**Указатели**

Три фундаментальных свойства для программы.

* где хранится информация;
* какое значение сохранено;
* разновидность сохраненной информации.

если home — переменная, то &home — ее адрес.

**Листинг 4.14. address. срр**

*// address .'срр -- использование операции & для нахождения адреса*

*#include <iostream>*

*int main()*

*{*

*using namespace std;*

*int donuts = 6;*

*double cups = 4.5;*

*cout « "donuts value = " « donuts;*

*cout « " and donuts address = " « Sdonuts « endl;*

*// ПРИМЕЧАНИЕ: может понадобиться использовать*

*// unsigned (Sdonuts) и unsigned (&cups)*

*cout « "cups value = " « cups;*

*cout « " and cups address = " « &cups « endl;*

*return 0;*

*}*

Ниже показан вывод программы из листинга 4.14 в одной из систем:

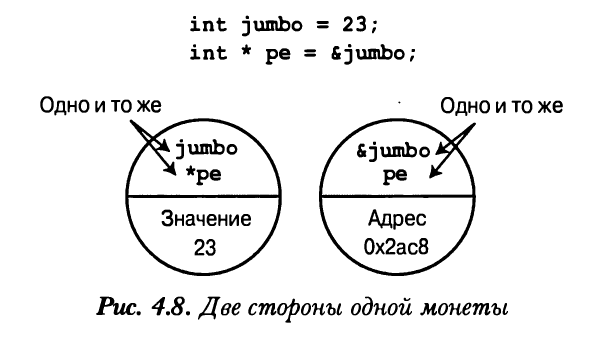
*donuts value = 6 and donuts address = 0x0065fd40*

*cups value = 4.5 and cups address = 0x0065fd44*

Новая стратегия хранения данных изменяет трактовку местоположения как именованной величины, а значения — как производной величины. Для этого предусмотрен специальный тип переменной — ***указатель***, который может хранить адрес значения. Таким образом, имя указателя представляет местоположение. Применяя операцию \*, называемую ***косвенным значением*** или операцией ***разыменования***, можно получить значение, хранящееся в указанном месте.

**Листинг 4.15. pointer. ерр**

*// pointer.ерр -- наша первая переменная-указатель*

*#include <iostream>*

*int main()*

*{*

*using namespace std;*

*int updates =6; // объявление переменной*

*int \* p\_updates; // объявление указателя на int*

*p\_updates = &updates; // присвоить адрес int указателю*

*// Выразить значения двумя способами*

*cout « "Values: updates = " « updates;*

*cout << ", \*p\_updates = " << \*p\_updates << endl;*

*// Выразить адреса двумя способами*

*cout « "Addresses: &updates = " « &updates;*

*cout << ", p\_updates = " << p\_updates << endl;*

*// Изменить значение через указатель*

*\*p\_updates = \*p\_updates + 1;*

*cout << "Now updates = " << updates << endl;*

*return 0;*

*}*

Ниже показан пример выполнения программы из листинга 4.15:

Values: updates = 6, \*p\_updates = 6

Addresses: &updates = 0x0065fd48, p\_updates = 0x0065fd48

Now updates = 7

**Объявление и инициализация указателей**

Компьютеру нужно отслеживать тип значения, на которое ссылается указатель. Например, адрес ***char*** обычно

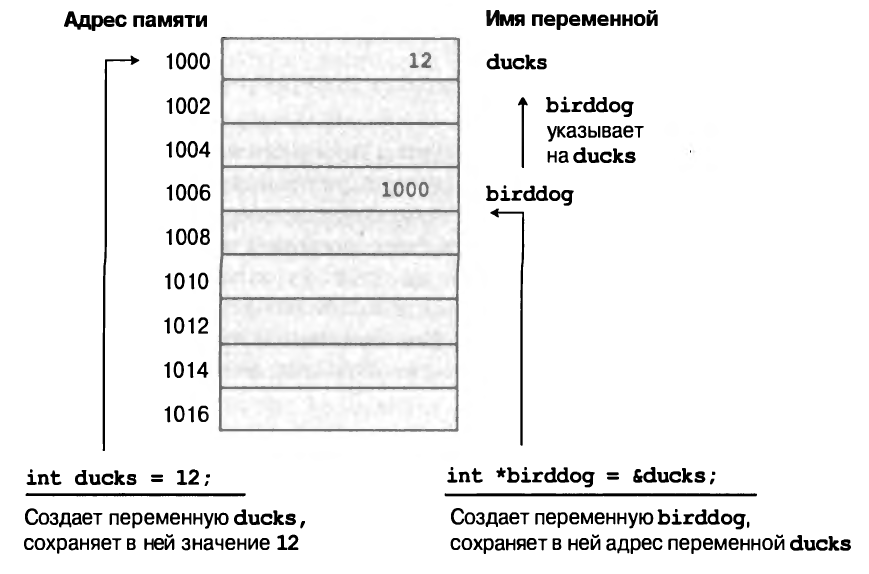
выглядит точно так же, как и адрес ***double***, но ***char*** и ***double*** использует разное количество байт и разный внутренний формат представления значений. Поэтому объявление указателя должно задавать тип данных указываемого значения.

Например, предыдущий пример содержит следующее объявление:

*int \* p\_updates;*

**На заметку!**

В языке C++ комбинация int \* представляет составной тип "указатель на int".



*double \* tax\_ptr; // tax\_ptr указывает на тип double*

*char \* str; // str указывает на тип char*

**Листинг 4.16. init\_ptr. срр .**

*// init\_ptr.cpp — инициализация указателя*

*#include <iostream>*

*int main ()*

*{*

*using namespace std;*

*int higgens = 5;*

*int \* pt = Shiggens;*

*cout « "Value of higgens = " « higgens*

*« "; Address of higgens = " « Shiggens « endl;*

*cout « "Value of \*pt = " « \*pt*

*« "; Value of pt = " « pt « endl;*

*return 0;*

*}*

**Опасность, связанная с указателями**

Опасность подстерегает тех, кто использует указатели неосмотрительно. Очень важно понять, что при создании указателя в коде C++ компьютер выделяет память для хранения адреса, но не выделяет памяти для хранения данных, на которые указывает этот адрес. Выделение места для данных требует отдельного шага. Если пропустить этот шаг, как в следующем фрагменте, то это обеспечит прямой путь к проблемам:

*long \* fellow; // создать указатель на long*

*\*fellow = 223323; // поместить значение в неизвестное место*

Внимание!

Золотое правило указателей: всегда инициализируйте указатель, чтобы определить точный

и правильный адрес, прежде чем применять к нему операцию разыменования (\*).

**Указатели и числа**

Указатели — это не целочисленные типы, даже несмотря на то, что компьютеры обычно выражают адреса целыми числами. Концептуально указатели представляют собой типы, отличные от целочисленных. Целочисленные значения можно суммировать, вычитать, умножать, делить и т.д. Но указатели описывают местоположение, и не имеет смысла, например, перемножать между собой два местоположения. В терминах допустимых над ними операций указатели и целочисленные типы отличаются друг от друга. Следовательно, нельзя просто присвоить целочисленное значение

указателю:

*int \* pt;*

*pt = 0хВ8000000; // несоответствие типов*

int \* pt;

pt = (int \*) 0xB8000000; // теперь типы соответствуют

**Выделение памяти с помощью операции new**

До сих пор мы инициализировали указатели адресами переменных; переменные — это именованная память, выделенная во время компиляции, и каждый указатель, до сих пор использованный в примерах, просто

представлял собой псевдоним для памяти, доступ к которой и так был возможен по именам переменных. Реальная ценность указателей проявляется тогда, когда во время выполнения выделяются неименованные области памяти для хранения значений.

*int \* pn = new int;*

Часть new int сообщает программе, что требуется некоторое новое хранилище, подходящее для хранения int. Операция new использует тип для того, чтобы определить, сколько байт необходимо выделить. Затем она находит память и возвращает адрес. Далее вы присваиваете адрес переменной рп, которая объявлена как указатель на int. Теперь рп — адрес, а \*рn— значение, хранящееся по этому адресу. Сравните это с присваиванием адреса переменной указателю:

int higgens;

int \* pt = &higgens;

**Общая форма** получения и назначения памяти отдельному объекту данных, который может быть как структурой, так и **фундаментальным** типом, выглядит **следующим образом:**

*имяТипа \* имя\_указателя = new имяТипа;*

**Листинг 4.17. use\_new. срр**

*// use\_new.cpp -- использование операции new*

*#include <iostream>*

*int main()*

*{*

*using namespace std;*

*int nights = 1001;*

*int \* pt = new int; // выделение пространства для int*

*\*pt = 1001; // сохранение в нем значения*

*cout << "nights value = "; // значение nights*

*cout << nights << " : location " << Snights << endl; // расположение nights*

*cout « "int "; // значение и расположение int*

*cout << "value = " << \*pt << " : location = " << pt << endl;*

*double \* pd = new double; // выделение пространства для double*

*\*pd = 10000001.0; // сохранение в нем значения double*

*cout « "double ";*

*cout << "value = " « \*pd << ": location = " « pd «*

*// значение и расположение double*

*cout << "location of pointer pd: " << &pd << endl;*

*// расположение указателя pd*

*cout << "size of pt = " << sizeof(pt);*

*cout « " : size of \*pt = " « sizeof(\*pt) « endl;*

*cout « "size of pd = " « sizeof pd;*

*cout « ": size of \*pd = " « sizeof (\*pd) « endl;*

*return 0;*

*}*

Ниже показан вывод программы из листинга 4.17:

*nights value = 1001: location 0028F7F8*

*int value = 1001: location = 00033A98*

*double value = le+007: location = 000339B8*

*location of pointer pd: 0028F7FC*

*size of pt = 4: size of \*pt = 4*

*size of pd = 4: size of \*pd = 8*

Нехватка памяти?

Может случиться так, что у компьютера не окажется достаточно доступной памяти, чтобы ж удовлетворить запрос new. Когда такое происходит, операция new обычно реагирует ге- щ нерацией исключения; способы обработки ошибок рассматриваются в главе 15. В более 1 старых реализациях new возвращает значение 0. В C++ указатель со значением о называется null-указателем (нулевым указателем). C++ ! ! гарантирует, что нулевой указатель никогда не указывает на допустимые данные, поэтому И он часто используется в качестве признака неудачного завершения операций или функций, % которые в противном случае должны возвращать корректные указатели. Оператор if, описанный в главе 6, поможет справиться с подобной ситуацией. А сейчас просто важно знать, и что в C++ предлагаются инструменты для обнаружения и реагирования на сбои при выделении памяти.

**Освобождение памяти с помощью операции delete**

Использование операции new для запрашивания памяти, когда она нужна — одна из сторон пакета управления памятью C++. Второй стороной является операция delete, которая позволяет вернуть память в пул свободной памяти, когда работа с ней завершена. Это — важный шаг к максимально эффективному использованию памяти. Память, которую вы возвращаете, или освобождаете, затем может быть повторно использована другими частями программы. Операция delete применяется с указателем на блок памяти, который был выделен операцией new:

*int \* ps = new int; // выделить память с помощью операции new*

*... // использовать память*

*delete ps; // по завершении освободить память*

*// с помощью операции delete*

*int \* ps = new int; // нормально*

*delete ps; // нормально*

*delete ps; // теперь- не нормально!*

*int jugs = 5; // нормально*

*int \* pi = &jugs; // нормально*

*delete pi; // не допускается, память не была выделена new*

Внимание!

Операция delete должна использоваться только для освобождения памяти, выделенной с помощью new. Однако применение delete к нулевому указателю вполне безопасно.