CSE 165/ENGR 140
Intro to Object Orient Program

Lecture 4 – C in C++
Announcement

- Lab #1 this Friday (1/30)
  - Due 2/5 (Thursday) at 11:59PM.

- Reading assignment
  - Ch. 3
Data types: pointers and references

- Pointers are variables containing a memory address.
- Every variable, object, and function has an address.
- References are introduced in C++ as a new way to work with the address of a variable.
  - it avoids the sometime heavy syntax needed to work with pointers, and allowing the same kind of functionality.
- Contrary to pointers references are always valid
  - pointer can be “null” or be of void type (void* pt), references cannot.
### Memory: address and value

<table>
<thead>
<tr>
<th>identifier</th>
<th>name</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>radius</th>
<th>i(0)</th>
<th>i(1)</th>
<th>i(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>steven</td>
<td>.345</td>
<td>-2.56</td>
<td>-.1</td>
<td>.2222222222</td>
<td>234</td>
<td>-10</td>
<td>1000</td>
</tr>
</tbody>
</table>

#### Address

<table>
<thead>
<tr>
<th>address</th>
<th>1459</th>
<th>1460</th>
<th>1461</th>
<th>1462</th>
</tr>
</thead>
</table>

#### Reference

<table>
<thead>
<tr>
<th>reference</th>
<th>&amp;name</th>
<th>&amp;f1</th>
<th>&amp;f2</th>
<th>&amp;f3</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>&amp;radius</th>
<th>&amp;i(0)</th>
<th>&amp;i(1)</th>
<th>&amp;i(2)</th>
</tr>
</thead>
</table>
Reference and De-reference

**type&**  reference definition

**type**  pointer definition

&& reference operator:  "address of"

* dereference operator:  "value pointed by"

Equivalent to:

&&:  Address of  John  is 52 Main Street

*:  Value at  52 Main Street  is John
andy = 25; // andy contains 25
fred = andy; // fred contains the value of andy (25)
ted = &andy; // tted contains the address of andy (1776)
De-reference operator

beth = ted;       // beth equal to ted (1776)
beth = *ted;      // beth equal to value pointed by ted (25)
Variables of pointer type

- Declaration:
  - `type * name;`
  - `type* name;`
  - `type *name;`

- Examples:
  - `int * address;`  // address is a pointer of type int
  - `char *p_ch;`     // p_ch is a pointer of type char
  - `float* p;`       // p is a pointer of type float
Example 1:
  ◦ int number;
  ◦ int* p_number = &number;

Example 2:
  ◦ int number;
  ◦ int* p_number;
  ◦ p_number = &number;

Example 3:
  ◦ *p_number = &number;  // makes sense?
  ◦ p_number = 1000  // makes sense?
Variables and pointers

- **Declaration:**
  - type * name1, * name2; // declares 2 pointers type*
  - type * name1, name2; // declares 1 pointer type and 1 variable type

- **Definition:**
  - int *p_i; int i, j;
  - i = 10;
  - p_i = &i;
  - *p_i = 20;
  - p_i = &j;
  - *p_i = 10;
//: C03:YourPets2.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
    cout << "pet id number: " << pet << endl;
}

int main() {
    int i, j, k;
    cout << "f(): " << (long)&f << endl;
    cout << "dog: " << (long)&dog << endl;
    cout << "cat: " << (long)&cat << endl;
    cout << "bird: " << (long)&bird << endl;
    cout << "fish: " << (long)&fish << endl;
    cout << "i: " << (long)&i << endl;
    cout << "j: " << (long)&j << endl;
    cout << "k: " << (long)&k << endl;
}
Reference examples

//: C03:PassReference.cpp
#include <iostream>
using namespace std;
void f ( int& r ) { // Accepting reference
    cout << "r = " << r << endl;
    cout << "&r = " << &r << endl;
    r = 5;
    cout << "r = " << r << endl;
}

int main() {
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f(x); // Looks like pass-by-value, but is actually
    pass by reference
    cout << "x = " << x << endl;
}
Types: operators

- Mathematical operators:
  - addition (+), subtraction (-), division (/), multiplication (*)
    - integer division truncates the result (it doesn’t round)
  - modulus (%); remainder from integer division.
    - cannot be used with floating-point numbers.
  - Assignment operators: +=, -=, *=, /=, etc

- Logical operators
  - **and** (&&), **or** (||) produce **true** or **false**
  - in C and C++ a statement is **true** if it has a non-zero value, and **false** if it has a value of zero.
  - Negation operator not (!)
  - **==** comparison operator
    - different from assignment op. (=)
Types: operators

- **Bitwise operators**
  - bitwise and (&)
    - 1 if both input bits are 1; otherwise 0
  - bitwise or (|)
    - 1 if either input bits are 1; 0 only if both are 0.
  - bitwise exclusive or, or xor (^)
    - 1 if an input bit is one, but not both; otherwise 0.
  - bitwise not (~)
    - unary operator that inverts the input bit
  - Also valid: &=, |=, and ^=

- **Shift operators**
  - left-shift operator (<<) and right-shift operator (>>)
    - Also valid: <<= and >>=
  - One bit is always lost in a shift operation

- **Ternary operator “? :”** (a? b:c)
//: C03:printBinary.cpp {O}
#include <iostream>
void printBinary ( const unsigned char val ) {
    for ( int i=7; i>=0; i-- )
        std::cout << ( val & (1<<i)? "1" : "0" );
}

unsigned char rotateLeft ( unsigned char val ) {
    int highbit;
    if (val & 0x80) // 0x80 is the high bit only
        highbit = 1; // (statement can be written with ?: operator as above)
    else
        highbit = 0;
    // Left shift (bottom bit becomes 0):
    val <<= 1;
    // Rotate the high bit onto the bottom:
    val |= highbit;
    return val;
}
Types: casting

- Types can be converted by C-like type-casts in parenthesis.

```cpp
//: C03:FunctionCallCast.cpp
int main() {
    float a = float(200);
    // This is equivalent to:
    float b = (float)200;
}
```

- In C++ we should use:
  - `static_cast`: simple casts for type conversion
  - `const_cast`: to remove const
  - `reinterpret_cast`: to cast an object to something completely different
  - `dynamic_cast`: type-safe cast with run-time checking, can be used only with pointers and references to objects
void func(int) {} 

int main() {
    int i = 0x7fff; // Max pos value = 32767
    long l;
    float f;
    // 1a) Typical castless conversions:
    l = i;
    f = i; // may generate a warning

    // 1b) C-style type casts:
    l = (long)i;
    f = (float)i; // no warning

    // 1c) C++ way:
    l = static_cast<long>(i);
    f = static_cast<float>(i);
//: C03:static_cast.cpp (continuation)

// 2a) Automatic narrowing conversions:
i = f; // May lose digits (will generate a warning)

// 2b) Says "I know," eliminates warnings:
i = (int)(l); // C style
i = (int)(f); // C style

// 2c) C++ way:
i = static_cast<int>(l); // C++ style
i = static_cast<int>(f); // C++ style
char c = static_cast<char>(i);

// 3a) Forcing a conversion from void* :
void* vp = &i;

// 3b) Old way produces a “dangerous” conversion:
float* fp = (float*)vp;

// 3c) The new way is equally dangerous:
fp = static_cast<float*>(vp);
// etc
}
Types: const_cast

```cpp
//: C03:const_cast.cpp
int main() {
    // 1) const values should not be modified:
    const int i = 0;

    // 2) But we can get a pointer to them and later modify them...:
    int* j = (int*)&i;          // Deprecated form
    j = const_cast<int*>(i);   // Preferred
    *j = 1;

    // 3) Can't do simultaneous additional casting:
    //! long* l = const_cast<long*>(i); // Error
}
```
```cpp
//: C03:reinterpret_cast.cpp
#include <iostream>
using namespace std;
const int sz = 100;

struct X { int a[sz]; }

void print(X* x) {
    for (int i = 0; i < sz; i++)
        cout << x->a[i] << ' ';
    cout << endl;
}

int main() {
    X x;
    print(&x);
    int* xp = reinterpret_cast<int*>(&x); // Cast to integer
    for (int* i = xp; i < xp + sz; i++)
        *i = 0;

    // Can't use xp as an X* unless you cast it back:
    print(reinterpret_cast<X*>(xp));
}
```
Types: reinterpret_cast

class BaseClass { ... };

class Class1 : public BaseClass {...}; // BaseClass "derives" Class1

class Class2 {...}; // BaseClass does not derive Class2

BaseClass *pb; // pointer to BaseClass
Class1 *p1; // pointer to Class1

p1 = static_cast<Class1*>(pb); // Ok as long as we know pb can point to Class1
p1 = (Class1*)(pb); // C-style also ok, same as static_cast<>

Class2 *p2; // pointer to Class2
p2 = static_cast<Class2*>(pb); // Compiler error, can't convert
p2 = (Class2*)(pb); // No compiler error...
  // Same as reinterpret_cast<>

p2 = reinterpret_cast<Class2*>(pb); // No compiler error.
  // A reinterpret_cast<> can only be justified in rare situations.
  // When you use a reinterpret_cast<> you tell the compiler that you
  // need a non-usual type of casting and that you know what you are doing...