|K <fabs(W=T*r-I*E +D*P) |fabs(D=t *D+Z *T-a *E)> K)N=1e4; else{ q=W/K *4E2+2e2; C= 2E2+4e2/ K
 *D; N-1E4&& XDrawLine(e ,z,k,N ,U,q,C); N=q; U=C; } ++p; } L-=_* (X*t +P*M+m*l); T=X*X+ l*l+M *M;
 XDrawString(e,z,k ,20,380,f,17); D=v/l*15; i+=(B *l-M*r -X*Z)*_; for(; XPending(e); u *=CS!=N){
                          XEvent z; XNextEvent(e ,&z);
                              ++*((N=XLookupKeysym
                                (&z.xkey,0])-IT?
                                N-LT? UP-N?& E:&
                                J:& u: &h); --*(
                                DN -N? N-DT ?N==
                                RT?&u: & W:&h:&J
                                 ); } m=15*F/l;
                                c+=(I=M/ l,l*H
                                +I*M+a*X)*_; H
                                =A*r+v*X-F*l+(
                                E=.1+X*4.9/l,t
                                =T*m/32-T*T/24
                                 )/S; K=F*M+(
                                 h* 1e4/l-(T+
                                 E*5*T*E)/3e2
                                 )/S-X*d-B*A;
                                 a=2.63 /l*d;
                                 X+=( d*l-T/S
                                  *(.19*E +a
                                  *.64+J/1e3
                                  )-M* v +A*
                                  Z)*_; l +=
                                  K ^_; W=d;
                                  sprintf(f,
                                  "%5d  %3d'
                                  "%7d",p =l
                                 /1.7,(C=9E3+
 O*57.3)%0550,(int)i]; d+=T*(.45-14/_*
X-a*130-J* .14)*_/125e2+F*_*v; P=(T*(47
*I-m* 52+E*94 *D-t*.38+u*.21*E) /1e2-W*
179*v)/2312; select(p=0,0,0,0,&G); v-=(
 W*F-T*(.63*m-I*.086+m*E*19-D*25-.11*u
  )/107e2)*_; D=cos(o); E=sin(o); } }
```

Lecture 3

Next Time……

- Next:
  - **Representation of Data**
    - 2.1 Bits and Bit Manipulation
    - 2.2 Integers
    - 2.3 Floating-Point Numbers
    - 2.4 Structured Data

Lecture 3 In the Lab……

- Excise 1:
  - **Unit 1. C Programming Model**
    - Decoding Lab: Understanding a Secret Message
- Rules:
  - **No changes to codes**
  - **Write your way for decoding**
- Hints:
  - **Memory allocation**
  - **Pointers & Address**