Zack Austin

CS 3370 Test1 Cue Sheet

Pointers and const:

Const before the asterisk ⇒ const contents “p is a pointer to a const int”

Const after the asterisk ⇒ const pointer “p is a const pointer to an int”

Casts:

static\_cast: converts among “related types” (e.g., numbers), restore a pointer from a void\*

dynamic\_cast: for “downcast” tests in class hierarchies

reinterpret\_cast: for low-level tomfoolery, e.g., casting from int to pointer, casting among unrelated pointer types (see Program 1!)

const\_cast (to remove const; rarely used)

Pointers and Arrays:

Array names in expressions “decay” into a pointer to the 1st element.

• a is the same as &a[0] • ⇒ \*a == a[0] •⇒ a + i == &a[i] • ⇒ \*(a + i) == a[i] • \*(p + i) == p[i] // \*\*\* for any pointer \*\*\*

Allocating multi-dimensional arrays on the heap:

\*(a+1)[2] == \*(\*(a+1)+2). Int (\*p)[3] = new int[2][3]. Int a[][3][4] {{{1,2,3,4},{5,6,7,8},{9,0,1,2}},{{3,4,5,6},{7,8,9,0},{1,2,3,4}}}; Int (\*p)[3][4] = a;

References, reference parameters:

refs are aliases: int i = 7; int& r = i; ++r; cout << i << endl; // 8

Ref params are like implicit pointer indirections. Transparent access to arguments. Returns must be via objects still scoped.

A reference is like a pointer that applies & and \* automatically when needed. It can be used on either side of an assignment.

On the left, it is an lvalue, Writes to memory (uses operator=). On the right it is an rvalue, Reads from memory (uses converts to Data).

Lvalue an address, rvalue values from an address. (Almost) Always pass objects by (const) reference as they give efficiency of refs and safety of const.

New, delete, and placement new operators, and their relationship with the functions operator new (size\_t) and operator delete (void\*):

New operator:

Does the following before returning a pointer: Allocates needed memory on the heap, calls the library function operator new ( ), Initializes the object by calling the proper constructor.

Delete operator:

Calls the destructor for the object, Returns the memory to the free store, via the library function void operator delete (void\*).

Placement New:

A special-purpose version of the new operator used by library developers that does in-place initialization. No memory is allocated!

e.g., to poke a value at a given memory address.

Semantics of post/pre-increment/decrement operators:

Unary operators. Post is higher precedence than pre! Pre returns an lvalue. ++ ++x is legal.

Post return an rvalue. x++ ++ is not legal. ++x++ is not legal.

Consider x = \*p++; ++ executes first but it yields the un-incremented val, then \* executes after the assignment, p is incremented as a side effect.

Static class data members, local static data in functions:

Function parameters and local variables have function scope: they are visible only inside the function body. But lifetime is a separate matter. Dependent on storage class.

Storage Class: automatic: Live and die with the function. Static: Alive for the entire program. Dynamic: Live on heap; programmer controlled.

Pointers to function:

Most useful when passing functions as arguments. Functions cannot be passed by value (duh). When you pass a function as an argument, a “pointer” is passed. Int extremum(int\* a, int n, int f(int,int)) {} // Or, int (\*f)(int,int).

int a[] = {1,2,3,4,5};

cout << extremum(a,5,mymax) << endl; // 5

int (\*pf1)(int,int) = &mymin; // & optional

cout << extremum(a,5,pf1) << endl; // 1

auto pf2 = mymin; // & optional (missing here)

cout << extremum(a,5,pf2) << endl; // 1

decltype(mymin) \*pf3 = mymin; // \* NOT optional in a declaration; pf3 is a pointer to a function

Pointer arithmetic:

char s[] = "desolate", \*p = s;

cout << \*p++ << endl; //d cout << \*(p++) << endl; //e cout << (\*p)++ << endl; //s cout << \*++p << endl; //o

cout << \*(++p) << endl; //l cout << ++\*p << endl; //m cout << ++(\*p) << endl; //n cout << s << endl; //detonate

int a[] = {10,15,4,25,3,-4}; int \*p = &a[2];

cout << \*(p+1) << endl; // 25 cout << p[-1] << endl; // 15 cout << p - a << endl; // 2 cout << a[\*p++] << endl; // 3

cout << \*(a+a[2]) << endl; // 3

Bitwise operations, sign extension with right shift:

<<Shift left, >>shift right, |bitwise OR, &bitwise AND, ^bitwise XOR (exclusive-or), ~bitwise NOT (flip all bits).

ANDing a bit with a 1 reads it (b & 1 == b). ORing a bit with 1 sets it to 1 (b | 1 == 1). ANDing a bit with 0 resets it (b & 0 == 0).

XORing a bit with 1 flips it (b ^ 1 == ~b). ORing a bit with 0 is a nop (b | 0 == b). ANDing a bit with 1 is a nop (b & 1 == b). XORing a bit with 0 is a nop (b ^ 0 == b).

The value of a right-shift expression x >> y is x / 2y, and the value of a left-shift expression x << y is x \* 2y. The right-shift operator causes the bit pattern in the first operand to be shifted to the right by the number of bits specified by the second operand.

Float x = 6.5; unsigned int n = \*reinterpret\_cast<unsigned int\*>(&x);

Cout << hex << n << endl; // 40d00000

Int sb = n >> 31; cout << sb << endl; // 0

Unsigned int expmask = ((1u << 8) - 1) << 23; cout << expmask << endl; // 7f800000

Int exp = (n & expmask) >> 23; cout << dec << exp << endl; // 129

Cout << ((n << 1) >> 24) << endl; // 129 (another way)

Int mask4 = 1u << 4; n |= mask4; // Set bit 4 cout << hex << n << endl; // 40d00010

n &= ~mask4; // Clear it cout << hex << n << endl; // 40d00000.