# LECTURE 12

Names, Scopes, and Bindings

# OVERVIEW

- Abstractions and names
- Binding time
- Object and binding lifetime

# ABSTRACTION

The term “high-level” programming languages comes from the higher degree of abstraction provided by these languages as opposed to assembly or machine language.

In this context, *abstraction* refers to the separation of features from the details of their implementation.

A high degree of abstraction is desirable because it allows developers to create programs which work well on a variety of architectures (implementations) but it also makes programs easier to understand.

# MACHINE INDEPENDENCE

Machine independence, or platform independence, is an attribute of programs which do not rely on features of any particular instruction set for their implementation.

For the most part, it has been a long time since greater machine independence has been a goal in the design of a new language.

However, the other advantage of abstraction, greater ease of programming, has consistently been a goal in the design of new languages.

# NAMES

A *name* is a mnemonic character string used to represent something else.

- Typically alphanumeric characters (i.e. “myint”) but can also be other symbols (i.e. ‘+’).
- Names enable programmers to refer to variables, constants, operations, and types instead of low level concepts such as memory addresses.
- Names are essential in high-level languages for supporting abstraction.
  - In this context, abstraction refers to the ability to hide a program fragment behind a name.
  - By hiding the details, we can use the name as a black box. We only need to consider the object’s purpose, rather than its implementation.

# NAMES

Names enable control abstractions and data abstractions in high level languages.

- **Control Abstraction**
  - Subroutines (procedures and functions) allow programmers to focus on a manageable subset of program text, subroutine interface hides implementation details.
  - Control flow constructs (if-then, while, for, return) hide low-level machine ops.
- **Data Abstraction**
  - Object-oriented classes hide data representation details behind a set of operations.

# BINDING

A *binding* is an association between a name and an entity. The *binding time* is the time at which a binding is created, or in other words, when an implementation decision is made. There are many different times when binding can occur:

- **Language design time:** the design of specific language constructs.
  - Syntax (names ←→ grammar)
    - if (a>0) b:=a; (C syntax style)
  - Keywords (names ←→ builtins)
    - class (C++ and Java), extern
  - Reserved words (names ←→ special constructs)
    - main (C)
  - Meaning of operators (operator ←→ operation)
    - + (add), % (mod), ** (power)
  - Built-in primitive types (type name ←→ type)
    - float, short, int, long, string

# BINDING

- **Language implementation time**: fixation of implementation constants.
  - Examples: precision of types, organization and maximum sizes of stack and heap, etc.
- **Program writing time**: the programmer's choice of algorithms and data structures.
  - Examples: A function may be called sum_grades(), a variable may be called x.
- **Compile time**: the time of translation of high-level constructs to machine code and choice of memory layout for data objects.
  - Example: translate “for(i=0; i<100; i++) a[i] = 1.0;”?
- **Link time:** the time at which multiple object codes (machine code files) and libraries are combined into one executable.
  - Example: which cout routine to use? /usr/lib/libc.a or /usr/lib/libc.so?

# BINDING

- **Load time**: when the operating system loads the executable in memory.
  - Example: In an older OS, the binding between a global variable and the physical memory location is determined at load time.
- **Run time**: when a program executes.
  - Example: Binding between the value of a variable to the variable.

# BINDING

- Early binding times (before run time) are associated with greater efficiency and clarity of program code.
  - Compilers make implementation decisions at compile time.
  - Syntax and static semantics checking is performed only once at compile time and does not impose any run-time overheads.
- Late binding times (at run time) are associated with greater flexibility (but may leave programmers sometimes guessing what's going on).
  - Interpreters allow programs to be extended at run time.
  - Languages such as Smalltalk-80 with polymorphic types allow variable names to refer to objects of multiple types at run time.
  - Method binding in object-oriented languages must be late to support dynamic binding.

# OBJECT LIFETIME

Key events in an object's lifetime:

- Object creation.
- Creation of bindings.
- The object is manipulated via its binding.
- Deactivation and reactivation of (temporarily invisible) bindings. (in-and-out of scope)
- Destruction of bindings.
- Destruction of object.

The time between binding creation and binding destruction is the *binding's lifetime*.
The time between object creation and object destruction is the *object's lifetime*.

# OBJECT LIFETIME

It is typical for a binding lifetime to be shorter than the object's lifetime.

- Example: passing an argument by reference to a function in C++. We are creating a new binding to an object that already existed. When we return from the function, the binding is destroyed but the object is still accessible.

```
void Twice(int&, int&);
int main()
{
    int x = 5, y = 8;
    Twice(x,y);
    cout << "x is: " << x << endl;
    return 0;
}

void Twice(int& a, int& b)
{
    a *= 2;
    b *= 2;
}
```

# DANGLING REFERENCE

When the binding lifetime exceeds the object's lifetime, we have a *dangling reference*. Typically, this is a sign of a bug.

```
…
myobject = new SomeClass;
foo(myobject);

foo(SomeClass *a)
{
  ……
  delete (myobject);  // myobject is a global variable
  a->action();
}
```

# DANGLING REFERENCE

Where is the dangling reference?

```
#include <iostream>
#include <string.h>
using namespace std;

char * ptr;          // a global variable
void foo();

int main() {
  foo();
  cout << ptr;
  return 0;
}

void foo() {
  char buff[1000];
  cin >> buff;
  ptr = strtok(buff , “;");
}
```

# DANGLING REFERENCE

Where is the dangling reference?

```
carnahan@diablo:~/COP4020>./a.out
hello;class
hello
```

```
#include <iostream>
#include <string.h>
using namespace std;

char * ptr;          // a global variable
void foo();

int main() {
  foo();
  cout << ptr;
  return 0;
}

void foo() {
  char buff[1000];
  cin >> buff;
  ptr = strtok(buff , ";");
}
```

# DANGLING REFERENCE

Where is the dangling reference?

```
carnahan@diablo:~/COP4020>./a.out
hello;class
hello
```

Behavior is undefined. We're referencing an object that doesn't exist anymore, but it hasn't been replaced on the stack.

```
#include <iostream>
#include <string.h>
using namespace std;

char * ptr;          // a global variable
void foo();

int main() {
  foo();
  cout << ptr;
  return 0;
}

void foo() {
  char buff[1000];
  cin >> buff;
  ptr = strtok(buff , ";");
}
```

# DANGLING REFERENCE

What will we get now?

```
#include <iostream>
#include <string.h>
using namespace std;
char * ptr;                    // a global variable
void foo();
void bar();
int main() {
  foo();
  bar();
  cout << ptr;
  return 0;
}
void foo() {
  char buff[1000];
  cout << "Foo_buff: ";
  cin >> buff;
  ptr = strtok(buff , ";");
}
void bar() {
  char buff[1000];
  cout << "Bar_buff: ";
  cin >> buff;
}
```

# DANGLING REFERENCE

What will we get now?

```
carnahan@diablo:~/COP4020>./a.out
Foo_buff: hello;class
Bar_buff: goodbye;class
goodbye;class
```

```
#include <iostream>
#include <string.h>
using namespace std;
char * ptr;                 // a global variable
void foo();
void bar();
int main() {
  foo();
  bar();
  cout << ptr;
  return 0;
}
void foo() {
  char buff[1000];
  cout << "Foo_buff: ";
  cin >> buff;
  ptr = strtok(buff , ";");
}
void bar() {
  char buff[1000];
  cout << "Bar_buff: ";
  cin >> buff;
}
```

# MEMORY LEAKS

When all bindings are destroyed, but the object still exists, we have a *memory leak*.

```
{
  SomeClass* myobject = new SomeClass;
  ...
  ...
  myobject->action();
  return;
}
```

# NEXT TIME

Continuing Names, Scopes, and Bindings with Memory Management Schemes