Principles of Programming Languages
Lecture 20

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig
PL Concepts and Constructs by Sethi
Concepts of PL by Sebesta
ML for the Working Prog. By Paulson
Prog. in Prolog by Clocksin and Mellish
PL Pragmatics by Scott

Components of an Imperative Language

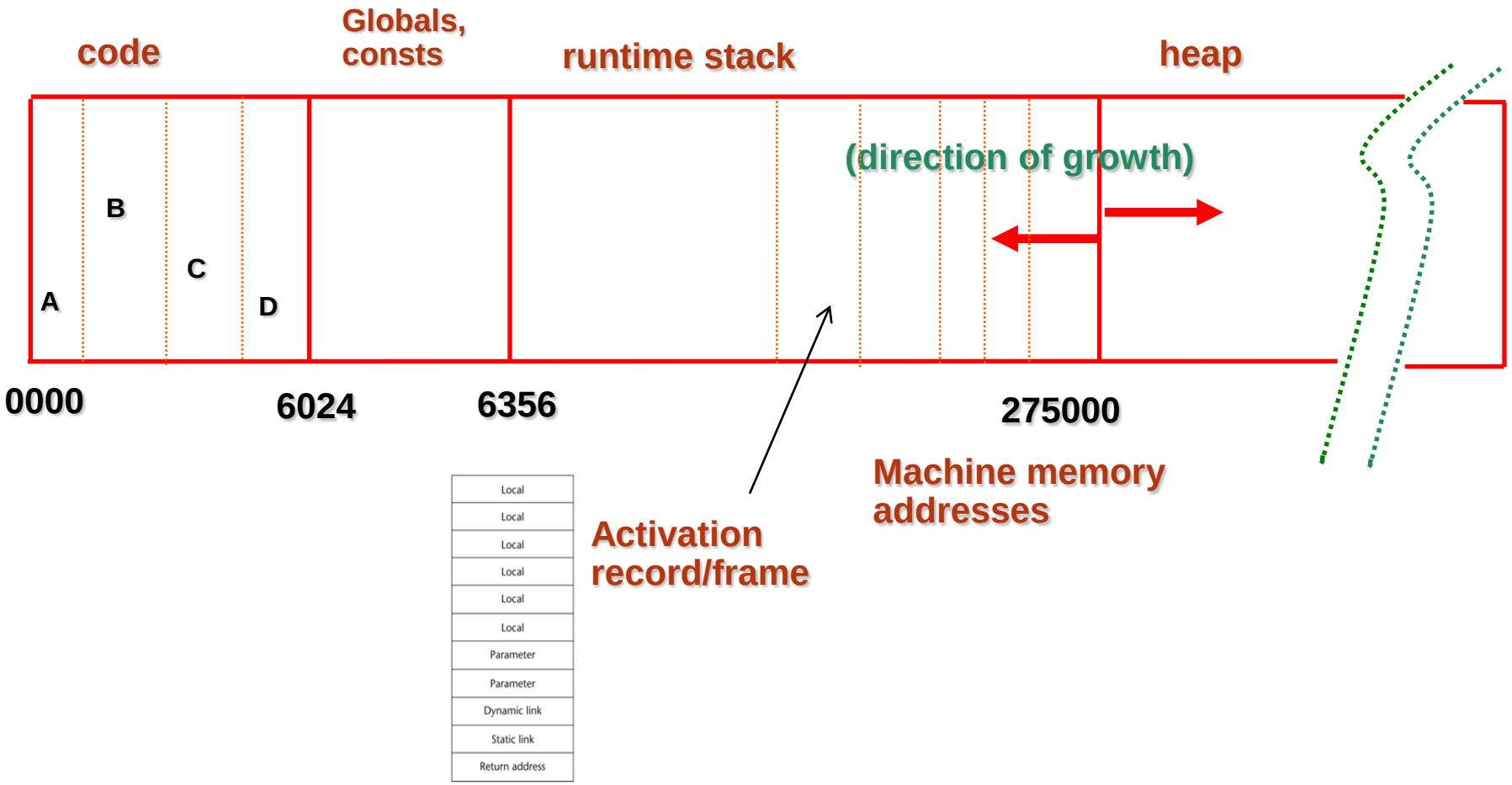
- **Data types**
- **Variables, operators & Expressions**
- **Iteration construct**
- **Branching construct**

→ **Subprogram construct**

- **Container construct**

Subprograms: big picture

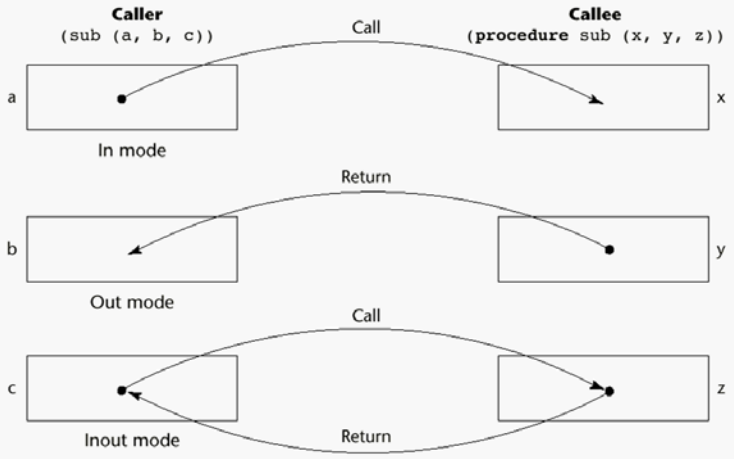
- **Sample memory layout**
  - A program with 4 sub-programs: A, B, C and D



Subprograms: parameter passing

- **How to treat arguments? how to implement argument passing?**

- **Semantic model:**
  - In mode
  - Out mode
  - In/out mode



- **Conceptual models of transfer:**
  - Physically move a value
  - Move an access path(pointer)

- **Implementation models:**
  - Pass by value
  - Pass by result
  - Pass by value-result
  - Pass by reference

Subprograms: parameter passing

- **Pass by Value**
  - Initial value of parameters are copied from current values of arguments
  - Final values of parameters are “lost” at return time (like local variables)

Subprograms: parameter passing

- **Pass by Value**
  - E.g.

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
          k := k+1;
          j := j+2;
    end r;
…
    m := 5;
    n  := 3;
    r(m,n);
    write m,n;
}
```

By Value:

k	j
~~5~~	~~3~~
6	5

Output:
5 3

Subprograms: parameter passing

- **Pass by Value**
  - Advantage:
    - Arguments protected from change in callee
  - Disadvantage:
    - Copying of values takes execution time and space, especially for aggregate values
  - PLs: C, Pascal, Ada, Scheme, Algol68

Subprograms: parameter passing

- **Pass by Result**
  - No initial value of parameters
  - Final values of parameters are copied back to arguments

Subprograms: parameter passing

- **Pass by Result**

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
        k := k+1;
        j := j+2;
    end r;
…
    m := 5;
    n  := 3;
    r(m,n);
    write m,n;
}
```

Error in procedure r: can't use parameters which are uninitialized!

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
        k := 1;
        j := 1;
    end r;
…
    m := 5;
    n  := 3;
    r(m,n);
    write m,n;
}
```

Subprograms: parameter passing

- **Pass by Result**
  - Disadvantage:
    - Requires copying of values, costs time and space, especially for large aggregates.
    - What if the argument is not a variable (e..g r(1,2))?
    - What if the variable is used twice in the argument list? (e.g. r(m,m))?
    - What about calculations to determine locations of arguments (e.g. c[m])?
  - PLs: Ada

Subprograms: parameter passing

- **Pass by Value-Result**
  - Initial values of parameters copied from current values of arguments
  - Final values of parameters copied back to arguments
  - Combines functionality of pass by value and pass by result for same parameter

Subprograms: parameter passing

- **Pass by Value-Result**

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
          k := k+1;
          j := j+2;
    end r;
…
    m := 5;
    n  := 3;
    r(m,n);
    write m,n;
}
```

By Value-Result

k	j
~~5~~	~~3~~
6	5

Output:
6 5

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
          k := k+1;
          j := j+2;
    end r;
 /* set c[m] = m */
    m := 2;
    r(m, c[m]);
    write c[1], c[2], …, c[10];
}
```

k	j
~~2~~	~~2~~
3	4

What element of c has its value changed? c[2]? c[3]?

Subprograms: parameter passing

- **Pass by Reference**
  - Formal parameters are pointers to the actual parameters (arguments)
  - Address computations are performed at procedure call
  - Changes to the formal parameters are thus changes to the actual parameters

Subprograms: parameter passing

- **Pass by Reference**

```
{   c: array [1..10] of integer;
    m,n : integer;
    procedure r (k,j : integer)
    begin
          k := k+1;
          j := j+2;
    end r;
…
    m := 5;
    n  := 3;
    r(m,n);
    write m,n;
}
```

k	j
--> m	-->n

m	n
~~5~~	~~3~~
6	5

Value update happens in storage of the caller while callee is executing

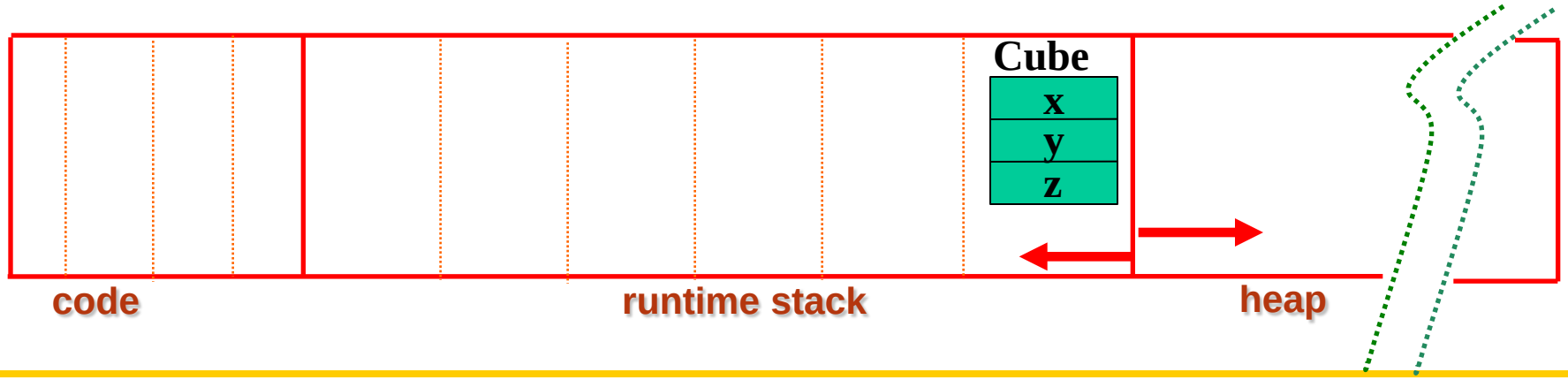
Subprograms: parameter passing

- **Pass by Reference**
  - Advantage:
    - More efficient than copying
  - Disadvantages:
    - Can redefine constants and expressions (e.g. Fortran66: foo(0,x))
    - Aliasing: when there are two or more different names for same storage location.
    - If an error occurs, harder to trace values since some side effected values are in environment of the caller.
  - PLs: Fortran, Pascal var params, Cobol

Explicit Mode Selection

- **Programmer chooses pass-by-value or pass-by-reference**

```
class Cube {              // C++ example
        public:
                Cube::Cube(float inx, float iny, float inz);
                float x; float y; float z;
};
float calcArea( Cube cube ){
   …..
}
                calcArea(someCube);
```



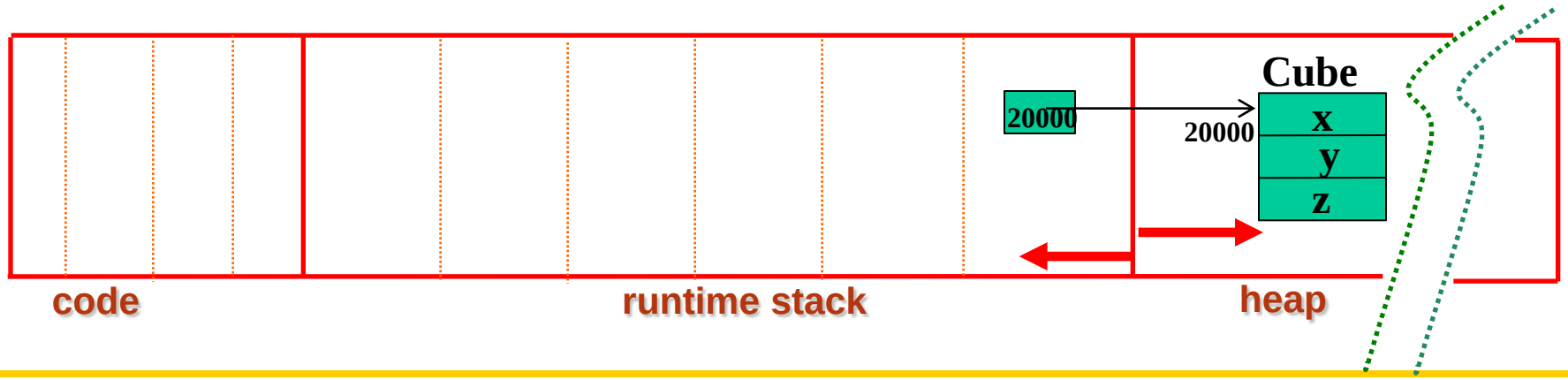
Explicit Mode Selection

- **Programmer chooses pass-by-value or pass-by-reference**

```
class Cube {              // C++ example
        public:
                Cube::Cube(float inx, float iny, float inz);
                float x; float y; float z;
};
```

```
float calcArea( Cube& cube ){
  …..
}
```

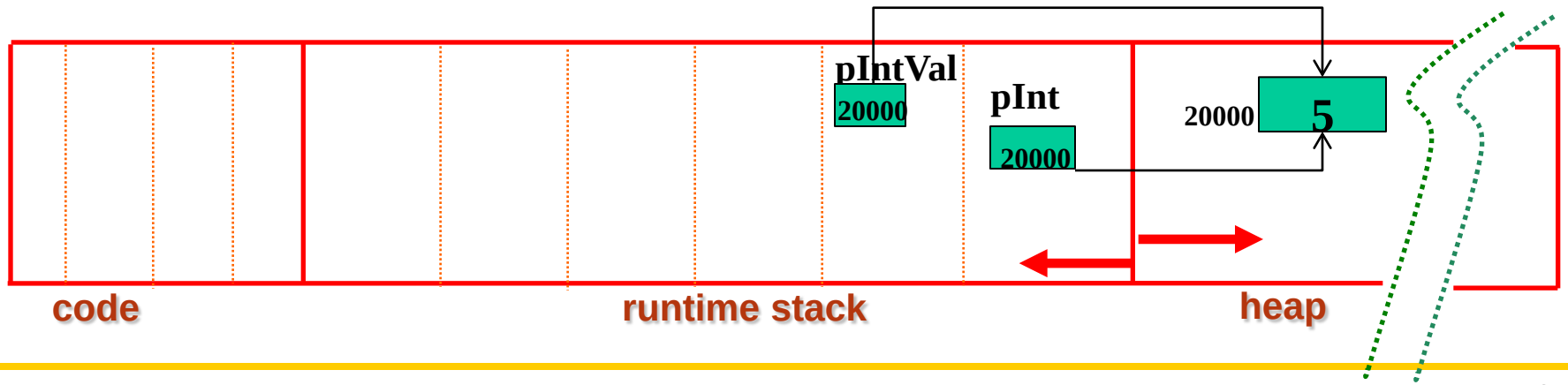
```
calcArea(someCube);
```



Working With Pointers

- **Since a pointer is primitive, it is passed by value → trouble for pointer manipulation!**
- **E.g.**

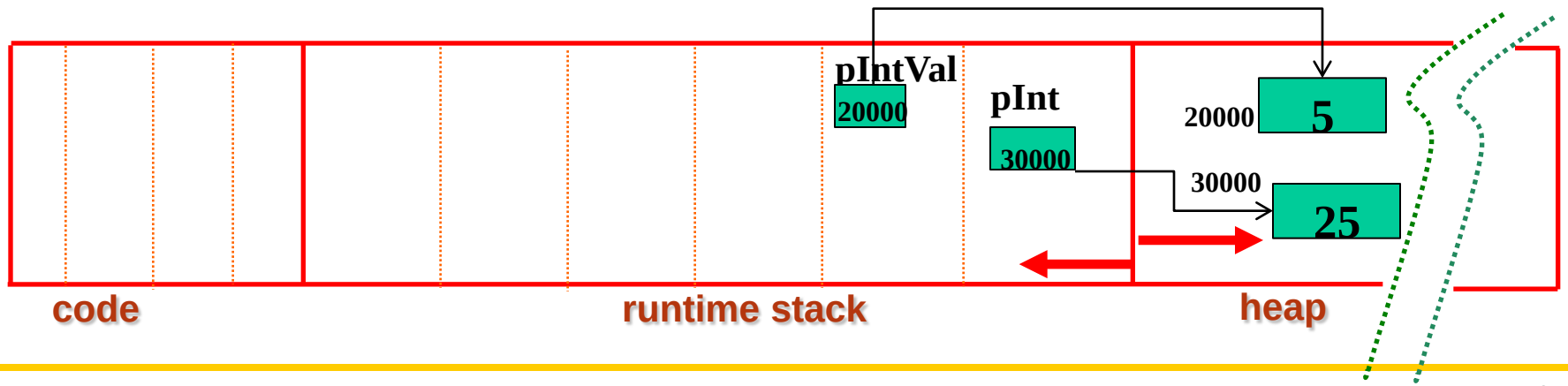
```
void changePointer(int* pInt){
        pInt = SomeOtherIntPointer;
}
int*  pIntVal;
*pIntVal = 5;
changePointer(pIntVal);
```



Working With Pointers

- **Since a pointer is primitive, it is passed by value → trouble for pointer manipulation!**
- **E.g.**

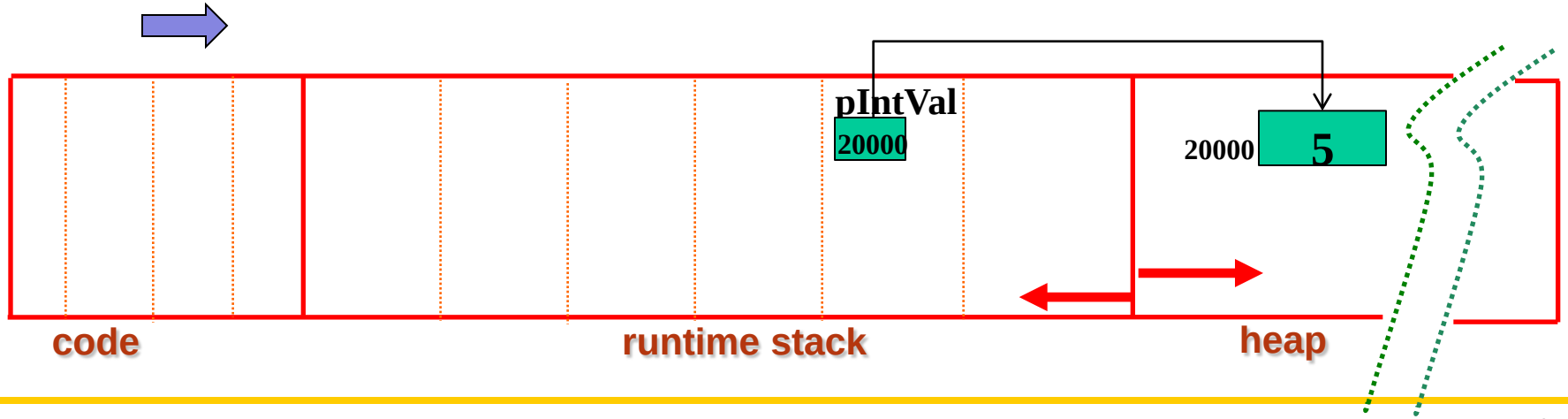
```
void changePointer(int* pInt){
    pInt = SomeOtherIntPointer;
}
int*  pIntVal;
*pIntVal = 5;
changePointer(pIntVal);
```



Working With Pointers

- Since a pointer is primitive, it is passed by value → trouble for pointer manipulation!
- E.g.

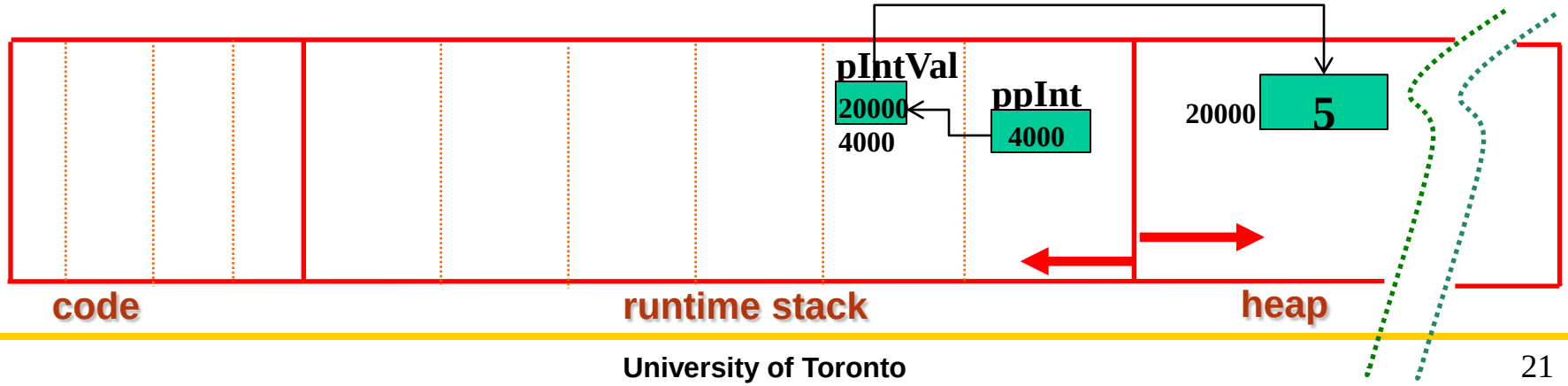
```
void changePointer(int* pInt){
        pInt = SomeOtherIntPointer;
}
int*  pIntVal;
*pIntVal = 5;
changePointer(pIntVal);
```



Working With Pointers

- Since a pointer is primitive, it is passed by value → trouble for pointer manipulation!
- E.g.

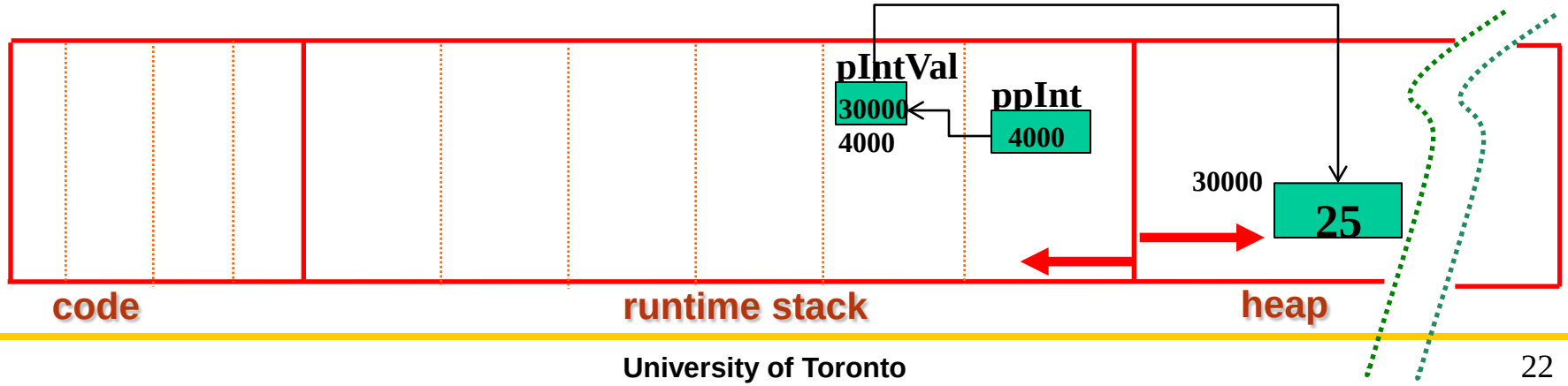
```
void changePointer(int** ppInt){
        *ppInt = SomeOtherIntPointer;
}
int*  pIntVal;                  *pIntVal = 5;
int ** AddressOfPointer;
AddressOfPointer = &pIntVal;
changePointer(AddressOfPointer);
```



Working With Pointers

- Since a pointer is primitive, it is passed by value → trouble for pointer manipulation!
- E.g.

```
void changePointer(int** ppInt){
    *ppInt = SomeOtherIntPointer;
}
int*  pIntVal;                  *pIntVal = 5;
int ** AddressOfPointer;
AddressOfPointer = &pIntVal;
changePointer(AddressOfPointer);
```



Syntactical Variations

- **Default values for params & variable number of params**

- **E.g.**

```
def testParamPassing(X,Y,Z=30):
        print "X is", X
        print "Y is", Y
        print "Z is", Z

if __name__ == "__main__":
        testParamPassing(10, 20)
        testParamPassing(10,20,40)
```

Syntactical Variations

- **Param names specified in function call**
- **E.g.**

```
def testParamPassing(X,Y,Z):
    print "X is", X
    print "Y is", Y
    print "Z is", Z

if __name__ == "__main__":
    testParamPassing(X=10, Y=20,Z=30)
    testParamPassing(Y=20, Z=30,X=10)
```