Overloading and Resolution

• Overloading: one name refers to multiple entities in scope.
• Overload Resolution: select one entity.
• Name itself isn’t sufficient in resolution: need extra information (often data types)

Function/Method Overloading

• C: no overloading
• C++/Java/Ada: resolution by number and types of parameters.
  – Perfect if exact match exists;
  – No perfect match: different conversion rules
    • Ada: automatic conversions not allowed.
    • Java: conversions allowed in certain directions.
    • C++: automatic conversions more flexible.
    • e.g.,
      • int sum(int a, int b) {...}
      • double sum(double a, double b) {...}
      • double sum(double a, int b) {...}
      • sum(1); sum(1, 2); sum(1.0, 2.0); sum(1, 2.0);
    • Ada even uses return type for resolution

Environment: Names to Locations

• Location: one specific attribute of names
• Environment: binding names to locations
  – Conceptually part of symbol table.
  – But usually considered separately.
• Static vs. dynamic
  – FORTRAN: completely static
  – LISP: completely dynamic
  – Algol-descendants (C, C++, Ada, Java): combination
  – global variables: static
  – local variables: dynamic

Environment: as Bindings

Environment: another view
Stack-Based Allocations

- **Stack**
  - static (global) area
  - automatically-allocated spaces (local variables, procedures (chapter 8) under the control of runtime system
  - (unallocated)

- **Heap**
  - manually-allocated spaces under the control of programmer

Example

A: { int x; char y; }  
B: { double x; int a; }  
C: { char y; int b; }  
D: { int x; double y; }  

Lifetime is the duration of B

Example

A: { int x; char y; }  
B: { double x; int a; }  
C: { char y; int b; }  
D: { int x; double y; }

x  
y  

Chapter 5  
K. Louden, Programming Languages
• What are the most difficult bugs that you have to deal with?

Heap-Based Allocation

• C
  int *x;
  x=(int *)malloc(sizeof(int));
  free(x);
• C++
  int *x;
  x = new int;
  delete x;
• Java
  Integer x = new Integer(2);
  //no delete
  //need garbage collection

Scope vs. Lifetime

• Lifetime beyond scope:
  – alive in scope hole
  – alive outside scope

• Scope beyond lifetime (unsafe)

Example: Alive in scope hole

A: {  int x;
     char y;
   } B: {  double x;
     int a;
   } C: { char y;
     int b;
   } D: { int x;
     double y;
   } x y x a

Example: Alive outside scope

int func(void) {
  static int counter = 0;
  counter += 1;
  return counter;
}

main()
{
  int i;
  int x;
  for (i=0; i<10; i++) { x=func(); }
  printf("%d\n", x);
}

Example: Scope beyond lifetime

Dangling pointer:

int *x, *y, *z;

x=(int *)malloc(sizeof(int));
y=x;
free(x);
z=malloc(sizeof(int));

printf("%d\n",y);
Box-and-Circle Diagram for Variables

Box-and-Circle Diagram for Variables

Java:
Student x = new Student("Amy");
Student y = new Student("John");
x.setAge(19);
x = y;
y.setAge(21);

• Alias that causes side-effects.
  Student x = new Student("Amy");
  Student y = new Student("John");
  x.setAge(19);
  x = y;
  y.setAge(21);

• Dangling References:
  int *x;
  { int y;
    y = 21;
    x = &y;
    printf("%d", *x);
  }

Problems with Pointers