Objects, Types, and Values

Chapter 3

Storage of Data

The “Hello, World!” program is boring.

To do tasks that are not boring, we need to be able to store data.

We need somewhere in the computer’s memory to place values we read from the user

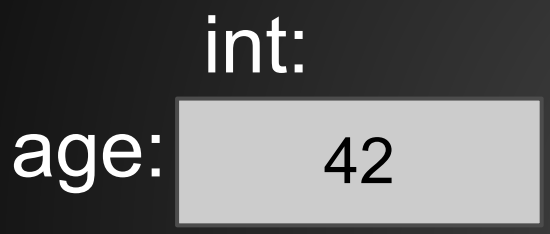
Objects

An *object* is a region of memory with a *type* that specifies what kinds of information can be placed in it. p60

A named object is called a *variable*.

For example, character strings are put into **string** variables and integers are put into **int** variables.

A picture



This would represent an object of type **int** named **age** containing the integer value **42**.

Example Code

```
#include <iostream>
using std::cin;
using std::cout;
int main()
{
    cout << “Please enter your first name (followed by ‘enter’):\n”;
    string first_name;
    cin >> first_name;
    cout << “Hello, ” << first_name << “!\n”;
    return 0;
}
```

What's it do?

The first line of main is a message that prompts the user for some input. It's usually called a prompt.

The next line defines a variable of type **string** called **first_name**

The 3rd line reads from the keyboard and stores the result in first_name.
The last line prints out the name in a message.

Definitions

The line: string first_name is a variable definition.

It sets aside an area of memory for holding strings of characters and gives it a name.

In general a *definition* of a variable introduces a new name into a program and sets aside memory for it.

See-in

cin >> first_name;

reads characters from the keyboard and stores them in the variable first_name.

How does it know when to stop?

Repetition

We could have written the last line like this:

```
cout << “Hello”;
cout << first_name;
cout << “!\n”;
```

Needless repetition because it provides opportunities for errors should be avoided.

Variables

"We can do nothing of interest with a computer without storing data in memory" p62

The places in which we store data are called *objects*.

To access an object we need a *name.*

A named object is called a ***variable*** and has a specific *type* that determines what can be put into the object and which operations can be applied

More variables

The data items we put in our variables are called *values*.

A statement that defines a variables is called a *definition*.

A definition can and usually should provide an initial value.

```
string name = “Annemarie”;
int number_of_steps = 39;
```

Variables 3

You can’t put the wrong type into a variable:

```
string name2 = 99;
int number_of_steps = “Annemarie”;
```

The compiler will check each variable and the type of data you are attempting to store in them.

Sample Types

```
int number_of_steps = 39; //int for integers
double flying_time = 3.5; //double for floating pt
char decimal_point = '.'; //char for 1 character
string name = "Kelly"; //string for char strings
bool tap_on = true; //bool for logical variables
```

Input and Type

The input operator >> (“get from”) is sensitive to type; that is it reads according to the type of variable you read into.

```
int main()
{
    cout << “Please enter your first name and
age\n”;
    string first_name;   //string variable
    int age;             //integer variable
    cin >> first_name;  //read a string
    cin >> age;          //read an integer
    cout << “Hello, ” << first_name << “ (age ” <<
age << “)\n”;
    return 0;
}
```

More input and Types

If we type in Carlos 22, we get out
Hello, Carlos (age 22)
How does that work?

What if we type in 22 Carlos?

Operations and operators

In addition to specifying what values can be stored in a variable, the type of a variable determines what operations we can apply to it and what they mean.

```
int count;
cin >> count;
string name;
cin >> name;

int c2 = count+2;
```

```
string s2=name +“Jr.”;

int c3 = count - 2;

string s3=name - “Jr”;
```

Memorize this. (no please don’t)

	bool	char	int	double	string
assignment	=	=	=	=	=
addition			+	+	
concatenation					+
subtraction			-	-	
multiplication			*	*	
division			/	/	
remainder			%		
increment by 1			++	++	
decrement by 1			--	--	
increment by **n**			+= n	+= n	

	bool	char	int	double	string
add to end					+=
decrement by n			-= n	-= n	
multiply and assign			*= n	*= n	
divide and assign			/= n	/= n	
remainder and assign			%= n		
read from s into x	s >> x	s >> x	s >> x	s >> x	s >> x
write x to s	s << x	s << x	s << x	s << x	s << x
equals	==	==	==	==	==
not equals	!=	!=	!=	!=	!=
greater than	>	>	>	>	>
greater than or equal	>=	>=	>=	>=	>=
less than	<	<	<	<	<
less than or equal	<=	<=	<=	<=	<=

Other operations

There are many floating point operations that we do not have operators for.

For example, square root. We do have functions that we can use and call to do these operations.

String have fewer operators but many named operations as we'll see later.

String operators

+ for strings means concatenation.
Concatenation means to join two string.
For example:
string s1 = “Dave”;
string s2 = “McPherson”;
s1+s2 would be “DaveMcPherson”

```
int main()
{
    cout << “Please enter two names\n”;
    string first;
    string second;
    cin >> first >> second;
    if ( first == second )
        cout << “that’s the same name twice\n”;
    if ( first < second )
        cout << first << “ is alphabetically before “ << second
<< “\n”;
    if ( first > second )
        cout << first << “ is alphabetically after “ << second
<< “\n”;
}
```

Assignment and initialization

```
int a = 3; //a starts out with value 3

a = 4; // a gets the value 4 (becomes 4)

int b = a; //b starts out with a copy of a's value

b = a+5; //b gets the value a+5

a = a + 7; //a gets the value a+7
```

What's going on there?

Take a look at that last one again.

Clearly, = does not mean equality.

= means assignment, that is, to place a new value in a variable.

a = a + 7 means this:

1. Get the value of a; that's integer 4.
2. Next, add 7 to that 4, yielding the integer 11.
3. Finally, put that 11 into a.

So, assignment and initialization

Two slides back we used “starts out with” and “get” to distinguish between two similar, but logically distinct operations.

Initialization (giving a variable its initial value)

Assignment (giving a variable a new value)

These operations are so similar, C++ uses the same notation, =

How can you tell them apart?

I’m glad you asked.

You can tell the two apart by the type specification (like int or string) that always starts an initialization.

An assignment does not have that.

Initialization always finds the variable empty.

Assignment does not. Assignment has to “clean up” or “empty” the memory before it uses it.

An Example

```
int main()
{
    string previous = “”;                // line 1
    string current;                      // line 2
    while ( cin >> current ) {           // line 3
        if ( previous == current )       // line 4
            cout << “repeated word: ” << current << “\n”;
        previous = current;
    }
    return 0;
}
```

Explanation

Line 1: Initialization of previous to “not a word”
Line 2: current word
Line 3: read a stream of words
Line 4: check if word is the same as last

We’ll come to the while statement soon enough.

Composite assignment operators

There are lots of shortcut notations in C++.
As you gain more experience give them a try.
For example:

```
int x = 0;
x++; //means to add 1 to x.
x += 1; //means to add 1 to x.
x = x + 1; //means to add 1 to x.
```

All three are the same meaning, but all are written differently.

Names

We name our variables so that we can remember them and refer to them from other parts of a program. p74

In a C++ program, a name starts with a letter* and contains only letters, digits and underscores.

*They can also start with underscores, but don't do this.

Examples

Good	Bad
x	2x
number_of_elements	timetomarket
FourierTransform	Start menu
z2	
Polygon	

NaMiNg

Names are case sensitive

```
int Main()
{
    STRING s = “Goodbye, cruel world!”;
    cOut << S << ‘\n’;
    Return 0;
}
```

Keywords

C++ has 70 keywords.
Quiz: name them...no.

You can’t use keywords as names.
You can use other names, like string, but it’s really not a good idea.

Choosing names

When you choose a name for your variables, functions, types, etc., choose meaningful names; that is choose names that will help people understand your program. p76

It is difficult to understand programs that are littered with easy to type names, e.g. k2, x1, y.

Don’t choose overly long names: thisIsTheLongestNameThatICouldFitOnThisLin

Types and objects

The notion of type is central to C++

A *type* defines a set of possible values and a set of operations (for an object).

An *object* is some memory that holds a value of a given type.

A *value* is a set of bits in memory interpreted according to a type.

A *variable* is a named object.

A *declaration* is a statement that gives a name to an object.

A definition is a declaration that sets aside memory for an object.

Type safety

Use of uninitialized variables is a very common unsafe type use.

An implementation is even allowed to give a hardware error when the unitialized variable is used.

Always initialize your variables!

Safe conversions

A safe conversion is one where the original value can be retrieved from the converted value.

For example

char c = ‘x’;

int i1 = c;

int i2 = ‘x’;

i1 and i2 both get the value 120.

char c2 = i1;

cout << c << ‘ ‘ << i1 << ‘ ‘ << c2 << ‘\n’;

x 120 x

Safe because the conversion back from int to char retains value for character

Other safe conversion

bool to char
bool to int
bool to double
char to int
char to double
int to double
All of these conversion “widen” the value, or at least don’t lose precision or information.

Unsafe conversions

double to int
double to char
double to bool
int to char
These conversion “narrow” to lose information.
For example, doubles are stored usually in 8 bytes, but ints are only stored in 4.